Structures de contrôle pour des comportements répartis

Laurence Duchien, Gérard Florin, Lionel Seinturier

RECHERCHE

Structures de contrôle pour des comportements répartis

Laurence Duchien* **– Gérard Florin*** **– Lionel Seinturier****

* *Conservatoire National des Arts et Métiers, Laboratoire CEDRIC*
292 rue Saint Martin, 75141 Paris Cedex 03
{duchien, florin}@cnam.fr

** *Université Pierre et Marie Curie, Laboratoire LIP6, Thème SRC*
4 place Jussieu, 75252 Paris Cedex 05
Lionel.Seinturier@lip6.fr

RÉSUMÉ. *Cet article propose un langage pour la description de comportements réalisés par des groupes d'objets répartis. Il introduit deux structures de contrôle : la condition et l'itération réparties. Ces structures généralisent à un niveau réparti, les notions de condition if-then-else et de boucle while. Le langage proposé utilise une logique modale dite logique épistémique de la connaissance pour caractériser les comportements de ces structures. Deux algorithmes illustrent l'utilisation de ce langage.*

ABSTRACT. *This article proposes a language to define behaviours performed by groups of distributed objects. Two control structures are introduced: the distributed condition and the distributed iteration. These structures are distributed generalizations of if-then-else statements and while loops. The language uses a modal logic known as the epistemic knowledge logic to characterize the behaviours performed by these structures. Two algorithms illustrate the language.*

MOTS-CLÉS : *structures de contrôle réparties, logique épistémique, groupes d'objets répartis*

KEY WORDS : *distributed control structures, epistemic logic, groups of distributed objects*

Technique et science informatiques. Vol. 19 - n° 9/2000, page 1245 à 1268

1. Introduction

Les plates-formes telles que CORBA [OMG 98] ou DCOM [GRI 97], mais aussi le langage Java et ses extensions (RMI [SUN 97] et EJB [SUN 99]) favorisent le développement d'applications concurrentes et réparties. Ces outils orientés objet facilitent la tâche des développeurs en ce qui concerne la gestion des communications et l'utilisation de processus concurrents. Ces deux concepts sont maintenant suffisamment intégrés pour que le programmeur ait peu à s'en soucier. Par contre, il reste toujours difficile de décrire les interactions entre les objets et leur enchaînement au sein d'une application.

Les applications réellement coopératives avec de forts besoins de coordination, telles que les collecticiels avec rédaction coopérative de documents et réconciliation entre versions ou encore les vidéoconférences avec coordination entre supports multiples, les algorithmes répartis du domaine des réseaux, les systèmes multi-agents pour l'intelligence artificielle distribuée sont réalisés grâce à l'énergie et à l'intelligence du concepteur. En effet, lorsque les échanges entre objets sont nombreux et nécessitent la mise en place d'une algorithmique répartie complexe, le concepteur a peu de moyens pour modéliser les interactions et vérifier que l'objectif de l'application est atteint.

Notre problématique est avant tout celle des styles de programmation. Nous souhaitons introduire un certain nombre de notions permettant de faciliter l'expression des algorithmes répartis. Une idée simple consiste à donner au concepteur la possibilité d'exprimer le but du comportement collectif ainsi que des grandes étapes à l'aide de structures de contrôle bien identifiées.

Ce travail de conception est à placer en amont de la conception habituelle des applications orientées objet : il permet de déduire les différents objets participants ainsi que leurs interactions. Les méthodologies orientées objet existantes de type OMT [RUM 91], Booch [BOO 94] ou UML [BOO 97] ont été définies selon une approche centralisée du monde objet. Elles ne donnent qu'une vision statique de la répartition en décrivant le placement des composants logiciels d'une application sur les différents sites d'un système. L'une des raisons de ce manque d'intérêt pour la répartition est que ces méthodes se centrent essentiellement sur l'étude de la manipulation d'ensembles de données. Ce style de conception convient parfaitement à des applications que l'on décentralise en séparant les fonctions clientes des fonctions serveurs.

Dans cet article, nous introduisons la notion de programme pour des groupes d'entités réparties, par exemple des groupes d'objets, cas le plus fréquemment rencontré dans les applications réparties actuelles. Un groupe représente l'ensemble des entités prenant part à la réalisation d'une application. Nous définissons un programme de groupe comme étant alors l'ensemble des comportements nécessaires à la réalisation de l'application répartie. Il est construit à partir de structures de contrôle (condition et itération réparties) que nous présentons dans cet article. Il permet d'abstraire les comportements locaux des objets et les échanges de messages entre les membres du groupe. Nous associons à chaque programme des assertions de logique épistémique [MEY 95]. Celle-ci permet de formaliser le but de l'application et de caractériser l'évolution du comportement en utilisant la notion de connaissance. Notre principal

apport est d'unifier dans une même approche une vision de la répartition en terme d'échanges de connaissances et une vision en terme de structures de contrôle. Cette abstraction permet de s'affranchir des détails de programmation et de présenter une vue synthétique des applications réparties.

L'article est organisé comme suit. Nous introduisons au paragraphe 2 la notion de langage de groupe. Le paragraphe 3 présente deux structures de contrôle, la condition et l'itération réparties, utilisées par les programmes de ce langage. Les paragraphes 4 et 5 explicitent deux exemples de mise en œuvre de ces programmes. Nous positionnons cette étude par rapport à des travaux existants au paragraphe 6. Finalement, le paragraphe 7 conclut cet article et propose des perspectives de travaux futurs.

2. Langage de groupe

Nous définissons dans cette partie, la notion de langage de groupe et les éléments que nous y associons. Le paragraphe 2.1 fournit le cadre général de notre étude. Le paragraphe 2.2 s'intéresse à la syntaxe des programmes de groupe. Elle repose sur une notation pseudo-algorithmique facilement accessible. Les instructions de ces programmes sont associées à des assertions exprimées à l'aide de la logique épistémique [MEY 95] que nous présentons au paragraphe 2.3. Le paragraphe 2.4 décrit les éléments (variables et instructions) de notre langage.

2.1. *Cadre général de l'étude*

Ce travail se place dans le cadre d'une méthode de conception pour des applications réparties que nous avons défini précédemment dans [BON 96] et [SEI 97]. Cette méthode introduit trois points de vue ou niveaux dans la conception d'une application répartie. Ce sont les niveaux groupe, objet et méthode. Le premier s'intéresse à la distribution et aux interactions : il permet de décrire le comportement global de l'ensemble des objets prenant part à la réalisation de l'application. Le deuxième point de vue traite de la concurrence et de la synchronisation : il définit les comportements locaux à mettre en œuvre dans les objets participant à la réalisation du comportement de groupe. Finalement, le dernier point de vue s'attache à la description des fonctionnalités de base de chaque objet. Ces trois niveaux correspondent aux trois grandes étapes d'une démarche descendante qui procède par raffinements successifs. Ils permettent d'aborder progressivement la conception des différentes fonctionnalités, de la définition d'un but global jusqu'à la programmation dans un langage objet. Des raffinements peuvent également être introduits au sein de chaque niveau.

Dans cet article, nous nous intéressons aux comportements relevant du niveau groupe. Le paradigme majeur que nous introduisons est celui de programme de niveau groupe. Nous considérons ainsi qu'une application répartie est un programme qui s'exécute sur un groupe d'objets répartis. Ces programmes manipulent des structures de données et des structures de contrôle dont il convient de préciser le sens.

2.2. *Définition et syntaxe*

2.2.1. *Notion de groupe*

La notion de groupe est essentielle dans notre approche. Le groupe offre un service en mettant en œuvre un protocole entre ses différentes entités. Le groupe est donc un élément désignable par les développeurs. Il permet de rassembler un ensemble d'entités participant à une tâche et de décrire de manière globale son comportement, en faisant abstraction de sa réalisation par des objets.

