UNE APPROCHE ORIENTEE MODELE POUR LA GESTION DES ARCHITECTURES LOGICIELLES DISTRIBUEES DYNAMIQUES

Karim Guennoun,
Doctorant en Systèmes informatiques
guennoun@laas.fr , + 33 5 61 33 78 15

Khalil Drira,
Chargé de recherche CNRS
khalil@laas.fr + 33 5 61 33 78 86

Michel Diaz,
Directeur de recherche CNRS
diaz@laas.fr + 33 5 61 33 62 56

Adresse professionnelle
LAAS-CNRS, 7 Avenue du colonel Roche
31077 Toulouse Cedex 4, France

Résumé : Cet article présente les graphes abstraits de composants (Abstract Component Graph **ACG**), une approche visant à coordonner l'évolution dans les architectures logicielles dynamiques distribuées. Elle permet, en plus de spécifier l'évolution dynamique à l'échelle de la configuration de l'architecture, la spécification de l'évolution dynamique à l'échelle du composant. Elle peut être employée pour simuler les différentes étapes d'instanciation du composant, le changement de comportement pendant l'exécution, la migration, et d'autres caractéristiques spécifiques aux architectures logicielles des systèmes distribués. Cette approche permet également la vérification de propriétés à ces deux niveaux des spécifications (i.e. configuration de l'architecture et structure du composant).

Summary : This article presents the Abstract Component Graph (**ACG**), an approach which addresses the coordination issue for distributed dynamic software architectures. It allows, in addition of the specification of architecture dynamic changes on the configuration level, the specification of dynamic changes at the level of the component. The ACG approach can also be used to simulate the various steps of component instantiation, behaviour changes on the run-time, migration, and other characteristics specific to distributed software architectures. This approach allows checking properties at these two levels of specification (i.e. architecture configuration and component structure).

Mots clés : Architectures logicielles, Dyanmicité, Coordination, Graphes, Edition partagée.

Une approche orientée modèle pour la gestion des architectures logicielles distribuées dynamiques

1 - INTRODUCTION

La distribution et l'évolution dynamique des composants et des schémas d'interaction sont deux exigences importantes qui rendent les modèles traditionnels de l'architecture logicielle inadéquats pour la conception des nouveaux systèmes logiciels tels que ceux dédiés à la collaboration distribuée d'activités de groupe (Adler (1995)). De telles applications font de la spécification de l'évolution dynamique un domaine de recherche important traité par différents travaux tels que ceux réalisés par Allen, Douence, et Garlan (1998), et par Riveil et Senart (2002).

La programmation orientée composant, d'après Wegner (1993), est susceptible d'être le modèle le plus prometteur pour les systèmes logiciels de prochaine génération, où les techniques doivent traiter la dynamicité pour décrire l'activation et la désactivation de composants. Cette technologie constitue une base appropriée pour développer des solutions qui traitent la dynamicité dans la conception des applications coopératives (Guerrero et Fuller (2001)). Pour de telles applications, l'adaptabilité aux changements spécifiques à la coopération peut exiger l'activation et l'intercommunication dynamiques de différents composants logiciels, à différents moments d'exécution. Ce problème est traité dans cet article comme une fonction de coordination se concentrant sur la gestion de l'évolution dynamique des architectures logicielles. Cette fonction de coordination est basée sur une description utilisant un graphe et un ensemble de règles de transformation qui seront appliquées selon un protocole bien défini.

Dans les divers travaux passés utilisant les graphes pour décrire les architectures dynamiques tel que ceux de Dorfel et Hofmann (1998), les graphes sont employés pour décrire des modèles d'architectures matérielles utilisés pour générer des implémentations VHDL/C. La dynamicité est considérée seulement au niveau des connexions définissant les flux de communication entre les composants.

Contrairement à notre approche, il n'est pas possible de créer des composants dynamiquement. La technologie orientée composant des architectures logicielles a l'avantage de permettre ce niveau de dynamicité. Notre approche s'appuie sur cette technologie et permet d'augmenter l'adaptabilité des architectures des applications distribuées.

