Execution of LOTOS specifications in a distributed environment[1]

Cheng Wu, Gregor v. Bochmann, Omar Bellal, Qiang Gao
Département d'informatique et de recherche operationnelle
Université de Montréal
Montréal, P.Q., Canada

Abstract

Prototyping techniques play an important role in the software development process; they allow system designers to analyze a system in an early development stage. In this paper, we present a model of distributed implementation of LOTOS specification. More precisely, we present an approach for executing a distributed system, specified in LOTOS, on a fixed number of computers, each implementing a part of the specified system. Our model consists of three functional parts, the first handles the behavior of LOTOS specifications and the other two deal with synchronization. For the first part, we use a set of trees to represent the parts of the specification which are executed at different sites. A tree reflects the dynamic relationship between active processes within a local system in view of their (local) synchronization. The second part deals with the dynamic relations for synchronization involving several interrelated sites. We use (virtual) rings to represent rendezvous relation among the trees on different sites. An algorithm for establishing these rings during the execution of a specification is presented. Based on these rings, the third part implements a distributed multi-way rendezvous algorithm, which provides the rendezvous synchrnoization required by the LOTOS semantics. This approach to the distributed implementation of LOTOS is symmetric, in the sense that all participating sites have equal roles. The paper also describes the implementation of this approach within a network of workstations.

1. Introduction

---

[1] This work was performed within the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols and was also supported by the Natural Sciences and Engineering Research Council of Canada under a strategic research grant, and the Ministry of Education of Quebec.

LOTOS [ISO 89][Bolo 87] is a formal description technique (FDT) which is standardized within International Standard Organization (ISO); it is intended for writing formal specifications of communication protocols and services of Open System Interconnection (OSI). It is also applicable to distributed systems. LOTOS is designed as an executable specification language. Execution of a specification plays an important role in the development process of software [Turn 89]. Some interpreters (or simulators) have been developed (e.g.[Bria 86] [Logr 88]) . They allow users to simulate the execution of a LOTOS specification and check whether it behaves correctly. However, it would be desirable to prototype a distributed system specified in LOTOS in a real distributed environment. This may allow users to analyze the system in an early development stage. However, little work has been done about execution of LOTOS specifications in a distributed environment.

In this paper, we address the question of distributed implementation of LOTOS specifications. We mean by this an implementation of a distributed system specified in LOTOS involving a fixed number of sites and each implementing a part of the specified system. We assume that the different sites communicate by passing message through an underlying reliable communication medium. Two issues are particularly critical for such a distributed implementation. One is distributed implementation of multiple rendezvous, that is an algorithm or a protocol which is used to synchronize the LOTOS processes in a distributed environment. This is a fundamental issue in distributed computing. The other is the distributed execution model for LOTOS which allows a specification to be partitioned into more than one part, where each part is processed at a separate site, and links are established among these parts for synchronization.

Certain features of the LOTOS language make the above issues difficult to realize. In LOTOS, multi-way rendezvous is allowed and processes communicate through gates, thus a process may not know the other processes which synchronize on the same gate. LOTOS also allows dynamic creation of processes and gates, and the number of processes involved in a rendezvous at a given gate may change dynamically from one occurrence of a rendezvous to another. These features of LOTOS make a distributed implementation of LOTOS specifications difficult.

The question of execution of LOTOS specification has been studied for several years. Some approaches presented in the literature try to directly translate LOTOS specifications into executable high level programming languages (e.g. Prolog [Logr 88], Parlog [Gilb 89]). Then the executions of LOTOS specifications depends on the

underlying high level programing languages. Some other works transform LOTOS specifications into abstract execution models, such as (finite) automata and trees, and later translate them into a system implement language such as C.

Automata (finite) is a well accepted execution model. A translation of a subset of LOTOS to EFSM (Extended Finite State Machine) is described in [Karj 88]. It cannot support distributed implementation because a LOTOS specification is represented by a single state machine. This approach also suffers state explosion. The work presented in [Dubu 89] tries to solve the state explosion problem by transforming a subset of LOTOS to multi-automata with ports: each automata represents a LOTOS process and ports represent the relation among LOTOS processes. This approach does support distributed implementation but the concept of ports used in this context is not suitable for the distributed control of synchronization.

Tree-oriented execution models were proposed in [Boch 89c] [Wu 90] [Nomu 90] [Sist 91]. In these models, tree structures represent parent-children process relationships in order to realize multi-way synchronization. The work of [Nomu 90] only considered a centralized implementation. The formalism of [Wu 90] supports non well-guarded behavior expressions, and allows for parallel processing in loosely as well as closely coupled systems. The work of [Sist 91] considers the distributed implementation, where processes are organized in a hierarchical topology and communicate with each other by message transfers. Our earlier work [Boch 89c] presented an algorithm to establish virtual rings which connect nodes of the tree. Rings represented the rendezvous relations among LOTOS processes and were used to synchronize distributed processes by a distributed synchronization algorithm [Gao 89].

Distributed multi-way rendezvous is another important issue. In multi-way rendezvous, more than two processes may synchronize in a single rendezvous. The problem of implementing multiple rendezvous captures two fundamental issues in distributed computing: mutual exclusion and synchronization. That is, a rendezvous algorithm makes sure that a rendezvous can only be executed if all the processes relating to it are ready to execute it (synchronization) and a process may be ready to execute more than one rendezvous for a given state, but it executes no more than one at a time (mutual exclusion).

It is not trivial to implement multi-way rendezvous in a distributed system where processes communicate with each other via asynchronous message transfers. The following properties are desirable:

1) A fully distributed and symmetric solution. That is, there is no central memory nor any pre-defined distinguished process that makes all the decisions; all the processes are initially indistinguishable except that they have their own process identifier.

2) A fair solution. That is, if an interaction is enabled infinitely often, it will be executed eventually.

3) The number of messages needed per interaction should be bounded.

4) A dynamic solution for the distribution of processes. That is, during the execution, the number of processes associated to a rendezvous may change and processes are dynamically distributed onto different machines.

The problem has been under study for a long time. A survey of this topic can be found in [Levy 88], and other algorithms can be found in [Rame 87] [Bagr87] [Gao 89] [Kuma 90] [Atti 90]. Except the one presented in [Gao 89], non of the others satisfies all the properties mentioned above. The algorithm of [Gao 89] uses ring structures (ring structures are also used in [Bagr 87] [Kurm 90]). A ring represents a rendezvous relation and processes which are connected by a ring synchronize for a rendezvous. The algorithm passes messages along rings to fulfill synchronization and mutual exclusion, and it also allows new processes to be dynamically inserted into rings.

In this paper, we mainly address the question of the distributed LOTOS execution model. In our model, a specification is represented by a so-called activity tree. The tree reflects the dynamic relationships between the active LOTOS processes within the system. The tree can be partitioned into sub-trees which represent the parts of the LOTOS specification (or LOTOS processes) which are executed in different sites. Then an algorithm is presented to dynamically establish rings among these sub-trees (and not among nodes of the tree as in [Boch 89c]). Rings correspond to gates of LOTOS and they represent rendezvous relations. Distributed rendezvous algorithms based on ring structures such as the one in [Gao 89] can be used to implement the synchronization between sub-trees.

Our model has several features. First, in contrast to [Sist 91], our model is a symmetric solution for the distributed implementation, that is, sub-trees in the system are treated equally. Second, our model consists of two functional parts. The first handles the behavior of LOTOS specifications within a single sub-tree and the second deals with

synchronization between the different sub-trees. There is a clear and simple interface between the two parts. Different LOTOS interpreters may be used for the first part. Third, rings represent rendezvous relations among sub-trees in our model. There are several distributed synchronization algorithms based on rings mentioned above which may be used for the distributed implementation. Fourth, the distributions of sub-trees over different processing sites may be done at run-time.

The paper is organized as the following. We give an overview of our approach in Section 2. We define a general LOTOS execution model which is based on a so-called activity tree in Section 3. In Section 4, we adapt this general model for the distributed implementation. We address questions such as the partitioning the activity tree into sub-trees and establishing rings among sub-trees. A pilot implementation is described in Section 5. Finally, Section 6 contains conclusions.

2. System overview

In this section, we give an overview of our distributed system for the execution of LOTOS specifications, as shown in Figure 1(a). The implementation environment involves a fixed number of sites (computers) which are connected by a transport service, which provides reliable end-to-end communication. A LOTOS specification is partitioned into sub-specifications and they are executed in different sites. At each site, there is a Local LOTOS Execution Engine (LLEE) component and a Synchronization Entity (SE) component. The allocation of the different parts of the LOTOS specification to the different sites is not discussed in detail in this paper. We either assume static allocation, or a single Allocation Management component which makes these allocation decisions dynamically.

2.1. Local LOTOS Execution Engine (LLEE)

After being allocated to a given site, a LOTOS sub-specification is handled by the LLEE component of that site. The structure of the LLEE is shown in Figure 1(b). It contains two main sub-components: LOTOS Execution (LE) and Distributed Coordination (DC). The function of the LE is the same as an ordinary LOTOS interpreter such as [Logr 88]. It calculates the possible actions of the LOTOS sub-specification. There are two types of actions. An action is called a local action if it only involves processes of the local sub-specification. An action is called a global action if it involves processes of the local sub-specification as well as processes of the sub-specifications handled in other sites. After

executing an action (local or global), the LE changes its local state as implied by the execution of the action. An LE consists of two parts: a Control Part which handles the control part of the LOTOS behavior, and a Data Part which handles the data part of the LOTOS behavior.