2.2.2. *Programme de groupe*

Un programme de groupe (voir grammaire figure 1) fournit le comportement du groupe d'objets réalisant l'application répartie. Il est divisé en méthodes représentant des sous-comportements particuliers. La composition du groupe (mot clé *membership*) est indiquée à l'aide d'un ensemble d'identificateurs permettant de désigner les objets membres. Nous supposons que ceux-ci existent, c'est-à-dire ont été instanciés, et que leur identité a été enregistrée (par exemple, dans un service de noms). Un invariant du programme (mot clé *environment*), exprimé sous la forme d'un prédicat épistémique peut être fourni. Il correspond à la définition du contexte d'exécution de l'algorithme réparti. Ces programmes manipulent des variables de groupe (déclarées par le mot clé *var*) qui correspondent à des structures de données réparties. Ce sont des agrégats de données localisées sur différents objets du groupe.

Les instructions d'un programme sont, soit des opérations d'affectation de variables de groupe, soit des appels de méthodes sur des groupes d'objets, soit des structures de contrôle de groupe. L'enchaînement séquentiel d'instructions est noté par un point-virgule. Une pré-condition et une post-condition peuvent être associées à chaque instruction. Elles fournissent, sous la forme de prédicats épistémiques, l'évolution du niveau de connaissance apportée par l'exécution de l'instruction. Elles constituent également un contrat qui caractérise cette exécution. La méthode *Main* correspond au traitement exécuté lors du lancement du programme.

2.2.3. *Structures de contrôle de groupe*

Les structures de contrôle de groupe orientent l'exécution d'un programme de groupe. Par analogie avec les structures de contrôle de l'algorithmique centralisée (condition, itération, etc.), elles définissent des schémas d'exécution au sein d'un groupe d'objets. Nous retenons deux structures de contrôle de groupe qui nous semble apparaître de façon récurrente dans de nombreuses applications : la condition répartie et l'itération répartie.

2.3. *Caractérisation des programmes par la logique épistémique*

Les invariants, les pré-conditions et les post-conditions mentionnés précédemment sont écrits à l'aide d'une logique modale dite logique épistémique de la connais-

```
<prog>::= prog_gr ident ';'
             <composition> <variables> <invariant>
             <methode> *
          end ident '.'

<composition>::= membership ':' '{' ident * '}'
<variables>::= ( var ( ident ':' <type> ';' )* ) | _
<invariant>::= ( environment <predicat> ) | _
<methode>::= method ident '(' <params> ')' <variables>
             begin <instrs> end ident ';'

<instrs>::= ( <pre> <instr> <post> ';' )*
<pre>::=  ( '{' <predicat> '}' ) | _
<post>::= ( '{' <predicat> '}' ) | _
<instr>::= ident <operateur> <expression>
         | ident '.' ident '(' <args> ')'
         | if <expression> then <instrs> else <instrs> end if
         | while <expression> do <instrs> end do
         ;
```

Figure 1. *Syntaxe des programmes de niveau groupe*

sance [MEY 95]. Son intérêt est d'introduire des modalités permettant de préciser le degré de répartition d'une connaissance dans un groupe.

Les différentes instructions du langage défini précédemment correspondent à des transitions faisant évoluer le groupe entre différents états successifs. En première approche, l'état d'un groupe d'entités réparties est un *n*-uplet regroupant les états locaux de chacune de ses entités et l'état de ses canaux de communication. Un état local ou l'état d'un canal de communication est vu comme une fonction entre un ensemble de variables et un ensemble de valeurs. Parmi tous les états globaux possibles, seuls certains ont un sens et peuvent être observés dans une exécution réelle. Le problème de la capture de tels états a reçu de nombreuses solutions, notamment avec les travaux de Chandy et Lamport [CHA 85]. Des propriétés sur les exécutions réparties sont alors exprimées sous forme de prédicats. Cette démarche impose que les variables prenant part à la définition des états soient accessibles.

Par ailleurs, les travaux de Fagin et al. [FAG 95] sur les systèmes multi-agents ont formalisé la notion de connaissance pour des ensembles d'entités réparties. Dans leur approche, une connaissance correspond à un fait, c'est-à-dire un prédicat, associé à une modalité décrivant la façon dont il est réparti. Plusieurs modalités sont proposées dont les plus importantes sont *D*, *K* et *E* respectivement appelées modalité de distribution, de connaissance et de connaissance de tous. Ainsi, *G* étant un groupe d'entités et ϕ un prédicat :

— $D_G\phi$ signifie qu'aucune entité de l'ensemble *G* ne connaît le fait ϕ (c'est-à-

dire, ne peut fournir sa valeur de vérité), mais qu'il existe un fait ψ qui permet au groupe d'inférer le fait ϕ,

— $K_i\phi$ signifie qu'un élément $i \in G$ connaît le fait ϕ,

— $E_G\phi$ signifie que tous les éléments de G connaissent le fait ϕ (en d'autres termes, ϕ est une connaissance de tous),

— $E_G^2\phi$ signifie que tous les éléments de G savent que tous connaissent le fait ϕ,

— $E_G^k\phi$ correspond quant à elle, à une connaissance de tous de niveau *k*: tout le monde sait que tout le monde sait . . . (*k* fois) que tout le monde connaît ϕ.

Cette approche, que nous adoptons, permet de définir la façon dont sont implantés les objets et d'envisager une application répartie comme un programme faisant évoluer l'état d'un groupe entre différents niveaux de connaissance. Chaque étape amène donc une évolution de cette connaissance et est annotée par une pré-condition et une post-condition épistémiques.

2.4. *Eléments de programmation*

2.4.1. *Variables de groupe*

Les aspects comportementaux des programmes de groupe s'appuient sur des structures de données dont il convient de préciser le type. Les types possibles pour ces données sont identiques à ceux que l'on trouve en programmation classique: types simples (entier, booléen, réel, caractère) et types composés (ensemble, arbre, pile). Lorsqu'une donnée est centralisée, c'est-à-dire localisée sur un seul objet, il existe différentes façons de la répartir dans un groupe. Par exemple, nous pouvons considérer que les données du groupe sont dupliquées ou réparties. Dans le cas de données dupliquées (totalement ou partiellement), tous les membres du groupe en possédent une copie dont la cohérence est assurée par un algorithme *ad-hoc*. Dans le cas de données réparties, aucun objet ne possède la connaissance complète de la structure, mais celle-ci est répartie dans le groupe. Par exemple, une façon de répartir une structure de type arbre, consiste à faire en sorte que chaque nœud connaisse son père et ses fils, mais qu'aucun objet ne connaisse la topologie complète de l'arbre.

Parmi ces structures de données, l'une d'entre elles présente un intérêt particulier: c'est l'ensemble de tous les objets du groupe. De nombreuses solutions pour la gestion de la composition d'un groupe sont proposées dans la littérature (voir par exemple [REN 96] ou [FEL 98]). Leur étude est hors du propos de cet article. Nous considérons seulement que nous sommes en présence d'une structure répartie de type ensemble possédant un mécanisme de gestion adéquat. Afin de simplifier les programmes, nous notons toutes les opérations ensemblistes (union, intersection, appartenance) sur ce type à l'aide de leurs symboles mathématiques habituels ($\cup, \cap, \in$).

Affectation d'une variable de groupe

L'affectation d'une valeur à une variable de groupe dépend du statut de cette dernière. Dans le cas de données dupliquées, l'affectation peut être effectuée de façon simple, par le possesseur d'une copie. L'algorithme de gestion de cohérence associé à la variable est alors chargé de propager cette mise à jour (voir par exemple [BUD 93] ou [SCH 93]). Dans le cas de structures de données réparties, le processus est plus complexe et nécessite, en général, une mise à jour différente sur chacun des objets possédant un fragment de la donnée. L'affectation est alors réalisée par un algorithme *ad-hoc*. Les algorithmes de parcours présentés aux paragraphes 4 et 5, peuvent servir de base à une telle tâche.