Des travaux antérieurs tels que ceux de Wermelinger et Fiadeiro (2002), et ceux de Ermel, Bardhol, et Padberg (2001) proposent également d'utiliser les graphes et les règles de transformation pour modéliser les problèmes de coordination. Cette approche est mise en application par Holzbacher, Perin, et Sudholt (1997) pour les systèmes de commande de chemins de fer, et par Le Metayer (1998) pour la spécification de styles d'architectures. Fahmy et Holt (2000) proposent d'employer la transformation de graphe pour traiter l'architecture logicielle décrite par un graphe où les nœuds représentent des composants et où les arcs représentent les relations d'interdépendance entre ces composants. Contrairement à notre approche, seulement deux transformations prédéfinies sont considérées. Elles laissent représenter différents niveaux de détails (ou d'abstractions) dans le graphe d'architecture en remplaçant un noeud par ses composants de constitution et l'opération inverse. Le travail de Li et Horgan (2000) traite également la transformation d'architecture en utilisant des méthodes formelles basées sur le formalisme des machines à états finies étendues et le langage SDL. Il est consacré à la simulation de modèles dans des environnements *workflow*.

Jager (2000) adopte la réécriture de graphes pour produire des outils de gestion pour l'exécution des modèles de processus en utilisant un formalisme spécifique appelé les réseaux dynamiques de tâches (*dynamic task nets*). Alancar et Lucena (1995) proposent une approche formelle pour gérer l'évolution des architectures logicielles. Leur travail s'applique à la gestion de configuration impliquant le choix et l'intégration de différentes versions de modules logiciels.

Dans la section 2, nous présentons l'approche **ACG** du point de vue description. Dans la section 3, nous présentons la technique fondamentale de transformation du point de vue du fonctionnement du système. Nous illustrons cette approche par l'exemple du modèle d'architecture pour l'édition partagée. Des perspectives de notre travail sont données en conclusion.

2 - LES GRAPHES D'ARCHITECTURE ET LES REGLES DE TRANSFORMATION

2.1 - La structure ACG

Les graphes abstraits de composants (ACG) est une structure marquée et générique que nous employons pour définir les graphes d'architecture (ACG

totalement instantié), et les graphes de règles (ACG partiellement instantié). La première structure décrit une architecture orientée composant comme un ensemble de composants associés aux nœuds du graphe et un ensemble d'arcs dénotant les relations d'interdépendance entre ces composants.

ACG: *P*(Nodes) * *P*(Edges)

Dans une structure ACG, les nœuds décrivent des composants logiciels et sont ainsi marqués par les champs suivants : la classe, le comportement, l'état du composant (actif ou inactif), les facettes de son comportement, sa localisation (la machine sur laquelle ils sont exécutés), et une liste supplémentaire de paramètres concernant le niveau applicatif. Nous distinguons deux catégories de nœuds : les nœuds de règles et les nœuds de graphes. La différence entre ces deux types de nœuds est que le premier est une abstraction (au niveau structurel ou au niveau fonctionnel) d'un type de composants et peut avoir ainsi des champs variables[1], alors que le second correspond à un composant instantié de l'architecture et ne peut donc avoir que des champs totalement instantiés.

Node: Class * State * Facet * Location * Parameters

Les arcs sont orientés et sont définis par les deux nœuds qu'ils relient. Ils constituent le deuxième dispositif élémentaire de la description d'architecture et peuvent modéliser un large panel de relations (e.g. relations de contrôle, de composition, de niveau applicatif).

Edge : Node * Node

2.2 - Structure des règles de transformation

Nous définissons une *règle de transformation* comme une partition d'un *graphe de règle* permettant de décrire les contraintes qui conditionnent l'évolution de l'architecture et les changements qui se produisent quand une règle est applicable. Au niveau supérieur de l'abstraction, une règle de transformation peut être vue comme un triplet $RT \equiv <Partition, constraints, Substitutions>$, où :

- ***Partition*** : est une décomposition du graphe de la règle de transformation en quatre zones :
  - La zone *Inv* : Un fragment du graphe de la règle qui devrait être identifié (par homomorphisme) dans le graphe d'architecture. Ce fragment du graphe restera inchangé après l'application de la règle.
  - La zone *Del* : Un fragment du graphe de la règle qui doit être identifié (par homomorphisme) dans le graphe de l'architecture. Le fragment du graphe qui lui a été associé par l'homomorphisme est supprimé après l'application de la règle.
  - La zone *Abs* : Un fragment du graphe de la règle qui ne doit pas être identifiable (par homomorphisme) dans le graphe de l'architecture pour que la règle de transformation soit applicable.
  - La zone *Add* : Le fragment du graphe de la règle qui sera ajouté après l'application de la règle.
- ***Constraints*** : Décrit les contraintes sur les champs des noeuds. La règle n'est applicable que si toutes ces contraintes sont satisfaites par les nœuds du graphe de l'architecture qui sont unifiés avec les nœuds de la règle. Une contrainte est un couple dont le premier champ est la fonction d'évaluation de la contrainte δ (une fonction prenant en paramètre un ensemble de nœuds et renvoyant un booléen), et le deuxième est l'ensemble des nœuds qui sera évalué par **δ**.