The DC consists of three procedures: Distribution, Ring Establishment and Local Rendezvous Manager (LRM) . The function of the Distribution procedure is to transfer a sub-specification, which is to be executed as a separate sub-tree on a different site. The function of the Ring Establishment procedure is to establish rings for rendezvous when the system transfers sub-specifications. The LRM does two things: It chooses a local action for execution, or it communicates with the SE component for the execution of a global action. For the latter, the LRM provides to the SE a list of actions which are locally possible and need to rendezvous with other sub-specifications. Each action in the list is associated with a ring identifier to which the action relates.

2.2. Synchronization Entity (SE)

The function of the SE's is to implement global synchronization involving more than one sub-specification. After receiving a list of possible actions, a SE communicates with the other SE's in the system through the underlying Transport Service. As mentioned in Section 1, there are several distributed synchronization algorithms which work in this context.

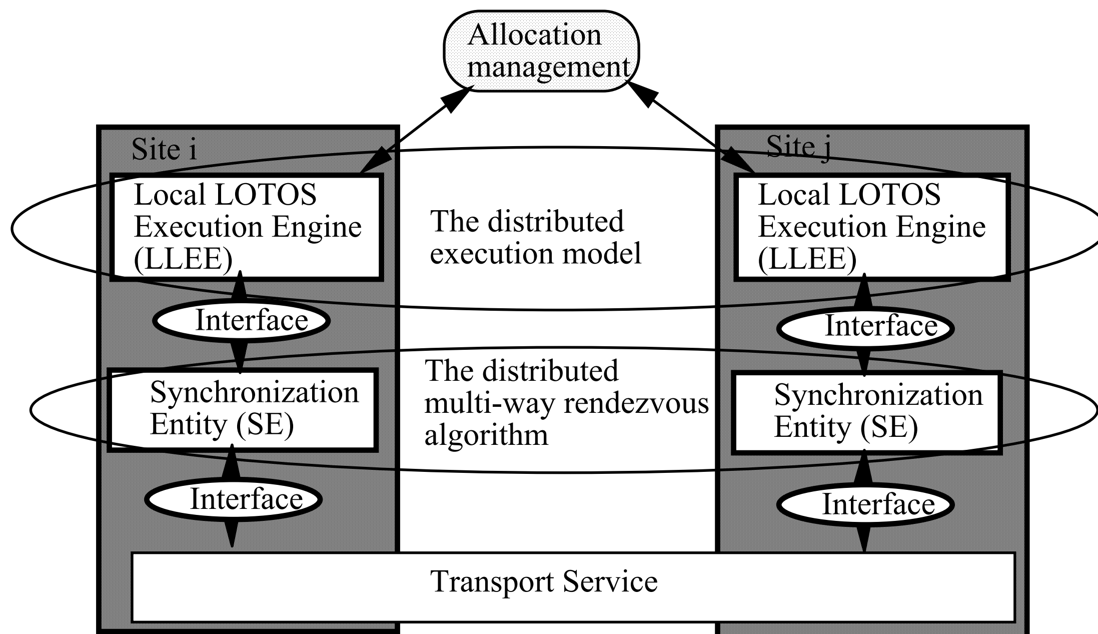
The algorithm of [Gao 89] which we have implemented proceeds along the following lines: The SE sends out a so-called Matching Detection message (MD) for one of the possible action it selects. The MD goes around the related ring to determine whether all connected SE's are ready to participate in a rendezvous. If they are ready, a so-called Matching message (M) is circulated around the ring to establish the rendezvous. Otherwise the SE, which initiated the MD message, sends out ASK messages to those SE's which did not agree to participate in the rendezvous. The result of the negotiation is either a rendezvous (circulation of the M message) or an abandon of the attempted rendezvous, indicated by the circulation of a so-called No Matching message (NM). The algorithm ensures that a rendezvous for a global action will be found if such a rendezvous is possible (see [Gao 89] for more details).

When a rendezvous has been selected, the SE sends the action with associated new parameter values and predicates (which are obtained through the synchronization algorithm) to the local LLEE for execution.

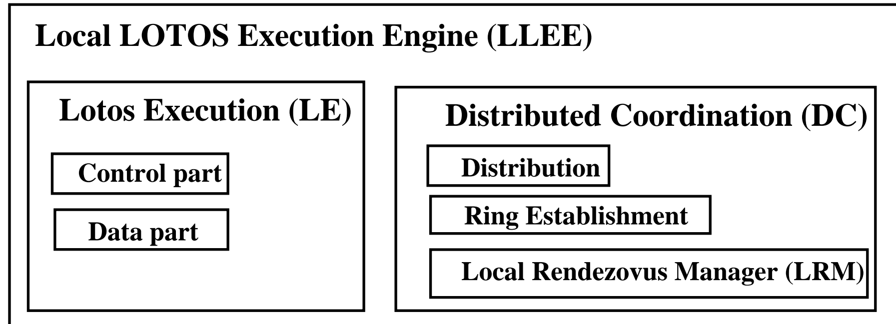
2.3. Allocation Management

Allocation management is a fundamental issue in distributed computing. It involves two basic questions: how to partition a system into sub-systems and how to distribute these sub-systems onto different sites.

Our system distributes sub-specifications in two modes: static and dynamic. In the static mode, a user has to indicate, in his LOTOS specification, the specific site where a sub-specification is executed. In the dynamic mode, the user only indicates which sub-specifications can be moved out from the local site to be executed in any other site of the system, and it is the Allocation Management component which does the distribution at run-time. In this mode, a sub-specification may be kept in the original site if no other sites are available.



(a) Overall system structure



(b) Structure of the Local LOTOS Execution Engine (LLEE)

Figure 1: Overall system structure

3. A general LOTOS execution model

In this section, we present a general execution model for LOTOS specifications. The model will be modified for distributed implementation in the next section. The execution model is based on an activity tree with attributes. The activity tree reflects the dynamic relations between the process invocations and activations of behavior expressions in the specified system; the functions related to the attributes control the execution of interactions. During the execution, the tree is dynamically changed, which reflects the dynamic change of the behavior of the specified system.

Figure 2 shows a LOTOS specification of an 'Example' system. After dropping money into a vending machine (VM), a boy may obtain a candy, if the latter is not eaten by one of the two little devils that are included in the VM ('m' denotes 'money', 'c' denotes 'candy', and 'e' denotes 'eat_candy').

```
Specification Example:noexit
        hide m, c in
        Boy[m, c] || VM[m, c]
        where
        process Boy[m, c]:noexit:=
          m; ( c; Boy[m, c]
                []
                Boy[m, c])
         endproc
        process VM[m, c]:noexit:=
         hide e in
         Machine[m, c, e]
         | [e] |
         (Devil[e] ||| Devil[e])
         where
        process Machine[m, c, e]:noexit:=
         m; ( c; Machine[m, c, e]
                []
                e; Machine[m, c, e] )
```

```
            endproc
            process Devil[e]:noexit:=
             e; Devil[e]
            endproc
            endproc
endspec
```

Figure 2: Example: Boy and Vending Machine (VM)

3.1. Activity tree

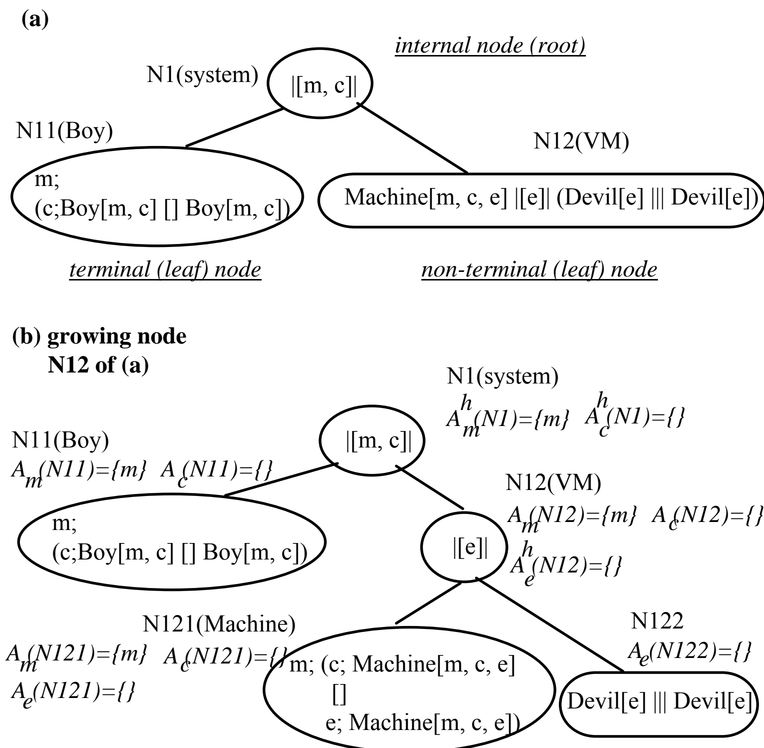
Various forms of trees have been used to represent the behavior of LOTOS specifications. For example, a so-called "behavior tree" [Logr 88] (also called "action tree" in [ISO 89]) is used to show all possible action sequences defined by a LOTOS specification. A static syntax tree can also be used to show the structure of a LOTOS specification and the relations among its actions. Given below is the context-free grammar of a simplified syntax for the temporal control part of LOTOS, which forms the basis on which we will define our activity tree. In the next sub-section, we will define 'attributes' to deal with interaction offers involving data parameters.