L'affectation de variables est notée par le symbole deux points égal ($:=$). L'initialisation des variables déclarées par un programme de groupe est considérée comme une simple affectation et est donc du ressort des méthodes définies dans ce programme. Le type ensemble étant couramment utilisé dans les programmes de groupe (par exemple, pour désigner le groupe lui-même ou certains sous-groupes jouant des rôles particuliers), nous retenons également l'opérateur de choix ensembliste deux points inclus ($:\subseteq$). Par exemple, l'expression $a :\subseteq b$ signifie qu'un sous-ensemble[1] de *b* est affecté à l'ensemble *a*.

2.4.2. *Instructions de groupe*

Les instructions d'un programme de groupe correspondent à des traitements coopératifs effectués par les objets composant le groupe. L'exécution d'une telle instruction correspond à la notion de phase. Intuitivement la phase délimite un intervalle de temps logique pendant lequel tous les objets du groupe prennent part à un même traitement réparti. Tous n'exécutent pas nécessairement le même traitement local au cours d'une phase mais, du point de vue de groupe, il existe un lien logique entre ces différents traitements. Par exemple, un protocole transactionnel de validation en deux phases [BER 87] met en jeu un groupe constitué d'une entité coordinatrice et de une ou plusieurs entités participantes et comprend une phase de vote et une phase de décision. La phase de vote consiste à s'assurer que tous les participants sont en mesure de réaliser la transaction. Cette instruction de groupe se traduit, pour l'entité coordinatrice, par une méthode qui contacte tous les participants et leur demande de réaliser une opération, et, pour les entités participantes, par une méthode qui évalue la faisabilité de l'opération demandée.

La figure 2 illustre la notion de phase dans le cadre de ce protocole. Cet exemple ne constitue pas une spécification exhaustive, mais fournit un premier niveau de présentation. Après une phase de vote chargée de déterminer la faisabilité de la transaction (méthode *Vote*), le protocole exécute une phase de décision (méthode *Décide*) dans laquelle le coordinateur prend la décision d'engager ou d'annuler la transaction. Nous

1. Cet opérateur de choix est non déterministe au sens où la composition exacte du sous-ensemble choisi n'est pas précisée (un exemple d'utilisation de cet opérateur est fourni au paragraphe 4). Sa mise en œuvre nécessite une étape supplémentaire de raffinement afin de préciser la façon dont les éléments à inclure sont effectivement choisis.

notons le résultat de la phase de vote à l'aide de la variable booléenne *cond*. Ce programme fait évoluer le groupe G composé du coordinateur et des participants à la transaction, de la façon suivante :

1. initialement, l'état transactionnel est réparti : personne dans le groupe ne le connaît, mais il existe un algorithme (le protocole de validation lui-même) et des conditions initiales qui permettent de le déterminer,

2. après la phase de vote, l'état transactionnel est connu du seul coordinateur mais pas des participants,

3. finalement, à la fin du protocole, participants et coordinateur connaissent l'état transactionnel et ils savent que les autres le connaissent également (c'est donc une connaissance de tous de niveau 2).

```
prog_gr Valide2Phases;
membership
  G ≙ {coordinateur, participant_1, ... , participant_n}
method Main()
var cond : boolean;
begin
  {D_G(et)}
    cond := G.Vote();
  {K_coordinateur(et)}
    G.Décide(cond);
  {E^2_G(et)}
end Main;
end Valide2Phases.
```

Figure 2. *Exemple d'enchaînement de phases*

Nous définissons et comme un prédicat représentant l'état transactionnel [2] : il vaut vrai lorsque la transaction est validable et faux sinon. Nous associons respectivement aux trois étapes précédentes, les expressions épistémiques $D_G(et)$, $K_{coordinateur}(et)$ et $E^2_G(et)$. Ces trois expressions expriment respectivement, que l'état transactionnel est une connaissance distribuée, puis que cette connaissance est instanciée chez le coordinateur, et qu'enfin, c'est une connaissance de tous de niveau 2.

Exécution d'une instruction de groupe

L'exécution d'une instruction de groupe concerne l'ensemble des objets du groupe. Elle constitue ainsi une phase dans le déroulement de l'algorithme réparti. Chaque objet exécute un traitement et les débuts d'exécution de ces traitements s'apparentent, pour les objets concernés, au début de la phase. L'union des états locaux des objets au

2. La variable booléenne *cond* est la traduction opérationnelle de ce prédicat.

début de chaque phase constitue un état global inévitable [CHA 85] pour le groupe. Cet état global peut être utilisé dans un processus de déverminage [PLA 95], de rejeu ou de détection de propriétés [BAB 86].

3. Structures de contrôle de groupe

3.1. *La condition répartie*

La condition répartie sélectionne une action à réaliser dans un groupe. C'est par exemple, le test de décision de validation ou d'annulation dans le protocole précédent. De manière plus générale, la condition répartie permet d'exprimer différentes alternatives. Pour cela, une procédure de choix sur l'état global du groupe et un ensemble de comportements alternants sont définis. Cette instruction est l'équivalent des tests ou des sélections multiples des langages de programmation.

Pour réaliser ce comportement, il est nécessaire d'identifier les objets qui participent à la condition répartie et de trouver un moyen d'évaluer cette condition. Celle-ci peut être locale ou globale. Dans le premier cas, son évaluation est immédiate. Dans le second cas, elle concerne plusieurs objets. Il faut alors appliquer un algorithme de collecte d'état global [CHA 85], assurer l'évaluation du prédicat associé et diffuser le résultat de cette évaluation. La mise en œuvre de ce processus peut se faire de différentes façons : soit un coordinateur unique est désigné, soit plusieurs objets évaluent la condition et appliquent un algorithme de consensus [GUE 97]. Il faut ensuite identifier les ensembles d'objets concernés par les comportements alternants et leur notifier le résultat de l'évaluation afin qu'ils entreprennent un de ces comportements.

Nous notons la condition répartie à l'aide des mots clés *if-then-else*. La figure 3 illustre son utilisation dans le cadre du protocole transactionnel de validation à deux phases. La méthode *Décide* de la figure 2 est ainsi remplacée par l'alternative suivante : si la condition correspondant au vote des participants est vérifiée, alors la transaction est validée (appel de la méthode *Valide*), sinon elle est annulée (appel de *Annule*). Il s'agit ici d'une collecte d'état global par le coordinateur. On peut également ajouter que ce second exemple est un raffinement du premier.

3.2. *L'itération répartie*

L'itération répartie est la généralisation à un niveau réparti, de la boucle ou de l'exécution de calculs répétitifs [HEL 89]. Elle est utilisée chaque fois qu'un comportement global doit être itéré. Une itération est composée de la description d'un calcul (le pas de l'itération) et d'une condition d'arrêt évaluée à chaque pas. Elle est aussi caractérisée par un invariant qui décrit l'état des variables au début de chaque pas. Lorsque la condition d'arrêt est atteinte, la conjonction de ce dernier et de l'invariant donne sa signification au résultat calculé.

La réalisation de ce comportement requiert l'identification des objets participants à

```
prog_gr Valide2Phases;
membership
  G ≙ {coordinateur, participant_1, ... , participant_n}
method Main()
var cond: boolean;
begin
  {D_G(et)}
    cond:= G.Vote();
  {K_coordinateur(et)}
    if cond = true
    then G.Valide();
    else G.Annule();
    end if;
  {E^2_G(et)}
end Main;
end Valide2Phases.
```

Figure 3. *Exemple de condition répartie*

l'itération et le comportement à itérer sur ces objets. On réutilise le principe d'évaluation de la condition répartie pour identifier les objets participant à la condition d'arrêt, pour trouver un moyen d'évaluer cette condition et pour notifier aux objets concernés sa valeur. Il faut, de plus, définir la procédure d'initialisation et la procédure de mise à jour de cette condition. Dans certains cas, l'itération répartie est une boucle infinie. C'est le cas des algorithmes de routage, comme RIP [MCQ 80] ou OSPF [MOY 91]. Comme pour la condition répartie, le test d'arrêt de l'itération peut être local ou global.