Const : *P*(δ (: *P*(Nodes) → boolean) * *P*(Nodes)))

- ***Substitutions***: Modélise les différentes substitutions que devrait subir les champs de certains nœuds du graphe après l'application de la règle. Les substitutions sont modélisées par la procédure de substitution σ qui prend en paramètres un ensemble de noeuds et permet de substituer à certains de leurs champs des nouvelles valeurs. Ceci permet de spécifier, outre des architectures évoluant au niveau de la configuration et impliquant les quatre opérations fondamentales (i.e. l'addition et la destruction de composants ou de connections), des évolutions dynamiques à l'échelle d'un composant (e.g. migration, changement de comportement) via des changements au niveau des attributs. Ainsi, par exemple, pour un composant d'édition partagée représentant une fenêtre d'édition, en liant son comportement à la valeur d'un attribut, on peut décrire l'évolution dynamique de ce comportement et ainsi pouvoir, par exemple, spécifier son passage du mode lecture vers un mode lecture/écriture et vice-versa. Cette spécification sera réalisée en décrivant quand est-ce que l'évolution dynamique est consistante avec, par exemple, la vérification de l'exclusion mutuelle sur la fenêtre d'édition, et comment cette évolution serait traduite au niveau implémentation avec, par exemple, le positionnement correcte du droit d'écriture et l'activation des boutons d'édition sur l'interface graphique.

Sub : *P*(σ (:*P*(Nodes) → void) * *P*(Nodes))

Ainsi avec les définitions précédentes, une règle possède la structure suivante :

Rule: $\underbrace{Inv * Del * Add * Abs}$ * Const * Sub

2.3 - Le protocole de coordination

Le protocole de coordination est en charge de gérer l'évolution dynamique de l'architecture du système. Ce protocole manipule, le graphe courant, les règles de coordination, et les événements à traiter, et

[1] Dans notre notation, les variables seront préfixées par le symbole "_". Par exemple, la notation <*E, _x,F,Ad1*> dénote un noeud de la classe *E* avec une facette appartenant à *F*, situé dans le site *Ad1*. La variable *_x* indique que le noeud est peut-être dans un état actif ou inactif.

associe à chaque type d'événements les règles de transformation correspondantes. Il associe, aussi, pour chaque type d'événements et pour chacune des règles lui correspondant, la procédure de transformation qui doit être appliquée à chaque règle (au niveau de son graphe, son champ *constraints*, et son champ *substitutions*) avant l'unification avec le graphe. Le protocole de coordination peut introduire aussi des événements spéciaux traduisant, par exemple, des vérifications de propriétés telles que des propriétés de *sûreté* ou de *complétude*. Ces propriétés sont décrites sous la forme d'une ou de plusieurs règles de coordination[2].

Protocol :

Graph * *P*(Rule) * *P*(EventType * Trans * *P*(Rule))

L'événement décrit l'action de déclenchement menant à l'application d'un ensemble de règles de transformation sur le graphe courant de l'architecture. Il peut être produit par le système lui-même ou par son environnement, et est décrit comme un couple contenant le type de l'événement, et les paramètres qu'il transporte.

Event: EventType * EventParameters

Le champ *Trans* permet de répercuter les paramètres des événements sur les règles de transformation. Les règles sont ainsi partiellement instantiées en affectant des valeurs à certaines de leurs variables.

Trans: P(α (: *P*(Nodes) * Event → void) * *P*(Nodes))

3 - APPLICATION DES REGLES DE TRANSFORMATION

Nous définissons la fonction d'unification comme une fonction récursive qui établit un homomorphisme entre la partition du graphe de la règle de transformation et le graphe de l'architecture (en tenant compte du champ *constraints* de la règle). Elle est basée sur les quatre définitions suivantes qui décrivent l'unification d'un ensemble de nœuds du graphe de règle avec un ensemble de nœuds du graphe de l'architecture.