In the following, 'B' is a non terminal (and also the starting) symbol of the grammar, and {[], ||, [>, ;, >>, stop, exit, i, g1, ..., gn} is the set of terminal symbols. Each g□{i, g1, ..., gn} denotes a gate in the system, each r□{[], ||, [>, ;, >>} denotes a LOTOS operator, and 'stop' and 'exit' denote the STOP and EXIT processes respectively.

(1) B∅ t for each t□{ stop, exit}
(2) B ∅ g;B for each gate g□{i, g1, g2, ..., gn}
(3) B ∅ B >>B
(4) B ∅ B [] B
(5) B ∅ B || B
(6) B ∅ B [> B

In contrast to the syntax tree of a LOTOS specification which represents the static structure of the text of the specification, the activity tree represents a dynamically changing system state during the execution of the specification. Nevertheless, it has certain similarities with the syntax tree in so far as the production rules of the activity tree correspond to the above syntax rules. The major difference with the behavior tree is that the activity tree is normally not completely expanded. It is grown in a top-down fashion, as explained below, starting with the root node which represents the system specification.

The activity tree reflects the possible activities and the dynamic relationships between the active behavior expressions during the execution of the specified system. An activity tree consists of leaf nodes and internal nodes. An internal node represents the relation between its descendent nodes, i.e. one of the LOTOS operators [], ||, |||, and [> etc., or contains the description of the behavior to be activated after the successful termination of its descendants, i.e. >>B (where B is a behavior expression). There are two kinds of leaf nodes: terminal and non-terminal. A terminal node corresponds to one of the following behavior expressions: 'stop', 'exit' and 'g;B', where 'g' is called an active gate and 'B' is the behavior expression which will be activated after a rendezvous happens at 'g'. A non-terminal node cannot directly participate in an interaction, it must first be expanded. A non-terminal node corresponds to a behavior expression 'B1#B2', where # is one of the operators ||, |||, [], [> and >>, and 'B1' and 'B2' are behavior expressions. During the growing phase, a non-terminal node may be expanded and may thus lead to new terminal nodes that may participate in interactions. Figure 3(a) shows the activity tree of the Example system before any money is dropped in. Figure 3(b) shows the tree after the expansion of the node N12 representing the vending machine (VM). Note that in the node N121, the invocation of the 'Machine[m, c, e]' process is replaced by its definition, as given in Figure 3.



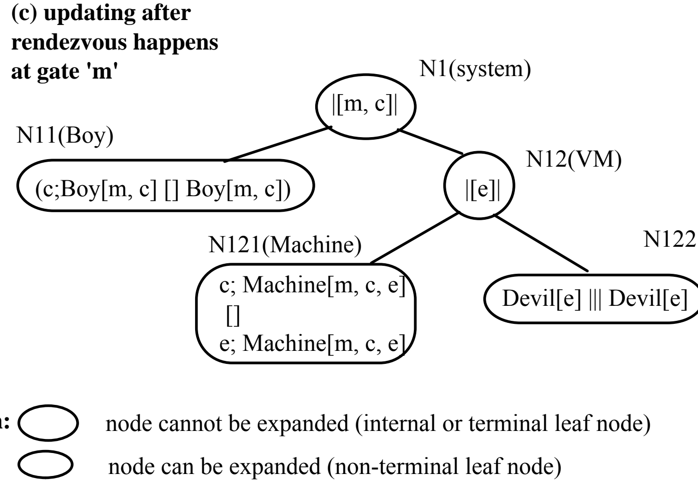


Figure 3: Different stages of the activity tree for the system of Figure 2.

When executing a LOTOS specification, the system behavior changes dynamically. So does the activity tree. The activity tree can be grown and updated. By growing, we mean that the system expands non-terminal nodes in order to find terminal nodes with possible interactions. By updating, we mean that, after a rendezvous, the system prunes those sub-trees of the activity tree which represent alternative behavior being not possible any more and let some behaviors (next behaviors) be active. Figure 4 shows the rules for growing. A non-terminal (leaf) node 'B1*B2' (* is one of [], ||, |||, and [>) can be expanded to become an internal node '*' and with two sons (terminal or non-terminal nodes) 'B1' and 'B2', as shown in Figure 4(a). Figure 4(b) shows a non-terminal node 'B1>>B2' can be expanded to become an internal node '>>B2' with a son (terminal or non-terminal) node 'B1'. We note that there is no growing rule corresponding to the syntax rule of process invocation. If the LOTOS behavior expression of a non-terminal node contains a process invocation, this invocation will be replaced by the behavior of the corresponding process definition with a substitution of its parameters, as defined by the LOTOS semantics. The behavior thus obtained is then the basis for further expansion of the node.

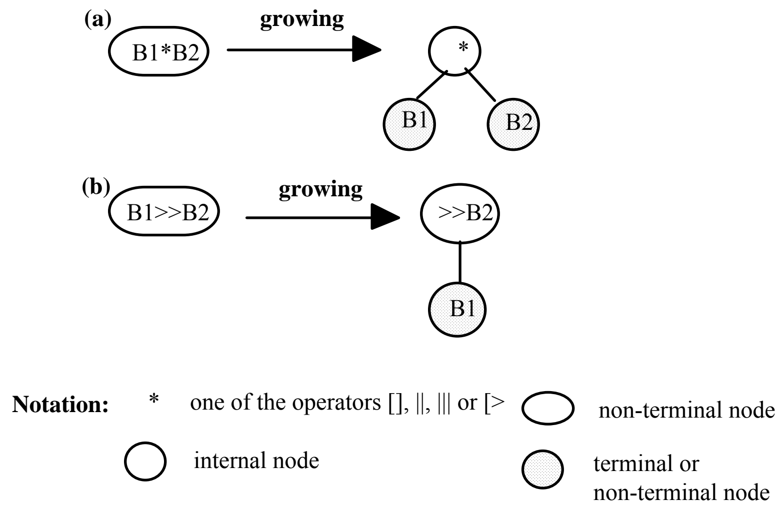


Figure 4: The growing rules

Figure 5 shows two of the rules of updating. After participation a rendezvous at gate 'g', a terminal node 'g;B' become (terminal or non-terminal) node 'B', as shown in Figure 5(a). Figure 5(b) shows a tree with root node '[]' and two sub-trees 'B1' and 'B2'. When a rendezvous happens in 'B2', 'B1' is pruned and 'B2' is updated to " B2' ". That is, the original tree become one with empty root node (which can be replaced by its son) and a sub-tree " B2' ". The full updating rules are given in [Wu 89] by comparing them with LOTOS semantics as defined by the transition system given in [ISO 89]. The growing and updating of the activity tree will be discussed in more detail in Section 3.3.

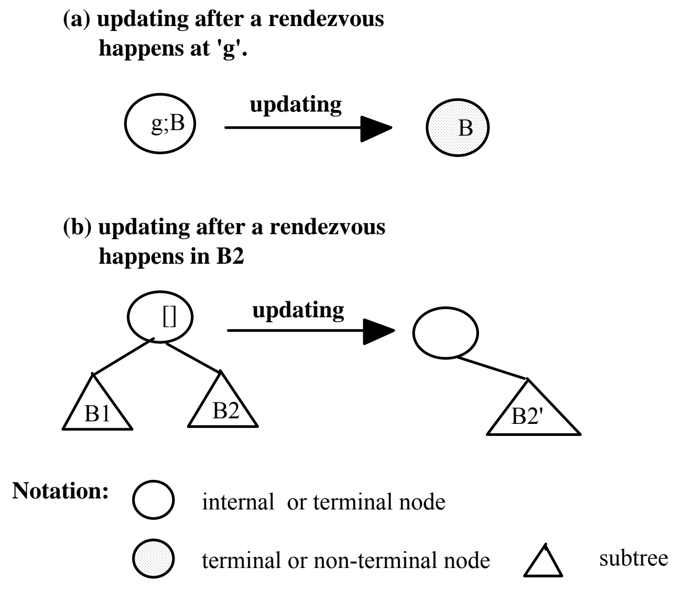


Figure 5: Two updating rules

3.2. Attributes

Attributes are defined in the activity tree. Their function is to determine which nodes participate in a rendezvous on a given gate. As in the case of attribute grammars [Boch 76c], the attributes are associated with the nodes of the tree. In contrast to attribute grammars, however, where the values of the attributes are evaluated once and for all for each given syntax tree, the values of the attributes associated with a node in the activity tree may change over time, as the structure of the activity tree changes.