Nous notons l'itération répartie à l'aide des mots clés *while-do*. La figure 4 présente l'exemple d'un algorithme de routage de type RIP qui recalcule (méthode *RecalculTables*) périodiquement les tables de routage localisées dans les objets du groupe. Avant cela, une phase de diffusion des tables (méthode *DistribuerTables*) et respectivement, de réception de ces tables (méthode *RecevoirTables*), sont nécessaires. L'invariant de boucle consiste à exprimer que la structure de données réparties représentant les chemins de moindre coût dans le graphe est remise à jour périodiquement par l'exécution de ces trois méthodes. Cette structure correspond, dans le programme de la figure 4, à la variable de groupe *tables*: c'est un tableau de matrices de routage qui contient les versions successives des tables de routage[3]. La variable *tablesPairs* contient les tables de routage diffusées par les objets à leurs pairs. Au début d'une itération, les tables de routage des époques inférieures à l'époque courante (variable *n*) sont connues de tous (prédicat épistémique ligne 1 figure 4). Après les phases de distribution et de collecte, chaque objet connaît en plus, les tables de ses pairs (ligne 4 figure 4). Finalement, après l'étape de recalcul, les tables de l'époque courante sont

3. Pour simplifier la présentation, nous supposons ici que les versions obsolètes des tables de routage sont conservées dans un tableau de dimension infinie. En pratique, cet historique est purgé à chaque itération.

```
   prog_gr RoutageReparti;
   membership
     groupe ≙ {site_1 , ... , site_n}
   var
     tables : array of topologieRoutage;
     tablesPairs : array of topologieRoutage;
     n : integer = 0;
   method Main()
   begin
     while true do
1:     {∀i < n, E_groupe(tables[i])}
2:       groupe.DistribuerTables();
3:       tablesPairs := groupe.RecevoirTables();
4:     {∀s ∈ groupe, K_s(tablesPairs[s])}
5:       tables := groupe.RecalculTables();
6:     {∀i ≤ n, E_groupe(tables[i])}
7:       n := n+1;
     end do;
   end Main;
   end RoutageReparti.
```

Figure 4. *Exemple d'itération répartie*

connues de tous (ligne 6 figure 4).

4. Premier exemple : parcours récursif d'un groupe d'objets

Dans ce paragraphe, nous illustrons l'utilisation de notre langage en présentant la définition d'un algorithme de parcours récursif d'un groupe d'objets. La version complète de cette présentation est disponible dans [SEI 97]. Ce type d'algorithme est utilisé par exemple, pour diffuser une information, collecter un état global ou encore initialiser une structure de données répartie. Il correspond, en algorithmique répartie, à la notion de vague récursive [FLO 93][TEL 94]. Il est utilisé, en général, avec des groupes d'objets de taille importante pour lesquels il n'est pas envisageable que chaque objet connaisse directement tous les membres du groupe. Les très grands réseaux en sont un exemple.

4.1. *Présentation générale de l'algorithme*

On suppose que le groupe d'objets est organisé sous la forme d'un graphe quelconque. Chaque objet connaît donc un ensemble d'objets voisins avec qui il est en

mesure de communiquer. Plusieurs variantes de l'algorithme de parcours sont envisageables. Un seul objet ou plusieurs objets peuvent démarrer le parcours. On parle d'objets initiateurs, par opposition aux objets participants. Afin de simplifier la présentation, nous nous limitons au cas où un seul objet déclenche le parcours. Ce dernier peut être propagé séquentiellement ou concurremment. Dans le premier cas, les objets propagent le parcours à un seul de leurs voisins, tandis que dans le second cas, plusieurs voisins sont concernés. La description que nous fournissons dans la suite recouvre ces deux cas.

L'objet initiateur démarre l'algorithme en propageant, par invocation de méthode, le parcours vers ses voisins. Lorsque la propagation atteint un objet, celui-ci poursuit le parcours parmi ses propres voisins. Le parcours se propage ainsi de proche en proche dans le graphe d'objets (d'où le terme de parcours récursif), jusqu'à ce que tous les objets aient été visités. Chaque objet a éventuellement plusieurs voisins, et reçoit donc plusieurs propagations du parcours. Il est nécessaire qu'une seule de ces propositions soit acceptée. En général, la première est choisie et les suivantes sont retournées. Lorsque un objet reçoit les retours de propagation de tous ses voisins, il retourne à son tour la propagation qu'il a reçue. Lorsque l'initiateur reçoit les retours de tous ses voisins, il sait que l'algorithme est terminé. Deux phases peuvent être distinguées dans cet algorithme : la phase de descente qui correspond à la propagation du parcours, puis la phase de remontée lorsque tous les objets ont été visités et que le parcours reflue vers l'initiateur.

4.2. *Hypothèses et but de l'algorithme*

4.2.1. *Hypothèses de bon fonctionnement*

Ce paragraphe détaille les hypothèses de bon fonctionnement de l'algorithme. Elles représentent, à l'aide d'un prédicat épistémique, l'invariant à respecter afin que celui-ci fonctionne correctement. Par exemple, il est nécessaire que le graphe d'objets reste connexe pendant toute la durée du parcours. Ces hypothèses, implicites dans la plupart des présentations actuelles, nous semblent indispensables à la sûreté de conception de ces programmes, et constituent une étape initiale essentielle en vue de la définition d'un système formel de preuve pour des comportements répartis.

Nous disposons d'un graphe orienté $G = (groupe, liaisons)$ où *groupe* est un ensemble d'objets et *liaisons* un ensemble de liaisons. Chaque objet connaît son propre identifiant. Les identifiants sont complètement ordonnés. Nous supposons également qu'il n'y a pas d'homonymes et que le graphe est défini par la connaissance du voisinage. Nous utilisons les notations suivantes : $id : groupe \rightarrow nat$ est une fonction qui permet d'identifier les objets et $voisins : groupe \rightarrow 2^{groupe}$ est une fonction qui définit le voisinage des objets. Les hypothèses de bon fonctionnement *HR* s'expriment à l'aide de la conjonction des six prédicats suivants :

— **H1** : Il n'y a pas d'homonymes.
$\forall i,j \in groupe, i \neq j \Rightarrow id(i) \neq id(j)$

— **H2** : Il existe un ordre total (noté $\leq$) sur les identifiants des objets.
$\forall i,j \in groupe, id(i) \leq id(j) \vee id(j) \leq id(i)$

— **H3** : Les liens de communication sont symétriques.
$\forall i,j \in groupe, j \in voisins(i) \Rightarrow i \in voisins(j)$

— **H4** : Le graphe est connexe, c'est-à-dire que l'on peut atteindre tous les objets à partir de n'importe lequel d'entre eux. La notation $voisins^k(i)$ désigne l'ensemble des voisins à distance *k* de l'objet *i*.
$\forall i \in groupe, \cup_{k=0}^{\infty} voisins^k(i) = groupe$

— **H5** : Chaque objet connaît son identifiant.
$\forall i \in groupe, K_i(id(i))$

— **H6** : Le graphe est défini localement par la connaissance du voisinage.
$\forall i \in groupe, \forall j \in voisins(i), K_i(id(j))$

4.2.2. *But de l'algorithme*

L'algorithme permet à l'initiateur d'apprendre que tous les objets ont été visités lors du parcours. En notant *visités* l'ensemble des objets visités par le parcours, ce but s'exprime à l'aide du prédicat : $K_{initiateur}(visités = groupe)$.