Définition 1 [*Unification de champs de nœuds*]

Unifiable(champ$_i$, champ$_j$) ≡

$$\begin{cases} \exists \_X \in Variables / champ_i = \_X, OU \\ champ_i = champ_j \end{cases}$$

Définition 2 [*unification de deux nœuds*]

Soient N_1=(N_1.champ$_1$,…,N_1.champ$_n$) un nœud d'un graphe de règle, et N_2=(N_2.champ$_1$,…,N_2.champ$_m$) un nœud d'un graphe d'architecture. Alors,

Uni*fiable(N_1,N_2)* ≡

$$\begin{cases} (n = m), Et \\ \forall i \in [1,..,n], Unifiable(N_1.champ_i, N_2.champ_i), Et \\ Unifiable(N_1.successeurs, N_2.successeurs), Et \\ Const(r), Et \\ \text{Pas d'inconsistance dans les unifications} \end{cases}$$

Définition 3 [*Unification de deux ensembles ordonnés de nœuds*]

Soit EO_1 un ensemble ordonné de nœuds d'un graphe d'une règle tel que $EO_1=[N_{1,1},...,N_{1,n}]$. Soit EO_2 un ensemble ordonné de nœuds d'un graphe d'architecture tel que $EO_2=[N_{2,1},...,N_{2,m}]$. Alors,

S_Unifiable(EO_1,EO_2) ≡

$$\begin{cases} (n = m), Et \\ \forall i \in [1,..,n], Unifiable(N_{1,i}, N_{2,i}), Et \\ Const(r), Et \\ \text{Pas d'inconsistance dans les unifications} \end{cases}$$

Définition 4 [*Unification de deux ensembles de nœuds*]

Soient *NR* un ensemble de nœuds d'un graphe de règle tel que $NR=\{N_{1,1},...,N_{1,n}\}$ et *NG* un ensemble de nœuds de graphe d'architecture tel que $NG=\{N_{2,1},\ldots,N_{2,m}\}$. Alors,

Unifiable(NR,NG) ≡

$$\begin{cases} (n \le m), Et \\ \exists \{N_{2,i1},..., N_{2,in}\} \subset \{N_{2,1},...N_{2,n}\}, \\ S\_Unifiable\ ([N_{1,1},..., N_{1,n}], [N_{2,i1},..., N_{2,in}]) \end{cases}$$

Exemple

Soit le graphe de règle *G(r)* et le graphe **G** décrits dans Fig1, avec :

Fig. 1 – *G(r) le graphe de la règle r, et G le graphe de l'architecture courante.*

On a alors,

1- **S_Unifiable**([n1,n2,n3],[n4,n5,n6])?

⇔

Unifiable(n1,n4) ***Et*** **Unifiable**(n1.Succ,n4.Succ) ***Et*** **Unifiable**(n2,n5) ***Et*** **Unifiable**(n2.Succ,n5.Succ) ***Et*** **Unifiable**(n3,n6) ***Et*** **Unifiable**(n3.Succ,n6.Succ)

⇒

(_d=nom$_2$) ***Et*** (10=10) ***Et*** (_d=nom$_1$) ***Et*** …

⇒

Inconsistance car *_d* est unifié avec deux valeurs différentes.

⇒ **Faux**.

[2] Un exemple est donné dans la section 4.2.2

2- **S_Unifiable**([n1,n2,n3],[n7,n6,n8])?
⇔
(_d=nom$_1$) ***Et*** (10=10) ***Et*** (_d=nom$_1$) ***Et*** (14=14) ***Et*** (nom$_1$=nom$_1$) ***Et*** (_f=17) ***Et*** (_f>15) ***Et*** **Unifiable**({n3},{n8}) ***Et*** **Unifiable**({n3},{n8}).
⇔
(_d=nom$_1$) ***Et*** (10=10) ***Et*** (_d=nom$_1$) ***Et*** (14=14) ***Et*** (nom$_1$=nom$_1$) ***Et*** (_f=17) ***Et*** (_f>15) ***Et*** (nom$_1$=nom$_1$) ***Et*** (_f=17) ***Et*** (_f>15) ***Et*** **Unifiable**({ },{ }) ***Et*** (_d=nom$_1$)
⇔ **Vrai** avec comme résultat les unifications: *_d=nom$_1$ et _f=17.*