Without loss of generality, we may assume that each node 'B' of an activity tree corresponds to a specification with the general structure 'P[S1]:= hide S2 in < expression>, where S1 and S2 are the gate lists. Here all free gates of <expression> must either be in S1 or S2. In most cases S2 will be empty, for instance, a node representing the behavior 'g1;B1 [] g2;B2' will be written as 'P[g1, g2] := hide in g1;G1 [] g2;B2'. An attribute Ag is defined in node 'B' for each g□S1≈S2. An attribute Ag is also called a 'hide attribute' (denoted as $A^h{}_g$) if g□S2. The value of attribute Ag is a set of interaction offers concerning the gate 'g'. We treat 'stop' and 'exit' as two kinds of special gates. Note, in LOTOS, gate instances are dynamically created. For example, the statement 'hide S in < expression>' introduces new gate instances. In the following, we use the word 'gate' to denote 'gate instance' and we assume that gate instances have unique names.

The attributes of the activity tree are 'synthesized', that is, they are evaluated by applying the evaluation rules from the bottom of the tree towards the top. The precise definition of these evaluation rules is given in the following table. In the table, 'Ag(B)' denotes the attributes of gate 'g' in node 'B', 'o_g' denotes the interaction offer of gate 'g', 'S' denotes a gate list, and 'B∅ B1#B2' denotes an internal node 'B' which has two son nodes, the left son 'B1' and the right son 'B2', where '#' is one of operators [], ||, [>, and >>. In the table, there are two functions **matched** and **derived**. Their formal definitions are given in [Wu 89]. They can be understood here by the example: **matched**('g?x:int!3?z:int', 'g?x:int?y:int!5') = **true** and **derived**('g?x:int!3?z:int', 'g?x:int?y:int!5') = 'g?x:int!3!5'.

Attribute evaluation rule
For leaf nodes:
A_{stop}(stop) =φ
A_{exit}(exit) ={o_{exit}}
Ag(g;B) ={o_g}

Ag(B) =ϕ if B is non-terminal node

For internal nodes:

Ag(B) = Ag(B1) if B∅ B1>>B2

Ag(B) = Ag(B1)≈Ag(B2) if B∅ B1 [] B2

Ag(B) = Ag(B1)≈Ag(B2) if B∅ B1 |S| B2 and (g□S and g ≠ exit)

Ag(B) ={o'_g |o'_g=derived($o1_g$,$o2_g$),

$o1_g$□Ag(B1), $o2_g$□Ag(B2) and matched($o1_g$, $o2_g$)=true}

if B∅ B1 |S| B2 and (g□S or g = exit)

Ag(B) = Ag(B1)≈Ag(B2) if B∅ B1 [> B2

(Note: Ag(B) =ϕ if gate 'g' is not defined in node 'B')

It is clear that a rendezvous is possible at gate 'g' if the attribute A^h_g at the node where 'g' is hidden contains an offer 'o_g'. All nodes that participate in the derivation of 'o_g' will be involved in the rendezvous. For example in Figure 2(b), a rendezvous can only happen at gate 'm' because A^h_m(N1)={m}, A^h_c(N1)={}, and A^h_e(N12)={}. Nodes N11 and node N121 will be involved in the rendezvous.

3.3 Three phases for the execution of interactions

The activity tree changes dynamically during the execution of LOTOS specifications through the repetition of the following three phases: growing, matching and updating. In the growing phase, the system expands non-terminal nodes until all or some terminal nodes with possible interactions are reached. After that, the system goes into the matching phase, by evaluating attributes, to find possible rendezvous usually involving several terminal nodes of the tree. If a possible rendezvous is found and executed, the matching phase is followed by the updating phase during which the system updates the tree, according to the rules discussed in Section 3.1 to reflect the state change implied by the rendezvous. If the matching phase does not lead to any rendezvous, the growing phase is resumed.

An example of growing is given by Figure 3(b) which shows the activity tree obtained by expanding the non-terminal node N12 of the Example tree of Figure 3(a). Figure 3(b) also shows the values of attributes in each node of the tree, which are obtained in the matching phase. An example of updating is given by Figure 3(c) which shows the Example tree of Figure 3(b) after a rendezvous at gate 'm' in which the terminal nodes N11 and N121 participated.

The dynamic features of the activity tree have certain advantages. As discussed in [Wu 90], it allows parallel processing: The three phases of growing, matching and updating in different parts of the activity tree could be processed largely in parallel. Different growing strategies, such random, breadth-first, and depth-first , can be applied to the activity tree. Some of them can deal with non-well guarded expressions, as discussed in [Wu 90]. Another interesting question is fairness. In [Wu 91], the concepts of "process fairness", "alternative fairness" and "channel fairness" are defined for LOTOS, and it is shown how they can be implemented based on the activity tree.

4. A distributed LOTOS execution model

In this section, we present a distributed execution model for the implementation of LOTOS specifications. It is obtained by modifying the general model of the previous section. The activity tree is partitioned into sub-trees which represent the LOTOS sub-specifications which could be executed in different sites. A sub-tree is handled by a LLEE of a given site (see Figure 1(a)) and the LE component of the LLEE (see Figure 1(b)) implements the functions of the general model, that is, it implements the three phases of growing, matching and updating.

In the growing phase, besides expanding non-terminal nodes to find terminal nodes with possible interactions, the LE may reach non-terminal nodes which represent sub-specifications to be executed in remote sites. In this case, the system calls the DC component of the LLEE (see Figure 1(b)) to transfer it to another site and to establish rings connecting them. After the growing phase, the LE component goes into the matching phase to find locally possible actions (local and global). The DC component makes a choice between the execution of local actions and the execution of global actions. If the DC chooses to execute global actions, the SE (see Figure 1(a)) is called to implement global synchronization. After executing an action, the LE component goes into the updating phase where it updates the local sub-tree.

4.1. Activity sub-trees

In this section, we discuss the decomposition of the global activity tree into sub-trees which we call *a-trees* in the following. In addition to the nodes of an activity tree, an a-tree may include two special kinds of nodes, a Rf-node and Rs-nodes, which represent the relations among the a-trees in the system. An Rs-node in an a-tree T1 represents another a-tree T2, which is logically a sub-tree of T1. An Rs-node is represented by an

expression 'Ts.Gs', where Ts is the identification of the a-tree T2 and Gs is a set of gates for which T2 may provide offers. An Rs-node is a leaf node in an a-tree. An Rf-node is a root node in an a-tree. It represents the super-tree of the a-tree. An Rf-node is represented by an expression 'Tf.Gf' with a similar meaning as for Rs-nodes. Figure 6 shows an example of three a-trees which represents the activity tree of Figure 3 (b). We assume here that each a-tree has an unique identifier. For instance, T1, T2 and T3 are the identifiers of a-trees in Figure 6.

We now give an informal description of the procedure for creating a-trees. Initially there is only one a-tree in the system, which is the activity tree. Let T1 be an a-tree and T2 be its sub-tree which will be partitioned from T1. Then we get two new a-trees T3 and T4. T3 is obtained from T1 by replacing the sub-tree T2 in T1 by an Rs-node 'Ts.Gs', where Ts=T4 and Gs is obtained by checking the behavior defined by T2. A gate **g** is in Gs if T2 may provide interaction offers for it. T4 is obtained from T2 by adding a Rf-node 'Tf.Gf' (root node) to it, where Tf=T3 and Gf is obtained by checking the path in T1 from the root to the sub-tree T2. A gate **g**, for which T4 may provide offers, is in Gf if there is node N along the path such that N is '||', or N is '|S|' and g□S, or N is an Rf-node 'Tf'.Gf'' and g□Gf'.

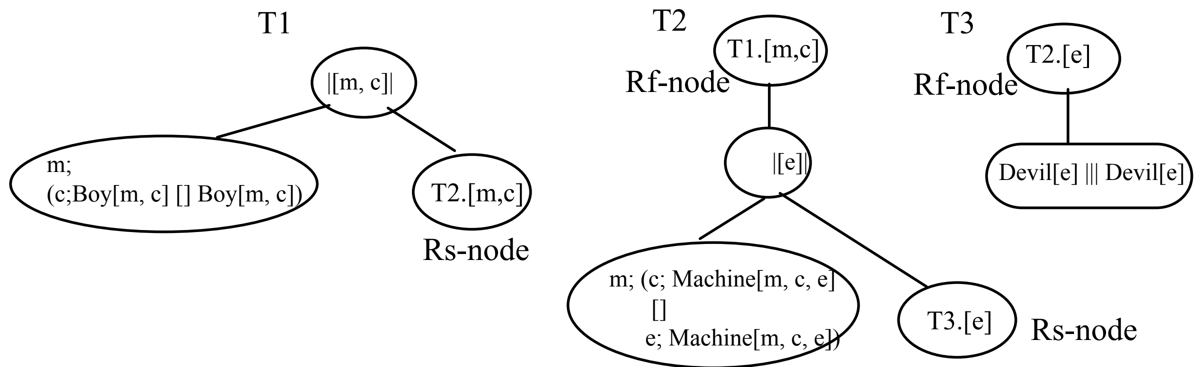


Figure 6: Three a-trees representing the activity tree of Figure 3(b)

4.2. Representing rendezvous relations by rings

In LOTOS, processes rendezvous at gates. However, there may be more than one group of processes that rendezvous at a given gate. For the example of the specification **P1[a] || (P2[a] ||| P3[a])**, gate **a** corresponds to two rendezvous relations. One involves processes **P1** and **P2** and the other involves processes **P1** and **P3**. Therefore we may use a tuple **<g, P>** to uniquely denote a rendezvous relation, where **g** is a gate name and **P** is a set of LOTOS process instances which rendezvous at **g**. However, in our context we are not

interested in LOTOS process instances, but only in sub-specifications which are executed on different sites. A sub-specification may contain more than one LOTOS process instance. Therefore, we use in the following a tuple **<g, S>** to denote a rendezvous relation, where **g** is a gate name and **S** is a set of sub-specifications which rendezvous at **g**.