4.3. *Programme de groupe*

La figure 5 présente le programme de groupe *ParcoursRecursif* réalisant un tel algorithme. Le groupe est composé d'un initiateur et de participants et la variable *visités* représente l'ensemble des objets visités par le parcours. La méthode *Main* déclenche la récursion avec l'objet initiateur (ligne 10 figure 5).

La méthode *Rec* représente, au niveau groupe, une étape de la récursion. La variable *ajoutés* représente l'ensemble des objets visités par l'étape courante de la récursion. Cet ensemble est ajouté au fur et à mesure, à l'ensemble *visités* (ligne 1 figure 5). Si tous les objets ont été visités, alors la méthode se termine[4]. Sinon, de nouveaux objets sont choisis (ligne 3) pour poursuivre le parcours (appel récursif de la méthode *Rec*, ligne 6). Un pré-traitement et un post-traitement (méthodes *PréTraitement* et *PostTraitement*, lignes 4 et 8), liés aux actions que l'on souhaite réaliser lors du parcours, sont appliqués à ces nouveaux objets. Ceux-ci sont choisis parmi les objets qui n'ont pas encore été visités et qui sont voisins d'objets visités (la fonction $d : groupe \times 2^{groupe} \rightarrow nat$ fournit la distance minimale d'un objet à un ensemble d'objets). L'appel récursif de la méthode *Rec* est annoté par les pré-condition et post-condition des lignes 5 et 7. Le premier prédicat signifie que tous les objets ajoutés lors de l'étape courante savent qu'ils sont visités. Le second spécifie, qu'après l'appel, ils savent en plus que les nouveaux objets choisis parmi leurs voisins l'ont également été. Finalement, la post-condition du premier appel de la méthode *Rec* reprend le but global de l'algorithme (ligne 11).

4. Cette condition répartie est implantée de la façon suivante : chaque objet décide localement si tous ses voisins ont été visités ou non (voir description au paragraphe 4.1).

```
   prog_gr ParcoursRecursif;
   membership
     groupe ≙ {initiateur, participant_1 , ... , participant_n}
   var visités: set of object = ∅;
   environment
     HR ≙ H1 ∧ H2 ∧ H3 ∧ H4 ∧ H5 ∧ H6

   method Rec( ajoutés: set of object )
   var ajout, nouveaux: set of object;
   begin
1:   visités:= visités ∪ ajoutés;
2:   if visités ⊂ groupe then
3:       nouveaux :⊆ ((groupe − visités) ∩ voisins(visités));
4:       groupe.PréTraitement( nouveaux );
5:     {∀s ∈ ajoutés, K_s(s ∈ visités)}
6:       groupe.Rec(nouveaux);
7:     {∀s ∈ ajoutés, K_s(({s} ∪ (nouveaux ∩ voisins({s}))) ⊆ visités)}
8:       groupe.PostTraitement( nouveaux );
9:   end if;
   end Rec;

   method Main()
   begin
10:    groupe.Rec({initiateur});
11: {K_initiateur(visités = groupe)}
   end Main;

   end ParcoursRecursif.
```

Figure 5. *Programme de groupe du parcours récursif*

La figure 6 illustre une étape d'exécution possible du parcours : en partant de l'objet initiateur 1, les objets 2 et 3 ont été ajoutés au cours de cette étape. L'ensemble *nouveaux* est donc un sous-ensemble de l'ensemble des objets 4 et 5 (qui sont les nouveaux voisins de l'ensemble *visités*). Il est important de noter qu'au niveau des objets, l'asynchronisme des propagations fait, qu'au cours d'une étape, un sous-ensemble des objets voisins (et non pas systématiquement tous les objets voisins) est choisi. Ici, par exemple, seul l'objet 4 est choisi.

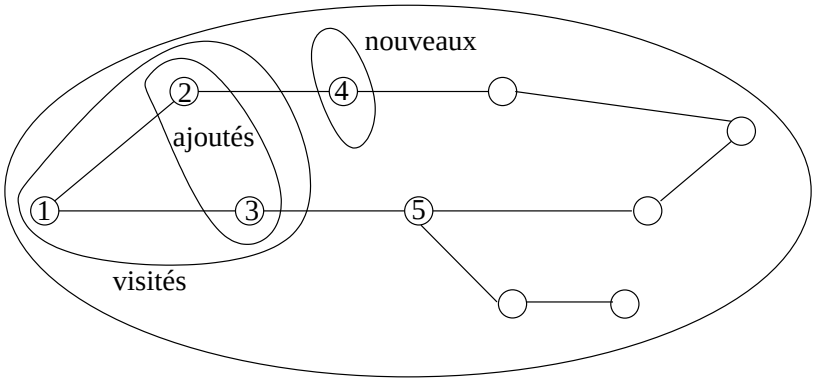

Figure 6. *Etape d'exécution possible d'un parcours récursif*

5. Deuxième exemple : parcours itératif d'un groupe d'objets

Ce paragraphe présente une deuxième version de l'algorithme de parcours d'un groupe d'objets. Dans la version précédente, chaque objet propage le parcours vers ses voisins. Ici, ce sont les objets qui, par scrutation de l'état de leurs voisins, décident si un parcours est en cours et s'il doit être propagé. La boucle de scrutation se poursuit tant que tous les objets n'ont pas été visités (d'où le terme parcours itératif).

Dans un premier temps, nous présentons le problème de manière informelle (paragraphe 5.1). Les paragraphes 5.2 et 5.3 fournissent respectivement, le but et la spécification de niveau groupe de l'algorithme. Enfin, certains éléments permettant la mise en œuvre au niveau objet sont introduits au paragraphe 5.4.

5.1. *Présentation générale de l'algorithme*

Comme précédemment, nous disposons d'un ensemble d'objets organisé selon une structure de graphe. Chaque objet vérifie périodiquement l'état de ses voisins. Ce contrôle d'activité joue un rôle similaire à celui d'un détecteur de pannes [CHA 91].

Le comportement de cet algorithme se poursuit tant que l'état du groupe n'est pas stable. De manière continue, les objets collectent l'état de leurs voisins et modifient leur état en fonction de cette collecte. Dans cet algorithme, le groupe peut avoir trois états : $\{repos, visite, retour\}$. L'état initial est *repos*. Le changement d'état local $repos \rightarrow visite$ sur un objet du groupe initie le parcours. Afin de simplifier la présentation, nous supposons qu'il y a un seul initiateur simultanément dans le groupe. Les objets s'apercevant du changement d'état local d'un de leurs voisins, modifient leur propre état local. Ainsi, le changement d'état se propage de proche en proche, dans tout le groupe. Le parcours reflue lorsque tous les objets ont été visités. Les objets passent alors de l'état *visite* à l'état *retour*. Lorsque l'initiateur passe à l'état *retour*, il sait que le parcours a atteint tous les objets et que le groupe est à l'état *retour*.

5.2. *Hypothèses et but de l'algorithme*

Les hypothèses de bon fonctionnement définies au paragraphe 4.2.1 portent sur la topologie du groupe d'objets. Elles restent valables ici. Le prédicat *HR* constitue donc l'invariant du parcours itératif.

Par ailleurs, le but de l'algorithme est de faire en sorte que l'initiateur du parcours apprenne que celui-ci est terminé, c'est-à-dire que tous les objets ont été atteints. Ce but s'exprime de la façon suivante : $K_{initiateur}(état = retour)$.

5.3. *Programme de groupe*

L'itération répartie est à la base de la description de cet algorithme : tant que l'état *retour* n'est pas atteint, le groupe échange des connaissances sur son état et modifie celui-ci en fonction des connaissances acquises.