Définition 5 [*Unification d'une règle avec un graphe*]
Soient *r* une règle et *g* un graphe, soit *précondition(r)* l'ensemble des nœuds et des arcs appartenant à *(Inv(r)∪Del(r))* et soit *restriction(r)* l'ensemble des nœuds et des arcs appartenant à *(Inv(r)∪Del(r)∪Abs(r))*, alors,
Unifiable*(r,g)* ≡

$$\begin{cases} \exists s_g = \left(\{n_0,...,n_i\},\{e_0,...,e_j\}\right) \in g, Telque \\ S\_Unifiable\left(precondition(r), s_g\right), Et \\ Const(r), Et \\ \left(\forall s'_g = \left(\{n_0,...,n_i,...,n_{i+k}\},\{e_0,...,e_j,...,e_{j+l}\}\right) \subset g\right) \\ \Rightarrow \neg\left(S\_Unifiable(restriction(r), s'_g)\right) \end{cases}$$

L'unification d'une règle *r* avec un graphe *g* suivra la démarche suivante:
1. Si *r* n'est pas unifiable avec *g*, alors *g* reste inchangé.
2. Sinon :
   2.1. On rajoute dans le graphe *g* des copies des nœuds et des arcs contenus dans le champ *Add(r)*.
   2.2. On détruit les nœuds et les arcs du graphe qui ont été unifiés avec des nœuds et des arcs du champ *Del(r)*.
   2.3. On modifie les champs des nœuds de *g* qui ont été unifiés avec les nœuds figurant dans le champs *Sub(r)* en appliquant les procédures de substitution de la règle.

4 - UN CAS D'ETUDE, L'EDITEUR PARTAGE

Nous considérons ici, la gestion de l'architecture d'une application d'édition partagée. Il s'agit d'une application coopérative et distribuée permettant à plusieurs utilisateurs de travailler sur un même document. Ce document est organisé en composantes (pouvant représenter des caractères, des lignes, des paragraphes, des pages etc.) numérotées de *0* à *n*. Un utilisateur se doit, avant d'écrire dans une zone (une zone est un ensemble de composantes consécutives), de la réserver. Une fois qu'un utilisateur a fini sa rédaction, il peut libérer la zone qu'il a auparavant réservée. Chaque composante du document ne peut se trouver que dans l'un des deux cas de figures suivants : ou bien elle est libre, ou bien elle est réservée par un utilisateur unique. Pour éviter une surcharge inutile dans la description du cas d'étude, nous allons nous limiter à décrire les attributs des noeuds qui sont vraiment significatifs pour la coordination de l'application d'édition partagée.

4.1- Modélisation de l'architecture du cas d'étude

L'architecture de l'application d'édition partagée est modélisée par un graphe linéaire représentant les zones consécutives du document (e.g. un nœud *n1* est fils d'un nœud *n2 si et seulement si* la zone représentée par *n1* est consécutive à celle représentée par *n2*). Les noeuds du graphe modélisant ces zones devront donc décrire les paramètres de début et de fin de zone, l'état de la zone (*F* pour l'état libre et *B* pour l'état réservé), le propriétaire de la zone[3], et la localisation de la zone[4] (e.g. l'adresse IP de la machine où se trouve le propriétaire). Les nœuds du graphe auront donc pour champs cinq paramètres (un exemple est donné dans Fig. 2).

document où les composantes, de 0 à 10 sont réservées par Jacques, les composantes de 11 à 20 sont libres, de 21 à 35 sont réservées par Bernadette, et de 26 à 50 sont réservées par Claude.

4.2- Les règles de coordination

4.2.1- Les règles de réservation et de libération des composantes
On a classé les règles de transformation de l'application d'édition partagée en trois catégories selon les événements qu'elles traitent. La première catégorie concerne la réservation, la deuxième concerne la libération, et la troisième la vérification des deux propriétés de sûreté et de complétude. Par souci de simplification, et pour éviter le cas particulier d'un graphe avec un noeud unique[5], nous rajoutons deux noeuds (auxquels nous affectons des champs garantissant qu'ils ne seront jamais détruits, et qu'ils ne lèveront jamais une violation des règles de sûreté ou de complétude) l'un en tête et l'autre en queue du graphe à structure linéaire.