In the following, a rendezvous relation **<g, S>** is represented by a ring structure. That is, all sub-specifications in **S** are connected by a ring which represents this relation. There are two advantages for using a ring structure. First, a ring structure is used by several distributed rendezvous algorithms [Bagr 87] [Gao 89] [Kuma 90]. Second, the ring structure is easy to manipulate. For example to insert a new node in a ring, the node which is responsible for the insertion of the new node only needs to know the address of its successor in the ring. This feature allows us to deal with the dynamic creation of LOTOS processes and their execution at remote sites.

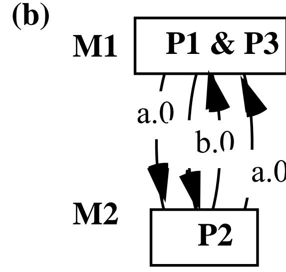
We use the following example to show the complex relations that may exist between sub-expressions, rings, and interaction offers. In fact, a given sub-specification may be connected to several rings corresponding to the same gate. Also, a sub-specification may provide more than one interaction offer on a given ring, and an interaction offer provided by a sub-specification may relate to several rings. Figure 7(a) shows the example specification (we assume that a sub-specification has the same identifier as the site on which it is executed). Suppose that **P1** and **P3** are executed on site **S1** and **P2** is executed on site **S2**, then there are two rings **a.0** and **b.0**, which represent rendezvous relations **<a, {S1,S2}>** and **<b, {S1,S2}>** respectively (see Figure 7(b)). Note that **P2** provides two interaction offers **a!1** and **a!2** for the same ring **a.0**. Figure 7(c) shows the case where **P1**, **P2** and **P3** are executed on sites **S1**, **S2** and **S3** respectively. Note that there are two rings **a.0** and **a.1**, which represent rendezvous **<a, {S1,S2}>** and **<a, {S1,S2,S3}>** respectively, concerning the gate **a**. For **P1**, the offer **a?x:int** relates to ring **a.0** and the offer **a?y:int** relates to ring **a.1**. And the offers **a!1** and **a!2** of **P2** relate to both ring **a.0** and ring **a.1**.

We assume here that each ring has a unique identifier in the system (e.g. a.0, a.1, b.0 in Figure 7(b)(c)). However, each ring may have different local references in different site, which will be explained in the following sections.

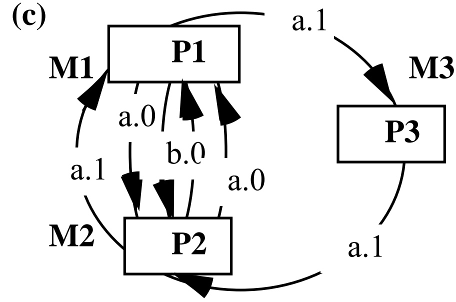
(a)

```
specification Spec[a,b]:exit
behavior P1[a,b] |[a,b]| P2[a,b]
where
process P1[a,b]:exit:=
  (a?x:int; b; exit)
  |||
  (P3[a] |[a]| (a?y:int; exit))
endproc
process P2[a,b]:exit:=
  (a!1; exit) [] (a!2; b; exi
endproc
process P3[a]:exit:=
  a?z:int; exit
endproc
endspec
```

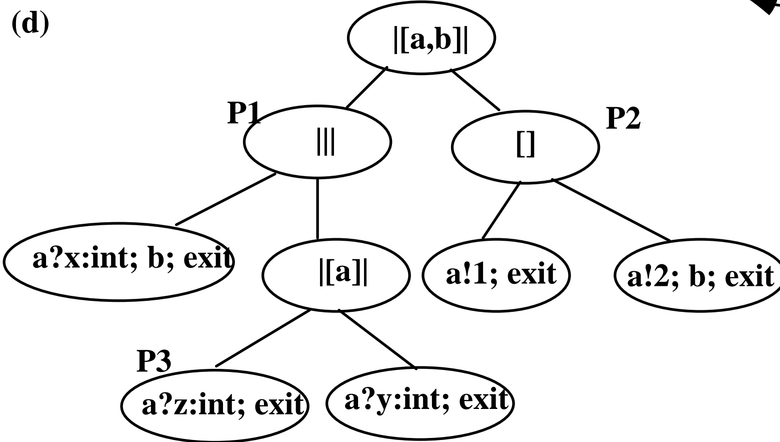
(b)



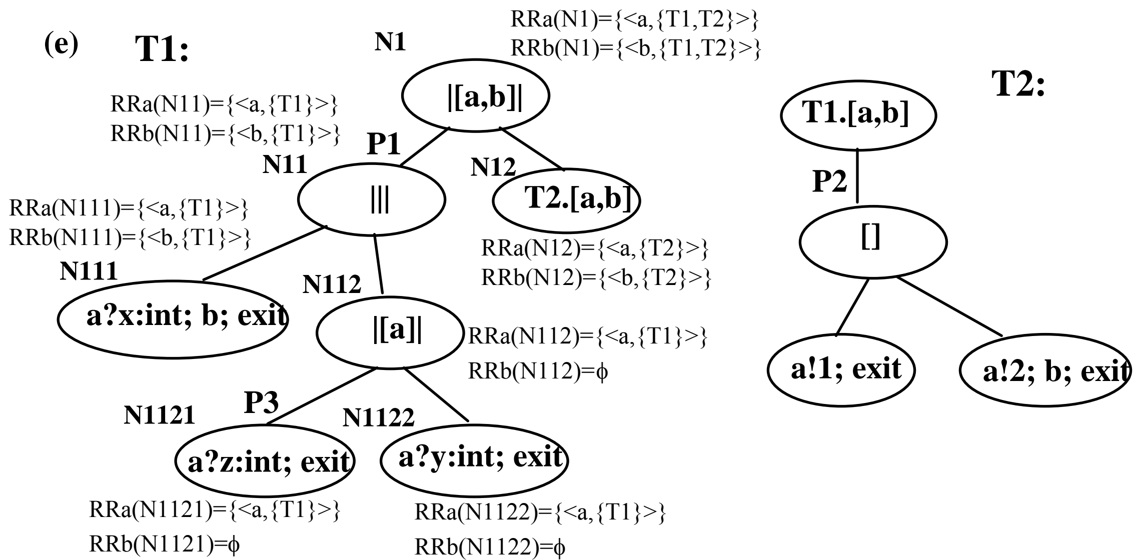
(c)



(d)



(e)



(f)