La figure 7 présente le programme de groupe correspondant à ce comportement. L'itération répartie se compose d'une phase (méthode *CalculEtatSuivant*) et d'une condition répartie (le test d'arrêt de la boucle). La propriété de stabilité à atteindre correspond au but global défini au paragraphe 5.2. C'est également le test d'arrêt de la boucle. La variable *état* représente l'état du groupe. Tous les objets du groupe exécutent de manière asynchrone (c'est-à-dire indépendamment les uns des autres et sans se coordonner), la phase *CalculEtatSuivant*.

```
prog_gr ParcoursItératif;
membership
  groupe ≙ {initiateur, participant_1 , ... , participant_n}
var état : enum {repos,visite,retour};
environment
  HR ≙ H1 ∧ H2 ∧ H3 ∧ H4 ∧ H5 ∧ H6

method Main()
begin
    while état ≠ retour do
      {K_initiateur(état ≠ retour)}
        état := voisins.CalculEtatSuivant();
    end do;
  {K_initiateur(état = retour)}
end Main;
end ParcoursItératif.
```

Figure 7. *Programme de groupe d'un parcours itératif*

Analogie avec le parcours récursif

On peut noter que les programmes des figures 5 et 7 ont le même but : parcourir l'ensemble des objets du groupe. Néanmoins, ce but est atteint avec des styles différents. Le premier programme adopte une démarche orientée par le contrôle (l'appel récursif d'une méthode), tandis que le comportement du second est plus orienté par les données (le test de la valeur *retour* pour la variable *état*). L'analogie peut être poursuivie en notant que, bien que syntaxiquement différents, les prédicats finals des deux algorithmes, $K_{initiateur}(visités = groupe)$ dans le premier cas et $K_{initiateur}(état = retour)$ dans le second, sont équivalents. En effet, il suffit de considérer pour cela que la variable *visités* comprend l'ensemble des objets dont l'état est *retour*.

5.4. *Définition d'une étape de l'itération*

Nous ne donnons pas le code de la méthode *CalculEtatSuivant*, mais nous définissons un ensemble de règles permettant de gérer les changements d'état. Le programme de groupe de la figure 7 particularise deux rôles distincts : celui de l'initiateur et celui des participants. Chaque objet gère une version locale de la variable de groupe $état$: nous la notons $état_i$. Nous introduisons également les variables locales $fils_i$. Elles correspondent, pour un objet *i*, à l'ensemble des objets qui modifient leur état de *repos* vers *visite* après avoir constaté un changement d'état sur l'objet *i*.

Chaque objet exécute une version locale du programme de niveau groupe. La phase de calcul de l'état suivant consiste à interroger l'état des objets voisins et à faire évoluer la variable locale $état_i$ selon les règles suivantes :

— $état_i = repos \wedge \forall j \in voisins_i, état_j = repos \Rightarrow état_i = repos$
— $état_i = repos \wedge \exists j \in voisins_i, état_j = visite \Rightarrow état_i = visite$
— $état_i = visite \wedge fils_i = \emptyset \Rightarrow état_i = retour$
— $état_i = visite \wedge \forall j \in fils_i, état_j = retour \Rightarrow état_i = retour$
— $état_i = retour \wedge \forall j \in voisins_i, état_j = retour \Rightarrow état_i = retour$

Initialement, tous les objets sont à l'état *repos*. L'initiateur passe à l'état *visite*, puis par propagation, ses voisins également. Un objet passe à l'état *retour* lorsqu'il n'a pas de fils, ou lorsque ses fils sont déjà à l'état *retour*.

6. Comparaison à des travaux existants

6.1. *La notion de groupe*

La notion de groupe n'est pas nouvelle. Par exemple, dans les travaux de normalisation du modèle ODP [ISO96], on trouve la notion d'objet composite, et dans ceux du projet SOR [MAK 94], la notion d'objet fragmenté. Même s'ils ne font pas apparaître

cette notion dans le même contexte, ces travaux suivent les mêmes approches. De nombreux environnements utilisent également de tels groupes lorsqu'il s'agit de définir des applications tolérantes aux fautes, de communications à plusieurs, ou de réplication de données. Les travaux autour des systèmes OGS [GUE 98], Psync [MIS 89], Amoeba [WOO 93], Emerald [BLA 93] ou Arjuna [LIT 94] en sont des exemples (voir également [HO 98] pour un recensement et une étude comparative des différents environnements orientés objets existants pour la programmation répartie). Ces systèmes fournissent des primitives et des protocoles permettant, entre autres, de gérer la composition des groupes, d'y diffuser de l'information selon différentes qualités de service (par exemple ordonnée causalement ou totalement), ou d'y opérer un consensus. Notre approche ne propose pas un nouvel environnement ou de nouvelles primitives dans ce domaine, mais offre un langage pour définir les comportements mis en œuvre dans de tels environnements. Les systèmes cités précédemment constituent donc un support naturel pour l'implantation des programmes de groupe présentés dans cet article.

La notion de groupe d'objets est également présente dans certains travaux portant sur la réflexivité. Cette notion désigne la capacité d'un système « à raisonner et à agir sur lui-même » [MAE 87]. Les systèmes réflexifs sont de plus en plus utilisés, notamment en informatique répartie, car ils permettent, via des méta programmes, de contrôler le comportement des programmes habituels. Parmi les nombreux systèmes réflexifs existants, les travaux autour de ABCL/R2 [MAT 91], OpenC++ v1 [CHI 93] ou PlasmaR [MIG 99] présentent des points communs avec notre approche. Ils introduisent la notion de réflexivité de groupe, qui vise à contrôler le comportement, non pas d'un seul objet, mais d'un groupe d'objets. Les structures de contrôle que nous introduisons dans cet article sont des exemples types de comportements qui peuvent être implantés avec de tels systèmes.

6.2. *Les structures de contrôle*

L'incorporation à notre langage de groupe de structures de contrôle comme la condition et l'itération réparties, partage également des points communs avec le domaine des patrons de conception (en anglais *design patterns*). Il s'agit en effet de définir des schémas de solutions génériques adaptables à des contextes récurrents et d'améliorer la structuration des applications. Ces travaux ont été initiés par Gamma et al. [GAM 95]. D'autres auteurs ont étendu cette approche à des environnements répartis. On peut citer les travaux liés à CORBA [MOW 97], ou encore à Eiffel Parallel [JEZ 96]. Nous nous intéressons essentiellement aux patrons dits comportementaux c'est-à-dire, selon les termes de Gamma et al., à des patrons qui caractérisent « la façon selon laquelle les classes ou les objets interagissent et se répartissent les responsabilités ».

Notre démarche est proche au sens où nous identifions dans les algorithmes répartis, des schémas d'interaction et de contrôle génériques. De la même façon que des structures algorithmiques (par exemple boucle, conditionnelle) sont disponibles dans

les langages séquentiels, nous proposons ainsi un langage adapté à l'algorithmique répartie. Des travaux ont déjà été entrepris dans ce sens, il y a une dizaine d'années. Citons par exemple, Helary et Raynal [HEL 89] qui ont proposé un schéma abstrait d'itération répartie. D'autres ont proposé une description des problèmes d'algorithmique répartie par blocs de construction [GAF 86]. Alors que ces travaux ont été menés dans un contexte de processus communiquant en mode message, notre approche est novatrice au sens où nous les unifions dans le cadre des programmes de groupe.

6.3. *Expression de la répartition*

La troisième originalité de notre approche est de caractériser le comportement des structures de contrôle réparties à l'aide de prédicats de logique épistémique. Cette démarche répond à des besoins similaires à ceux de la programmation par contrat [MEY 92] et des langages de définition de contraintes comme OCL [OCL97] : les comportements sont associés à des pré-conditions et des post-conditions logiques qui en précisent la spécification. La logique épistémique augmente le pouvoir d'expression de ces approches avec des modalités décrivant explicitement la façon dont une connaissance est répartie au sein d'un groupe.