[3] Dans le cas d'une zone libre ce paramètre sera positionné à *nobody*

[4] Ce paramètre sera positionné à *nowhere* dans le cas d'une zone libre

[5] Le traitement de ce cas particulier doublerait inutilement le nombre de règles de réservation et de libération

Considérons un utilisateur "*owner*" sur la machine "*host*" et voulant réserver la zone [*begin,end*]. Pour que cela puisse être possible, il doit exister dans le document une zone libre [$begin_f$,end_f] telle que [*begin,end*] $\subseteq$ [$begin_f$,end_f]. Quatre cas de figure sont possibles:

1- Cas1 : ($begin_f$ = *begin*) ***Et*** (end_f = *end*),

2- Cas 2 : ($begin_f$ = *begin*) ***Et*** (end_f > *end*),

3- Cas 3 : ($begin_f$ < *begin*) ***Et*** (end_f = *end*),

4- Cas 4 : ($begin_f$ < *begin*) ***Et*** (end_f > *end*).

Considérons maintenant un utilisateur "*owner*" sur la machine "*host*" et possédant la zone *z*, lors de la libération de cette zone, quatre cas de figures sont possibles:

1- Cas 5 : La zone précédant *z* et celle qui la suit sont libres,

2- Cas 6 : La zone précédant *z* et celle qui la suit sont réservées,

3- Cas 7 : La zone précédant *z* est libre, et celle qui la suit est réservée,

4- Cas 8 : La zone précédant *z* est réservée et celle qui la suit est libre.

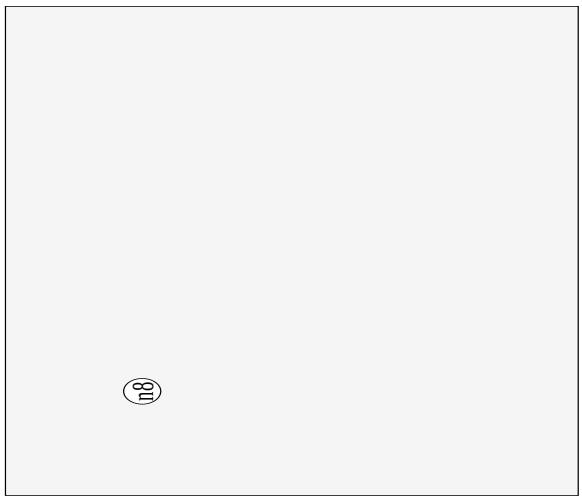

Fig. 3- *Les graphes des règles de réservation et de libération*

On obtient donc les quatre règles de réservation (*r1,r2,r3,r4*) et les quatre règles de libération (*r5,r6,r7,r8*) présentées dans Fig. 3. La réservation (respectivement la libération) d'une zone pour un utilisateur est donc possible *si et seulement si* l'une des quatre règles *r1, r2, r3,* ou *r4* (respectivement *r5, r6, r7,* ou *r8*) est applicable au graphe courant. Il est, aussi, à noter que la fonction d'unification telle qu'elle a été définie nous assure qu'un utilisateur ne peut libérer une zone que s'il en est le propriétaire.

Boolean **supAtt1**(*P(Nodes)* {N_1;N_2})
{ *return* (N_1.champ(1) > N_2.champ(1)); }

Boolean **supAtt0** (*P(Nodes)* {N_1;N_2})
{ *return*(N_1.champ(0) > N_2.champ(0)); }

Node : n1 = {_x; _y;B;_owner1;_host1},
n2 = {_begin;_end;F;nobody;nowhere},
n3 = {_begin;_end';F;nobody;nowhere},
n4 = {_begin';_end;F;nobody;nowhere},
n5 = {_begin';_end';F;nobody;nowhere},
n6 = {_a;_b;B;_owner2;_host2},
n7 = {_begin';_begin-1;F;nobody;nowhere},
n8 = {_begin;_end;B;_owner;_host},
n9 = {_end+1;end';F;nobody;nowhere};