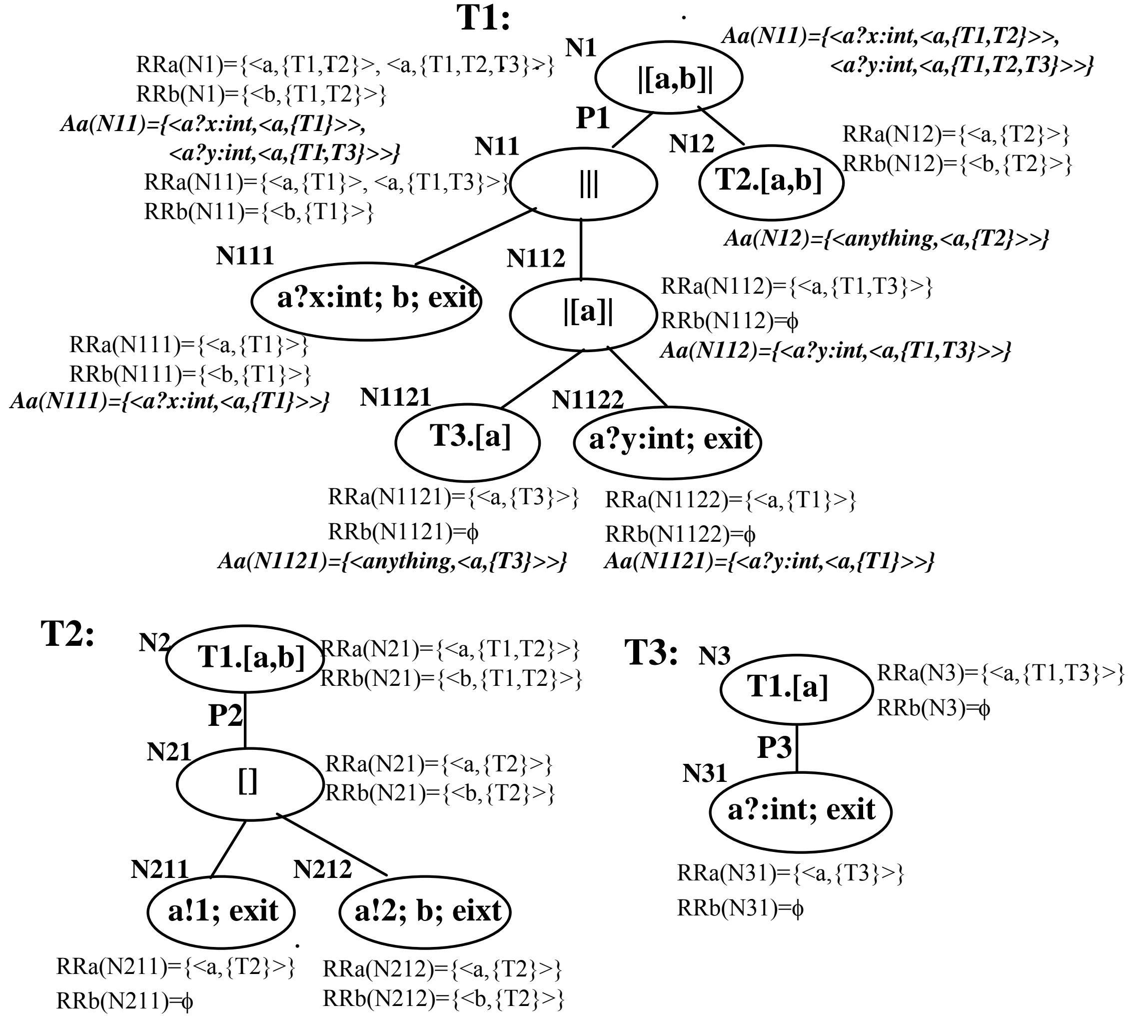


Figure 7: An Example

4.3. Ring establishment

In the above two sub-sections, we considered that a LOTOS specification consists of a set of sub-specifications and we used a-trees to represent these sub-specifications. We also showed that different kinds of ring structures representing rendezvous relations may connect these sub-specifications. In this section, we present an algorithm to dynamically establish these rings.

4.3.1. The general idea

Like the activity tree of Section 3, an a-tree records the relationships of LOTOS processes. For example, Figure 7(d) shows the a-tree of the specification of Figure 7(a). The tree shows that **P1** and **P2** rendezvous at gates **a** and **b**, because they share a parent node **|[a,b]|**. Suppose that **P2** is moved out to execute on another site, then we obtain the two a-trees in Figure 7(e). The a-tree **T1** of Figure 7(e) shows that **P1** and **T2**, which represents **P2**, rendezvous at gates **a** and **b**. Figure 7(f) shows three a-trees, which are obtained when **P3**, a sub-process of **P1**, is moved out to execute on a third site. The a-tree **T1** of Figure 7(f) shows that **T3**, which represents **P3**, rendezvous at gate **a** with **(a?y:int;exit)** in **P1** as well as with **T2** which represents **P2**. It also shows that **T2** rendezvous at gates **a** and **b** with **(a?x:int; b; exit)** of **P1**.

The general idea of the ring establishment algorithm is the following. Each time when a sub-tree **T'** is moved out from an a-tree **T''** to be executed on another site, we check the relationship between **T'** and the rest of **T''**, which may include Rs-nodes and Rf-nodes denoting other a-trees. If **T'** has no rendezvous relationship concerning gate **g** with the rest of **T''**, then there is no ring concerning this gate connecting **T'** and **T''**; if **T'** has a rendezvous relationship concerning gate **g** with some nodes of T'' and these nodes include neither Rs-nodes nor the Rf-node, a ring concerning the gate is created between **T''** and **T'**; if **T'** has a rendezvous relationship concerning gate **g** with some nodes of **T''** which include some Rs-nodes or the Rf-node, **T'** should be inserted into the corresponding rings. For example, after partitioning the a-tree of Figure 7(d), we have two a-trees in Figure 7(e). By checking the relationship between the two trees, two rings are created to connect the two trees (see Figure 7(b)). By continuing the partition of the a-tree **T1** in Figure 7(e), we obtain the three a-trees shown in Figure 7(f). The corresponding rings are shown in Figure 7(c).

4.3.2. Rendezvous relation attributes and their evaluation rules

A ring represents a rendezvous relation. In this section, *rendezvous relation attributes* are defined for the nodes of all a-trees to check the relationship among them. The value of a rendezvous relation attribute concerning gate **g** is a set of tuples **<g, T>**, where **g** is a name of a gate and **T** is a set of identifiers of a-trees in the system. A tuple **<g, T>** denotes the rendezvous relation at gate g involving all **Ti□T**. An attribute value including several tuples concerning gate **g** means that there is more than one rendezvous relation concerning this gate.

The following table shows the evaluation rules for rendezvous relation attributes, where RRg(N) denotes the rendezvous relation attribute for gate **g** at node **N** of an a-tree. Rendezvous relation attributes are evaluated from the bottom up through the tree. Note that the rendezvous relation attributes of a given a-tree give only a partial picture of the global rendezvous relations. This is so because certain rendezvous relations may not involve the a-tree in question, and/or the set T of a-tree identifiers of certain rendezvous relations included in an attribute may miss certain a-trees which are involved only indirectly through the a-nodes which are referenced in the Rs-nodes and Rf-nodes. For example, the a-tree **T2** of Figure 7(f) does not include a reference for a-tree **T3**. Therefore a tuple **<g, T>** is more accurately interpreted in this context saying that there is a rendezvous relation at gate **g** which involves at least all Ti□T. The global rendezvous relations can be constructed based on the (local) rendezvous relation attributes of all the a-trees in the system, as discussed in the next sub-section.

Rendezvous relation attribute evaluation rules

For leaf nodes:

RRg(N) =<g,{T_s}>	if N is Rs-node and N = $T_s.G_s$ and g□G_s
RRg(N) =ϕ	if N is Rs-node and N = $T_s.G_s$ and g□G_s
RRg(N) = <g,{T_{local}}>	if N is not Rs-node and the behavior defined by N does or will provide offers concerning gate g; T_{local} is the identifier of the local a-tree
RRg(N) = ϕ	if N is not Rs-node and the behavior defined by N does not or will not provide any offers concerning gate g

For internal nodes:

RRg(N) = RRg(N1) ≈ RRg(N2)	if N∅ N1>>N2
RRg(N) = RRg(N1) ≈ RRg(N2)	if N∅ N1 [] N2
RRg(N) = RRg(N1) ≈ RRg(N2)	if N∅ N1 |S| N2 and g□S
RRg(N) = {<g,T>| T=T1 ≈ T2 and<g,T1>□RRg(N1) and <g,T2>□RRg(N2)}	if N∅ N1 |S| N2 and g□S
RRg(N) = RRg(N1) ≈ RRg(N2)	if N∅ N1 [> N2
RRg(N) = RRg(N1)	if N∅ N1 and N is Rf-node and N = $T_f.G_f$ and g□G_f
RRg(N) = {<g,T>| T=T_f ≈ T1 and<g,T1>□RRg(N1)}	if N∅ N1 and N is Rf-node and N = $T_f.G_f$ and g□G_f

For example, the **RRa(N1)= {<a,{T1,T2,T3}>, <a,{T1,T2}>}** at the root of the a-tree **T1** of Figure 7(f) indicates that there two rendezvous relations concerning gate **a**. The first one involves at least **T1**, **T2** and **T3**, and the second one involves at least **T1** and **T2**.

4.3.3. Ring establishment algorithm

We assume in the following that a sub-tree, which will be moved away from a given a-tree **Tlocal**, does not include any Rs-node. When the sub-tree is moved, it will be replaced by an Rs-node. Before the moving, the sub-tree has the value of **Tlocal** for its rendezvous relation attributes concerning related gates, because it represents a local behavior. After the move, the related Rs-node has the value **Tmove** as its rendezvous relation attributes concerning related gates, where **Tmove** is the identifier of the newly created a-tree. Thus the distribution of the sub-tree may cause, through the evaluation rules of the table above, a change of values of the related rendezvous relation attributes in the root of the a-tree **Tlocal**.

Let **RRg** be the newly evaluated rendezvous relation attribute concerning gate **g** after the moving of **Tmove**. We have the following ring establishment rules:

1) For elements in **RRg** which do not include **Tmove**, no action is required;

2) For each **<g, {Tlocal , Tmove}>** □**RRg**, a new ring concerning the gate **g** will be created which connects **Tlocal** and **Tmove** and the local system identifies the ring by the tuple **<g, {Tlocal , Tmove}>;**

3) For each **<g, U>** □ **RRg** such that **Tmove**□**U** and case (2) does not apply, we define **V** to be obtained from **U** by replacing **Tmove** by **Tlocal**. We distinguish the following two cases:
a) if **<g,V>** □ **RRg**, **Tmove** will be inserted in the ring identified by **<g,V>** and the local identifier of the ring is changed from **<g,V>** to **<g,U>**;
b) if **<g,V>**□ **RRg**, then a ring, which is the duplication of the ring identified by **<g,V>**, is created and **Tmove** is inserted. This new ring is identified by **<g,U>**.