Notre démarche, bien que moins formelle, partage également certains points communs avec des approches comme UNITY [MIS 88] ou TLA [LAM 94] qui permettent de décrire et de raisonner sur des programmes parallèles et distribués. Ces formalismes proposent des modèles de programmation classique, alors que nous nous plaçons au niveau groupe. Ils s'appuient sur la logique temporelle, alors que nous utilisons la logique épistémique pour exprimer les propriétés des programmes. Ces deux logiques modales se différencient essentiellement par le fait que la première raisonne sur les évolutions futures d'un système, tandis que la seconde s'intéresse aux évolutions passées. Ce choix nous semble intéressant pour la description de programmes répartis qui manipulent par essence des structures de données réparties et où les états globaux cohérents tels que les ont défini Chandy et Lamport dans [CHA 85], ne sont connus qu'*a posteriori* (c'est-à-dire, tout état capturé appartient au passé). Comme nous le mentionnons dans le paragraphe suivant, l'unification de ces deux logiques dans un même système formel constitue une perspective de notre travail.

7. Conclusion

Cet article propose un langage de description d'algorithmes répartis. Le paradigme majeur introduit consiste à considérer de tels algorithmes non pas seulement comme des ensembles de processus ou d'objets communiquants, mais comme de véritables programmes qui s'exécutent sur des groupes d'objets répartis. Nous définissons pour cela la notion de programme de groupe. Un tel programme a alors pour vocation de décrire et de synthétiser l'ensemble des comportements entrepris par le groupe d'objets qui participe à la réalisation de l'algorithme réparti.

Nous avons défini, au paragraphe 2, la syntaxe de ce langage. Deux points majeurs font, selon nous, son originalité : il s'appuie sur un petit nombre de structures de contrôle réparties bien définies, et il utilise une logique modale épistémique [MEY 95] [FAG 95], pour caractériser l'exécution des programmes. Les deux structures de contrôle présentées au paragraphe 3 sont la condition et l'itération répartie. Elle généralisent pour des groupes d'objets répartis, les notions de condition *if-then-else* et de boucle *while* des langages de programmation habituels. Elles représentent des comportements génériques rencontrés fréquemment dans les algorithmes répartis. Il est donc intéressant d'en systématiser la définition afin de faciliter leur utilisation. Elles sont utilisées comme instructions dans les programmes de groupe. La seconde originalité de notre langage réside dans l'emploi de la logique épistémique. Celle-ci étend la logique du premier ordre avec des modalités décrivant la façon dont une connaissance est répartie. Elle fournit donc une vision de haut niveau d'un algorithme réparti et permet d'en modéliser l'évolution en terme de connaissances échangées et manipulées. Les prédicats épistémiques sont utilisés comme invariants des programmes et comme pré-condition et post-condition des instructions. Ils constituent donc des contrats devant être respectés par l'exécution du programme. Les paragraphes 4 et 5 fournissent des exemples d'utilisation du langage de niveau groupe, des structures de contrôle et de la logique épistémique.

De nombreuses perspectives sont envisageables, aussi bien au plan pratique, qu'au plan théorique. En premier lieu, la réalisation d'un interpréteur pour le langage de niveau groupe permettra d'en valider le bien-fondé.

Par ailleurs, les structures de contrôle décrites dans l'article réalisent des mécanismes de synchronisation au sein d'un groupe d'objets répartis. C'est donc une synchronisation de type inter-objets. D'autres aspects nécessitent d'être abordés comme par exemple la synchronisation intra-objet qui consiste à coordonner les activités concurrentes au sein d'un objet. Dans des travaux précédents [SEI 97, SEI 98], nous avons proposé pour cela un langage et une sémantique à base de logique temporelle d'actions [LAM 94]. Nous envisageons d'unifier ces deux aspects (synchronisations intra-objet et inter-objets) dans un formalisme unique. A terme, cela permettra d'aboutir à un système formel permettant de décrire complètement la synchronisation d'une application répartie et de conduire un certain nombre de preuves sur ces aspects. Cette perspective nécessite également de choisir, parmi toutes celles existantes (voir [MEY 95]), une axiomatisation de la logique épistémique adaptée à cet objectif.

Une troisième perspective consiste à poursuivre l'intégration des structures de contrôle et de données réparties dans le cadre d'un formalisme unique. L'objectif est d'aboutir à une définition claire de la notion d'objet réparti dans laquelle le groupe d'objets est vu comme un objet à part entière. Même si des travaux comme SOS [MAK 94], Globe [STE 99] ou AspectIX [HAU 98] introduisent la notion proche d'objet fragmenté, et si dans [CHA 99] et [DUC 99], nous tentons de proposer un début de réponse avec la notion de type abstrait coopératif, il nous semble que de nombreux travaux restent à mener pour aboutir à cet objectif.

8. Bibliographie

[BAB 86] BABAOGLU O., FROMENTIN E. et RAYNAL M., « A Unified Framework for the Specification and Run-time Detection of Dynamic Properties in Distributed Computations ». *The Journal of Systems and Software*, vol. 3, n° 33, p. 287–298, 1986.

[BER 87] BERNSTEIN P., HADZILACOS V. et GOODMAN N., *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BLA 93] BLACK A. et IMMELL M., « Encapsulating Plurality ». In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, vol. 707 de *Lecture Notes in Computer Science*, p. 57–79. Springer-Verlag, Juillet 1993.

[BON 96] BONNET L., DUCHIEN L., FLORIN G. et SEINTURIER L., « Some Specification Steps of a Spanning Tree Algorithm with an Object-Oriented Approach ». In *Proceedings of the 1st IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96)*, p. 115–131. Chapman & Hall, 1996.

[BOO 94] BOOCH G., *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings Plub. Company, 1994. 2nd Edition.

[BOO 97] BOOCH G., RUMBAUGH J. et JACOBSON I., « *The Unified Method for Object-Oriented Development* ». Rational Software Corporation, 1997. Version 1.0.

[BUD 93] BUDHIRAJA N., MARZULLO K., SCHNEIDER F. et S.TOUEG, *The Primary-Backup Approach*. Distributed Systems, ACM Press Book, 1993.

[CHA 85] CHANDY K. et LAMPORT L., « Distributed Snapshots: Determining Global States of Distributed Systems ». *ACM Transactions on Computer Systems*, vol. 3, n° 1, p. 63–75, Février 1985.

[CHA 91] CHANDRA T. et TOUEG S., « Unreliable Failure Detectors for Reliable Distributed Systems ». In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC'91)*, p. 325–340, Août 1991.

[CHA 99] CHAMPAGNOUX P., DUCHIEN L., ENSELME D. et FLORIN G., « Cooperative Abstract Data Types ». In *Proceedings of the 14th International Workshop on Algebraic Development Techniques (WADT'99)*, Septembre 1999.

[CHI 93] CHIBA S. et MATSUDA T., « Designing an Extensible Distributed Language with Meta-Level Architecture ». In *Proceedings of ECOOP'93*, vol. 707 de *Lecture Notes in Computer Science*, p. 482–501. Springer-Verlag, Octobre 1993.

[DUC 99] DUCHIEN L.. « Modèles de programmation, services systèmes et réflexivité pour la coopération de groupes d'objets répartis ». Habilitation à diriger des recherches, Université Joseph Fourier, Grenoble, Décembre 1999.

[FAG 95] FAGIN R., HALPERN J., MOSES Y. et VARDI M., *Reasoning about Knowledge*. MIT Press, 1995.

[FEL 98] FELBER P., GUERRAOUI R. et SCHIPER A., « The Implementation of a CORBA Group Communication Service ». *Theory and Practice of Object Systems*, vol. 4, n° 2, 1998.

[FLO 93] FLORIN G., GOMEZ R. et LAVALLÉE I., « Recursive Distributed Programming Schemes ». In *Proceedings of the International Symposium on Autonomous Decentralized Systems (ISADS'93)*, Avril 1993.