Graph : Inv = ({n1;n6},{ }),
Del(r1) = ({n2},{(n1,n2);(n2,n6)}),
Del(r2) = ({n3},{(n1,n3);(n3,n6)}),
Del(r3) = ({n4},{(n1,n4);(n4,n6)}),
Del(r4) = ({n5},{(n1,n5);(n5,n6)}),
Add(r1) = ({n8},{(n1,n8);(n8,n6)}),
Add(r2)=({n8;n9},{(n1,n8);(n8,n9);(n9,n6)},
Add(r3)=({n7,n8},{(n1,n7);(n7,n8);
(n8,n6)}),
Add(r4)=({n7;n8,n9},{(n1,n7);(n7,n8);
(n8,n9);(n9,n6)});
'onst : C(r2)={(supAtt1,{n9;n8})},
C(r3)={(supAtt0,{n8;n7})},
C(r4)={(supAtt1,{n9;n8});
(supAtt0,{n8;n7})};
'ule : r1=(Inv,Del(r1),Add(r1),{ },C(r1)),
r2 = (Inv,Del(r2),Add(r2),{ },C(r2)),
r3=(Inv,Del(r3),Add(r3),{ },C(r3)),
r4 = (Inv,Del(r4),Add(r4),{ },C(r4)),
r5 = (Inv,Add(r1),Del(r1),{ }, { }),
r6 = (Inv ,Add(r2),Del(r2),{ }, { }),
r7 = (Inv ,Add(r3),Del(r3),{ }, { }),
r8 = (Inv ,Add(r4),Del(r4),{ }, { }),

.2.2 - Vérification des propriétés de sûreté et de omplétude.

es deux propriétés de sûreté[6] et de complétude[7] eront vérifiées par l'unification des deux règles les représentant (Fig. 4) déclarées comme suite :

Boolean **Intersect** (*P(Nodes)* {n1;n2})
{*return* ((n1.champ(0) ≥ n2.champ(0)) ***AND***
(n1.champ(0) ≤ n2.champ(1)))};}

Node: n10 = {_x; _y;F;nobody;nowhere},
n11={_a;_b;F;nobody;nowhere},
n12={_x;_y;_state1;_owner1;_host1},
n13={_a;_b;_state2;_owner2;_host2};
Graph: Inv(r9)=({n10;n11},{(n10,n11)}),
Inv(r10)=({n12;n13},{ }),

[6] La propriété de sûreté est violée ssi il existe deux zones consécutives libres

[7] La propriété de complétude est préservée ssi pour chaque paire de zones, leur intersection est vide.

```
        Empty=({ },{ });
Constr: C(r10)={(Intersect,{n12;n13})};
Rule:   r9=(Inv(r9),{ },{ },{ },{ }),
        r10=(Inv(r10),{ },{ },{ },C(r10));
```

```
        node1 = (0, N - 1, F, nobody, nowhere);
Graph : G = ({InitialNode; node1; FinalNode},
        {(InitialNode,node1);(node1,FinalNode)});
Protocol: EPCoo = (G,
                ("completness", {r10}, { })),
```

Fig.
et d

Une unification possible de l'une de ces deux règles indiquerait que la propriété correspondante est violée dans l'architecture courante.

3.3- Les événements traités par l'application

L'application d'édition Partagée doit traiter quatre types d'événements. Le premier concerne une demande d'un utilisateur de réserver une zone du document. Un événement de ce type doit donc transporter le début et la fin de la zone à réserver, le nom ou l'alias du demandeur, et la machine sur laquelle il se trouve. Le deuxième concerne une demande de libération. Un événement de ce type transportera exactement le même nombre et types de paramètres qu'un événement de réservation. Le troisième concerne une demande de vérification de la propriété de sûreté sur l'architecture courante, ce type d'événement ne transportera aucun paramètre et le quatrième concernera la propriété de complétude et ne transportera également aucun paramètre. On obtient donc la description suivante :

```
Event :
Book = ("book",{begin(:Int);end(:Int);
                owner(:String); host(:Int)}),
Free = ("free",{begin(:Int);end(:Int);
                owner(:String);host(:Int)}),
VerifySafety = ("safety",{ }),
VerifyCompletness = ("completness",{ });
```

3.4- Protocole de coordination

L'événement *Book* doit répercuter ses paramètres sur les nœuds des règles *r1,…,r4* et donc les nœuds *n1,…,n9*, l'événement *Free* sur les règles *r5,…,r8* et donc les nœuds *n1,…,n9*, l'événement *VerifySafety* sur la règle *r9* et donc les nœuds *n10* et *n11*, et l'événement *VerifyCompletness* sur la règle *r10* et donc les nœuds *n12* et *n13*. Au vu de ce qui a été développé précédemment, le protocole de coordination pour l'architecture de l'application d'édition partagée peut être décrit sous la forme suivante :

```
Node :  InitialNode = (T, T, B, nobody, nowhere),
        FinalNode = (⊥, ⊥ , B, nobody, nowhere),
```