We consider the example of Figure 7. The a-trees **T1** and **T2** in Figure 7(e) represent **P1** and **P2**, respectively, in the specification of Figure 7(a), and the two rings **a.0** and **b.0** connect **T1** and **T2** (see Figure 7(b)). The related rendezvous relation attributes at the root of **T1** are **RRa(N1)={<a,{T1,T2}>}** and **RRb(N1)={<b,{T1,T2}>}**, and they refer to

the rings **a.0** and **b.0**, respectively. Now consider that **P3** is moved to site **S3** to execute. We have the a-trees **T1**, **T2** and **T3** in Figure 7(f). The rendezvous relation attribute at the root of **T1** becomes **RRa(N1)= {<a,{T1,T2}>, <a,{T1,T2,T3}>}**, which is different form the one in Figure 7(e), and **RRb(N1)= {<b,{T1,T2}>}**, which is the same as in Figure 7(e). The value of **RRa(N1)** leads to the ring establish rule 3 above. This means that the ring **a.1**, which is the duplication of the ring **a.0** locally identified as **<a,{T1,T2}>**, is created and **T3** is inserted. This new ring is identified by **<a,{T1,T2,T3}>**. We have the new ring structure of Figure 7(c).

4.4. Matching rules

In this section, we define the rules which are used in the matching phase to find locally possible action offers for a given a-tree. Through these rules, which are similar to those of Section 3.2, the local system can not only determine whether a given action offer is local or global, but also determine the rendezvous relations (rings) to which a given global action offer relates. The idea is that each offer is associated with a tuple **<g, T>** which denotes a rendezvous relation (see Section 4.3.2). The attribute evaluation rules of Section 3.2 are modified to match offers and also to evaluate related rendezvous relations. When an offer of a possible action is found, by comparing the tuple **<g, T>** with ring identifiers of the local system, one can identify whether it is local or global, and also the rings to which it relates.

The following table shows the rules. An offer is in a form **<og, <g, T>>**, where **og** is an ordinary offer for gate **g** (see Section 3) and **<g, T>** denotes a rendezvous relation. We introduce a specific kind of offer **anything** to the Rf-node or Rs-nodes. The two functions **matched_r** and **derived_r** are defined based on the two functions **matched** and **derived** of Section 3.2.

Modified attribute evaluation rules for a-tree T_{local}	
For leaf nodes:	
Ag(B) = {<anything,<g,{T_s}>>}	if B is Rs-node and B = $T_s.G_s$ and g□G_s
Ag(B) =ϕ	if B is Rs-node and B = $T_s.G_s$ and g□G_s
A_{stop}(stop) =ϕ	
A_{exit}(exit) ={<o_{exit},<exit,{T_{local}}>>}	
Ag(g;B) ={<o_g,<g,{T_{local}}>>}	
Ag(B) =ϕ	if B is non-terminal node
For internal nodes:	
Ag(B) = Ag(B1)	if B∅ B1>>B2

Ag(B) = Ag(B1)≈Ag(B2)   if B∅ B1 [] B2
Ag(B) = Ag(B1)≈Ag(B2)   if B∅ B1 |S| B2 and (g□S and g≠exit)
Ag(B) = {<o_g,<g,T>> | o_g=derived_r(o1g, o2g) and
    matched_r(o1g, o2g) and T= T1≈T2 and
    <o1g,<g,T1>>□Ag(B1) and <o2g,<g,T2>>□Ag(B2) }
        if B∅ B1 |S| B2 and (g□S or g=exit)
Ag(B) = Ag(B1)≈Ag(B2)   if B∅ B1 [> B2
Ag(B) = Ag(B1)   if B∅ B1 and B is Rf-node and B = Tf.Gf and g□Gf
Ag(B) ={<o_g,<g, T>> | o_g=$o1_g$) and T= {Tf}≈T1
    and <$ol1_g$, <g, T1>> □Ag(B1)}
        if B∅ B1 and B is Rf-node and B = Tf.Gf and g□Gf

The definition of the function matched_r:

1) matched_r(anything, oj) = true
2) matched_r(oi, anything) = true
3) matched_r(oi, oj) = matched(oi, oj)

The definition of the function derived_r:

1) derived_r(anything, oj) = oj
2) derived_r(oi, anything) = oi
3) derived_r(oi, oj) = derived(oi, oj)

Let **ROOT** denote the root of the a-tree **Tlocal**. Then **Ag(ROOT)** represents the set of locally possible actions. If **<og, <g, {Tlocal}>>□ Ag(ROOT)**, then **og** is a local action offer. If **<og, <g, T>>□ Ag(ROOT)** and there is more than one element in **T**, then **og** is a global action offer which relates to the ring locally identified as **<g, T>**. For example, by applying the rules above to the a-tree **T1** in Figure 7(f), we have **A_a(N1)={<a?x:int , <a, {T1,T2}>>, <a?y:int, <a, {T1,T2,T3}>>}**. We see that the offer **a?x:int** relates to the ring **a.0** identified by **<a, {T1,T2}>** , and the **offer a?y:int** relates to the ring **a.1** identified by **<a, {T1,T2,T3}>>** (see Section 4.3).

5. Implementation

The general structure of our system is shown in Figure 1. The functions of its components are described in Sections 2 and 4. In this section we describe an implementation of these concepts in an environment of several UNIX work-stations connected by a local area network. The major part of the system is programmed in

Prolog, and the Transport service provided by Unix Socket is used for the communication among Prolog programs at different sites.

The execution of a LOTOS specification proceeds in several phase. First, the user has to partition the LOTOS specification. For this purpose, he/she may use the special pre-defined data type **move** in the LOTOS specification. The data type **move** supports the operations such as **not_moving, moving, address1, ..., addressn** which are interpreted by the system. A process definition containing a parameter of type **move** indicates that instances of the process could move to a remote site for execution. For example, in the following specification, the definition of process P contains a parameter of type **move**; its instance **P[g](not_moving)** will be kept at the local site; its instance **P[a](moving)** will be executed on a remote site which is selected by the system; and its instance **P[a](address)** will be executed on the site whose address is **address**.

specification Spec[a] :=
behavior P[a](not_moving) || P[a](moving) || P[a](address) || (a; exit)
where
process P[a](m: move):= a; i; exit **endproc**
endspec

Second, the specification is compiled by the ISLA compiler [Logr 88] into an internal Prolog representation. Third, the user defines the number of sites in the system. Then he/she initializes the system by loading at each site a copy of the internal Prolog representation of the specification. The execution is initiated at one site and is later propagated to the other sites in the system, as explained in Section 4.

5.1. The implementation of Local LOTOS Execution Engine (LLEE)

5.1.1. The implementation of LOTOS Execution (LE)

The LE is implemented based on the execution model described of Section 3. It consists of two parts: a *data part* and a *control part*. The data part, which is SVELDA developed by Ottawa University [Logr 88], handles the data part of LOTOS. The control part handles the control part of LOTOS. It implements the three phases of growing, matching and updating (see Section 3).

We use a set of dynamic Prolog clauses (facts) to implement the a-tree. Each of these dynamic clauses corresponds to a node in the tree, and contains information such as the

type of the node (terminal, non-terminal, Rs-node, Rf-node, ...), the behavior associated with the node, and the list of attributes assigned to the node at some stage of execution.

The organization of an a-tree as a set of related clauses makes it more accessible and quite controllable. Our system supports different growing strategies such as depth-first, breadth-first and a combined strategy. The following parameters determine the growing strategy:

- the sub-list of non-terminal nodes to be expanded,
- the order in which these non-terminal nodes should be expanded
- the level of expansion for each non-terminal node.

5.1.2. The implementation of Distributed Coordination (DC)

The DC component is implemented based on the distributed execution model described in Section 4. It consists of three procedures: Distribution , Ring Establishment and Local Rendezvous Manager (LRM).

The function of the Distribution procedure is to distribute sub-specifications onto different sites in the system in a dynamic mode or in a static mode (see Section 2). It is called by the system when a non-terminal node, which represents a sub-specification to be executed in a remote site, is found in the growing phase. In the dynamic distribution mode, the Distribution consults the Allocation Management component (see Figure 1) for a free site.

The Ring Establishment procedure is called when the system distributes sub-specifications. It evaluates rendezvous relation attributes and establishes rings by applying the algorithm of Section 4.3.3. To facilitate the manipulation of the duplication of rings in our implementation, we consider that a link (arc) between two sites may belong to more than one ring. Thus, to duplicate a ring, the site may simply duplicate the link to the next neighbor. To do this, a special naming scheme was designed for rings in our implementation, which will not be discussed here.

The LRM does two things: It chooses a local action for execution and communicates with the SE component for the execution of global actions (see Section 2). The LRM applies the matching rules of Section 4.4 to identify a global action offer and the related ring. When communicating with the SE, the LRM sends all the possible global action offers to the SE together with related ring identifiers.

5.2. Allocation Management

Allocation management is concerned with allocating the different parts of a given specification onto the different sites for execution. It involves two basic questions: how to partition a system into sub-systems and how to distribute these sub-systems within a given distributed environment. The partitioning may have a strong impact on the overall efficiency of the system, due to the relatively low speed of inter-site communication compared with the communication within a single site. One can have different kinds of allocation schemes. For an automated solution, the system given a specification and a distributed environment, automatically partitions the specification to gain the best performance. It is clear that this is not an easy question. The partitioning can also be done by the designer, because he/she may have better knowledge about the specified system. One has a static solution if any decision is made before the initialization; this is easy to implement. One has dynamic solution if decisions are made at run-time; this provides more flexibility.