[GAF 86] GAFNI E., « Perspectives on Distributed Network Protocols: A Case for Building Blocks ». In *Proceedings IEEE Military Communications Conference*, Octobre 1986.

[GAM 95] GAMMA E., HELM R., JOHNSON R. et VLISSIDES J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GRI 97] GRIMES R., *DCOM Programming*. Worx Press, 1997.

[GUE 97] GUERRAOUI R. et SCHIPER A., « Consensus: The Big Misunderstanding ». In *Proceedings of 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems*, p. 183–188. IEEE Computer Society Press, Octobre 1997.

[GUE 98] GUERRAOUI R., FELBER P., GARBINATO B. et MAZOUNI K., « System Support for Object Groups ». In *Proceedings of the 13th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'98)*, Octobre 1998.

[HAU 98] HAUCK F., BECKER U., GEIER M., MEIER E., RASTOFER U. et STECKERMEIER M., « AspectIX: An Aspect-Oriented and CORBA-Compliant ORB Architecture ». In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processingthe (Middleware'98)*, Mars 1998.

[HEL 89] HELARY J. et RAYNAL M., « Un schéma abstrait d'itération répartie ». *Techniques et Sciences Informatiques*, vol. 8, n° 3, p. 259–267, Mars 1989.

[HO 98] HO W. et JÉZÉQUEL J., « Object-Oriented Frameworks for Distributed Systems: A Survey ». Rapport technique 1223, IRISA, Décembre 1998.

[ISO96] « ISO/IEC IS 10746-1, IUT-T Rec X901, ODP Reference Model Part-1. Overview and Guide to Use », Mai 1996.

[JEZ 96] JEZEQUEL J. et PACHERIE J., « Parallel Operators ». In *Proceedings of ECOOP'96*, vol. 1098 de *Lecture Notes in Computer Science*, p. 275–294. Springer-Verlag, Juillet 1996.

[LAM 94] LAMPORT L., « The Temporal Logic of Actions ». *ACM Transactions on Programming Languages and Systems*, p. 872–923, Mai 1994.

[LIT 94] LITTLE M. et SHRIVASTAVA S., « Object Replication in Arjuna ». Rapport technique 50, Broadcast Project Technical Report, Octobre 1994. Department of Computer Science, University of Newcastle.

[MAE 87] MAES P., « Concepts and Experiments in Computational Reflection ». In *Proceedings of OOPSLA'87*, vol. 22 de *SIGPLAN Notices*, p. 147–155. ACM Press, Décembre 1987.

[MAK 94] MAKPANGOU M., GOURHANT Y., NARZUL J. L. et SHAPIRO M., « Fragmented Objects for Distributed Abstractions ». In *Readings in Distributed Computing Systems*, p. 170–186. IEEE Computer Society Press, 1994.

[MAT 91] MATSUOKA S., WATANABE T. et YONEZAWA A., « Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming ». In *Proceedings of ECOOP'91*, vol. 512 de *Lecture Notes in Computer Science*, p. 231–247. Springer-Verlag, Juillet 1991.

[MCQ 80] MCQUILLAN J., « The New Routing Algorithm for the Arpanet ». *IEEE Transactions on Communication*, Mai 1980.

[MEY 92] MEYER B., « Applying Desing by Contract ». *IEEE Computer*, vol. 25, n° 10, p. 40–52, Octobre 1992.

[MEY 95] MEYER J. et VAN DER HOEK W., *Epistemic Logic for AI and Computer Science*. Cambridge University Press, 1995.

[MIG 99] MIGEON F.. « Etude et implantation de mécanismes réflexifs dans un langage concurrent ». Thèse de Doctorat de l'Université Paul Sabatier, Toulouse, Juillet 1999.

[MIS 88] MISRA J. et CHANDY K., *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[MIS 89] MISHRA S., PETERSON L. et SCHLICHTING R., « Implementing Fault-Tolerant Replicated Objects using Psync ». In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 1989.

[MOW 97] MOWBRAY T. et MALVEAU R., *CORBA Design Patterns*. Wiley, 1997.

[MOY 91] MOY J., « *RFC 1247, OSPF version 2* ». IETF, Juillet 1991.

[OCL97] OMG & UML, « *Object Constraint Language Specification* », Septembre 1997. Document ad/97-08-08.

[OMG 98] OMG. « Common Object Request Broker : Architecture and Specifications ». OMG Document-98/02, Object Management Group, 1998.

[PLA 95] PLACIDE P., DUCHIEN L., FLORIN G. et SEINTURIER L., « A Consistent Global State Algorithm to Debug Distributed Object-Oriented Applications ». In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, Mai 1995.

[REN 96] VAN RENESSE R., BIRMAN K. et MAFFEIS S., « Horus: A Flexible Group Communication System ». *Communications of the ACM*, vol. 39, Avril 1996.

[RUM 91] RUMBAUGH J., BLAHA M., PREMERLANI W. et AL., *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[SCH 93] SCHNEIDER F., *Replication Management using State-Machine Approach*. Distributed Systems, ACM PressBook, 1993.

[SEI 97] SEINTURIER L.. « Conception d'algorithmes répartis et de protocoles réseaux en approche objet ». Thèse de Doctorat du Conservatoire National des Arts et Métiers, Paris, Décembre 1997. *http://cedric.cnam.fr/~seintur/these*.

[SEI 98] SEINTURIER L., DUCHIEN L. et FLORIN G., « CAOLAC : un protocole à méta-objets pour la synchronisation d'objets concurrents ». *L'Objet*, vol. 4, n° 3, p. 241–272, Septembre 1998.

[STE 99] VAN STEEN M., HOMBURG P. et TANENBAUM A., « Globe: A Wide-Area Distributed System ». *IEEE Concurrency*, p. 70–78, Janvier 1999.

[SUN 97] SUN, « *Java Remote Method Invocation* ». Sun Microsystems, 1997.

[SUN 99] SUN, « *Entreprise Java Beans* ». Sun Microsystems, 1999.

[TEL 94] TEL G., *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[WOO 93] WOOD M., « Replicated RPC using Amoeba Closed Group Communication ». In *IEEE International Conference on Distributed Systems*, 1993.

photo

Laurence Duchien *est maître de conférences au Conservatoire National des Arts et Métiers à Paris. Elle a obtenu son doctorat en informatique à l'Université Pierre et Marie Curie en 1988, et son habilitation à diriger des recherches à l'Université Joseph Fourier en 1999. Elle a rejoint l'équipe "Objets, Temps et Ordres dans les Systèmes Répartis" du Laboratoire CEDRIC en 1991. Ses activités de recherche concernent, entre autres, les algorithmes répartis pour les applications coopératives et les modèles de spécification et de preuves d'applications réparties orientées objet.*

photo

Gérard Florin *est professeur des universités au Conservatoire National des Arts et Métiers à Paris. Il a obtenu son doctorat de spécialité en informatique et son doctorat d'état en informatique à l'Université Pierre et Marie Curie, respectivement, en 1975 et en 1985. Il a dirigé le Laboratoire CEDRIC de 1988 à 1998. Ses activités de recherche incluent les réseaux de Petri stochastiques et concernent actuellement, l'informatique répartie tolérant les pannes.*

photo

Lionel Seinturier *est maître de conférences à l'université Pierre et Marie Curie. Ses activités de recherche sont réalisées au sein du thème "Systèmes Répartis Coopératifs" du laboratoire LIP6. Il a obtenu son doctorat en informatique au Conservatoire National des Arts et Métiers à Paris en 1997. Il travaille sur la conception d'applications réparties en approche objet. Ses principaux thèmes de recherche concernent la gestion de la synchronisation, la réflexivité et la programmation par aspects dans les applications réparties.*