Où,

```
Void  T ( Event e , P(Nodes) nodes)
    {for ( n in nodes)}
        {if (n.champ(0) ==_begin)
            {n.champ(0)=e.parameters(0);}
         if (n.champ(0) ==_end+1)
            {n.champ(0)=e.parameters(1)+1;}
         if (n.champ(1) ==_end)
            {n.champ(1)=e.parameters(1);}
         if (n.champ(1) ==_begin-1)
            {n.champ(1)=e.parameters(0)-1;}
         if (n.champ(3) ==_owner)
            {n.champ(3)=e.parameters(2);}
         if (n.champ(4) ==_host)
            {n.champ(4)=e.parameters(3);}}}
```

CONCLUSION

Nous avons présenté une approche orientée modèle pour la synthèse des protocoles de coordination pour les applications logicielles distribuées orientées composant. En nous basant sur des descriptions sous formes de graphes et de règles de transformation, nous avons pu introduire une approche pour décrire et gérer l'évolution dynamique d'une architecture logicielle de façon consistante. En comparaison avec celle présentée par Le Metayer (1998), notre approche permet une description plus fine et plus large puisqu'elle nous permet de modéliser des contraintes plus complexes pour l'évolution dynamique de l'architecture. Elle introduit des contraintes structurelles additionnelles modélisées par les deux champs *Inv* et *Abs*, et des contraintes fonctionnelles sur les champs de nœuds, permettant le changement de comportements et de propriétés du composant par l'intermédiaire du champ *Sub*stitutions. La vérification des propriétés via une modélisation par des règles de transformation est également possible.
Dans le cas particulier des logiciels de support des activités coopératives distribuées, cette coordination permet d'éviter les conflits entre les participants lors des accès simultanés aux mêmes objets de l'espace de travail partagé. Nous avons illustré et éprouvé notre approche pour les activités

d'édition en groupe avec des documents partagés à structure linéaire dynamique. La démarche est généralisable pour les structures arborescentes statiques et dynamiques. L'approche est en cours d'expérimentation avec les technologies EJB et CCM.
Nous travaillons actuellement sur la description des architectures dynamiques sous forme de styles d'architectures modélisés par les grammaires de graphes. Cette approche nous permettra de vérifier la conformité du protocole de coordination avec le style d'architecture spécifié.

BIBLIOGRAPHIE

Allen, R., Douence, R., Garlan, D. (1998), "Specifying and analyzing dynamic software architectures", in 1998 conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal.

Adler, R.M. (1995), "Distributed coordination models for client/server computing", in IEEE Computer, vol. 28, n° 4, pp14-22.

Alencar, P.S.C., Lucena, C.J.P. (1995), "A formal description of evolving software systems architectures", in Science of Computer Programming, vol. 24, n° 1, pp 41-61.

Dorfel, M., Hofmann R. (1998), "A prototyping system for high performance communication systems", in RSP'98, Leuven, Belgium.

Ermel, C., Bardhol, R., Padberg, J. (2001), "Visual design of software architecture and evolution based on graph transformation", in Uniform Approaches to graphical process specification techniques, Genove, Italy.

Fahmy, H., Holt, R.C. (2000), "Software architecture transformations", in ICSM, pp 88-96.

Guerrero, L.A., Fuller, D.A. (2001), "A pattern system for the development of collaborative applications", in Information and Software technology, vol. 43, n° 7, pp 457-467.

Holzbacher, A., Perin, M., Suldholt, M. (1997), "Modelling railway control systems using graph grammars: a case study", in the second international conference on Coordination Languages and Models, Berlin, Germany, pp 172-186.

Jager, D. (2000), "Generating tools from graph-based specifications", in Information and Software Technology, vol. 42, n° 2, pp 129-139.

Li, J.J., Horgan, J.R. (2000), "Applying formal description techniques to software architectural design", in Computer Communications, vol. 23, n° 12, pp 1169-1178.

Le Metayer, D. (1998), " Describing software architecture styles using graph grammars", in IEEE Transactions On Software Engineering, vol. 24, n° 7, pp 521-533.

Riveil, M., Senart, A. (2002), "Aspects dynamiques des langages de description d'architecture logicielle", in L'objet: Coopération dans les systèmes a objets", vol. 8, n° 3, pp 109-129.

Wegner, P. (1993), "Tradeoffs between reasoning and modelling", MIT Press.

Wermelinger, M., Fiadeiro, J.L. (2002), "A graph transformation approach to software architecture reconfiguration", in Science Of Computer Programming, vol. 44, pp 133-155.