Our implementation assumes that the user partitions the LOTOS specification before its execution. The distribution can be done in two modes: static and dynamic, as discussed in Section 2. The static mode is straight forward to implement. For the dynamic distribution, we have built a centralized allocation manager which records the free LLEE's in the system and allocates them to new sub-specifications according to the requests received from the active LLEE's.

The function of the central allocation manager could be implemented in a distributed manner. It could also apply some more sophisticated allocation strategy to improve system performance. In our system, a site manipulates only one a-tree at a time. We have not perused these questions further in our implementation.

5.3. Implementation of the distributed multi-way rendezvous

For the implementation of the distributed multi-way rendezvous, we use the algorithm of [Gao 89]. The algorithm has been formally specified in the specification language Estelle. It then has been simulated using a simulation tool called VEDA [Jard 85b] in order to validate the protocol [Gao 91]. The algorithm will be implemented in C code using a semi-automated translation approach [Boch 87h]. On a given site, an entity of the algorithm (a SE of Figure 1) communicates with a local execution model of Section 4 (an LLEE of Figure 1) through UNIX sockets.

For debugging the implementation of the LLEE's, we designed a Central Rendezvous Manager which simulates the function of the distributed multi-way rendezvous algorithm in a centralized manner. It is a Prolog program and communicates with the LLEE's in the system through UNIX sockets. The interface between the Central Rendezvous Manager and the LLEE's is exactly same as between the SE's and the LLEE's.

5.4. Practical Experiments

The LLEE's work properly in an environment with a Central Rendezvous Manager. We have successfully executed a LOTOS specification of the dining philosophers in a distributed environment involving eight sites.

6. Conclusions

We presented a new solution for the distributed implementation of LOTOS specifications. Our system consists of two functional parts, one handles the behavior of LOTOS specifications locally and the other deals with synchronization. There is a clear and simple interface between them and they work independently. In our system, sub-specifications which are executed in different place are treated equally, that is, there is no master-slave relationship between different sites. We introduce the concept of a-trees to execute LOTOS specifications. It has the advantage of being easy to control and can deal with non-well-guarded expression of LOTOS (see [Wu 90]). Ring structures are used to represent rendezvous relations between sub-specifications. Distributed synchronization algorithms, based on ring structures, are used to synchronize sub-specifications. The distribution of sub-specifications can be done at run-time and allows for various allocation strategies. A prototype of this system has been implemented and has been used for certain experiments.

A related interesting question is the execution of LOTOS specification with fairness. The concepts of process fairness, alternatives fairness and channel fairness have been defined for LOTOS in [Wu 91]. The fair execution of LOTOS expressions, based on the activity tree, is also described for a centralized environment. The distributed implementation of fairness is more difficult to realize. It depends on a fair distributed rendezvous algorithm [Atti90] [Wu 91c]. The algorithm of [Gao 89] respects fairness in a probabilistic manner, based on random choice. This question needs further study.

Other interesting issues are related to the problem of partitioning a given specification into several components that could be executed at different sites, and the question of

allocating the different components, possibly dynamically depending on the requirements of the data being processed. The system described here provides a framework in which different approaches to the partitionning and allocation problems can be explored. However, we have not addressed these problems in any specific terms. Futher work would be useful in this area.

We also need to study the performance aspects of our system. The question relates with aspects such as: the styles of specifications, implementation strategies, etc. Because our system is designed for proto-typing, we only consider implementation-oriented LOTOS specifications (that is, the structure of a specification reflects the structure of its implementation). Further work is needed to evaluate different implementation strategies such as different distributed synchronization algorithms, different strategies of manipulating of trees (i.e. growing, matching and updating of trees) against different implementation environments.

Acknowledgement: The authors would like to thank Dr. Anindya Das, Dr. Reinhard Gotzhein and Dr. Roland Groz for helpful discussions and comments.

Reference

[Atti 90] R. C. Attie, I. R. Forman and E. Levy, *On Fairness as an abstraction for the design of distributed systems,* Proc. of The 10th International Conference on Distributed Computing Systems, Paris, France, 1990.

[Bagr 87] R. Bagrodia, *A Distributed Algorithm to Implement N-Party Rendezvous,* LNCS 287, Springer, 1987, pp. 138 - 152.

[Boch 76c] G. v. Bochmann, *Semantic evaluation from left to right*, Communication ACM, Vol. 19, 1976, pp. 55-62.

[Boch 87h] G. v. Bochmann, G. Gerber and J.-M. Serre, *Semiautomatic implementation of communication protocols*, IEEE Tr. on SE, Vol. SE-13, No. 9, September 1987, pp. 989-1000 (reprinted in "Automatic Implementation and Conformance Testing of OSI Protocols", IEEE, edited by D.P.Sidhu, 1989).

[Boch 89c] G. v. Bochmann, Q. Gao and C. Wu, *On the distributed implementation of LOTOS*, FORTE'89 (IFIP), Vancouver, in "Formal Descriptions Techniques II", Notth-Holland, S.T.Vuong editor,1989, pp.133-146.

[Bolo 87] T. Bolognesi and E. Brinksma, *Introduction to the ISO Specification Language LOTOS*, Computer Network and ISDN Systems, Vol. 14, No. 1, 1987, pp. 25 - 59.

[Bria 86] J. P. Briand, M. C. Fehri, L. Logrippo and A. Obaid, *Executing LOTOS specifications*, in Protocol Specification, testing and Verification, B.Sarikaya and G.Bochmann (eds), North Holland, 1986.

[Dubu 89] E. Dubuis, *An algorithm for translating LOTOS behavior expressions into automata and ports,* Proc. of FORTE'89, Vancouver, 1989.

[Gao 89] Q. Gao and G. v. Bochmann, *Distributed Implementation of LOTOS Multiple-Rendezvous,* Participant's proc. of The 9th International Symposium of Protocol Specification, Testing, and Verification, Enschede, The Netherlands, 1989.

[Gao 91] Q. Gao and et.al, *Validation of Virtual Ring Algorithm of Veda*, in preparation, 1991.

[Gilb 89] D. R. Gilbert, *A LOTOS to PARLOG translator,* in: Turner (Eds.), Formal Description Techniques, Elsevier Science Publishers B.V. (North-Holland), 1989.

[ISO 89] ISO, *LOTOS: a formal description technique,* IS8807, 1989.

[Jard 85b] C. Jard, R. Groz and J. F. Monin, *VEDA: a software simulator for the validation of protocol specifications*, Proc. COMNET '85 (IFIP), Computer Network Usage: Recent Experiences, North Holland, Oct. 1985.

[Karj 88] G. Karjoth, *Implementing process algebra specifications by state machines*, Proc. IFIP Symposium on Protocol Specification, Testing and Verification, Atl. City, 1988.

[Kuma 90] D. Kumar, *An implementation of N-way Synchronization Using Tokens,* Proc. of The 10th International Conference on Distributed Computing Systems, Paris France, 1990.

[Levy 88] E. Levy, *A Survey of Distributed Coordination Algorithms*, MCC Technical Report Number STP-271-88, 1988.

[Logr 88] L. Logrippo and e. al., *An interpreter for LOTOS: A specification language for distributed systems*, Software Practice and Experience, Vol. 18(4), pp.365-385, April 1988.

[Nomu 90] S. Nomura, T. Hasegawa and T. Takiznka, *A LOTOS Compiler and Process Synchronization Manager,* Participant's proc. of Tenth International IFIP Symposium on Protocol Specification, Testing, and Verification, Ottawa, 1990.

[Rame 87] S. Ramesh, *A New and Efficient Implementation of Multi-process Synchronization,* Proc. of PARLE, Eindhoven, 1987.

[Sist 91] R. Sisto and L. Ciminiera, *A Protocol for Multirendezvous of LOTOS Processes*, IEEE Transaction on Computers, Vol. 40, No. 1, April, 1991, pp. 437 -447.

[Sjod 89] P. Sjodin, *A Distributed Algorithm for Synchronous Process Communication at Ports,* Participant's proc. of 9th International Symposium of Protocol Specification, Testing, and Verification, Enschede, The Netherlands, 1989.

[Turn 89] K. Turner, *A LOTOS based development strategy,* Participant's proc. of FORTE'89, Vancouver, 1989.

[Wu 89] C. Wu and G. v. Bochmann, *An execution model for LOTOS specifications,* No. 701, Dept. I.R.O., Universite de Montreal, Oct. 1989.

[Wu 90] C. Wu and G. v. Bochmann, *An Execution Model for LOTOS Specifications,* Proc. of IEEE Global Telecommunications Conference (GLOBCOM'90), San Diego, Dec. 2-5, 1990.

[Wu 91] C. Wu and G. v. Bochmann, *Fairness in LOTOS,* FORTE'91(IFIP), Sydney, Nov. 19 - Nov. 22, 1991.

[Wu 91c] C. Wu, G. V. Bochmann and M. Yao, *Fairness of N-party synchronization and its implementation in a distributed environment*, in preparation, 1991.