Université de Versailles Saint-Quentin-en-Yvelines

Institut National des Télécommunications

Rapport de Stage
DEA Méthodes Informatiques des Systèmes Industriels
(M.I.S.I)

Applications multi-composants et déploiement sur terminaux mobiles

Abdelkrim BELOUED

Responsable de DEA : Ahmed MEHAOUA

Responsable de stage : Chantal TACONET

Septembre 2004

Ce stage de DEA a été réalisé au sein du laboratoire **Systèmes Répartis** du département **Informatique** de l'**Institut National des Télécommunications**

Remerciements

Je tiens à remercier, M. Guy Bernard pour m’avoir accueilli dans l’équipe Systèmes Répartis de l’Institut National des Télécommunications.

Je tiens à exprimer mes plus sincères remerciements à Mme Chantal Taconet pour son rôle d’encadrant, ses conseils et pour sa disponibilité.

Je remercie Dhouha Ayed pour l'aide qu’elle m’a fourni au cours de mon stage, et ses conseils.

Je remercie Salah Ammour, Nabiha Belhanafi et Nabil kouici pour le partage de leurs connaissances informatiques.
Merci à toute l’équipe systèmes répartis de l’INT.

Merci aux membres de ma famille.

Un grand merci à Khelifa et Mounir pour l’aide qu’ils m’ont fourni au cours de mon stage.
Une pensée chaleureuse à tou(te)s mes ami(e)s.

Sommaire

Table des figures

Chapitre 1

Introduction :

En raison de l'émergence des réseaux sans fil et des terminaux mobiles, les utilisateurs accèdent à leurs applications à partir d'environnements différents en terme de plate-forme logicielle et matérielle. Par conséquent, il est important que les applications s'adaptent aux différents contextes d'exécutions.
Le contexte d'exécution d'une application est l'ensemble des paramètres caractérisant son environnement d'exécution, il peut représenter les capacités du terminal, sa connexion au réseau, sa localisation, …etc.
Le déploiement d'une application sur un site donné représente toutes les étapes nécessaires à son installation et son instanciation sur ce site, et permet à l'utilisateur d'accéder à l'application.

Le concept de composant, introduit pour faciliter la conception des applications, peut faciliter l'adaptation des applications aux différentes caractéristiques du terminal en déployant les composants les mieux adaptés à ces caractéristiques.
Les modèles de composant existants comme CCM, EJB et .NET permettent de déployer les applications multi-composants d'une façon statique et sans prise en compte du changement du contexte d'exécution. Plusieurs travaux ont été initiés afin de compléter leurs fonctionnalités en leur permettant de faire un déploiement adaptatif de sorte que les composants constituant l'application soient déployés, lors de l'accès au service, en fonction du contexte dans lequel l'application va être exécutée, et que ces mêmes composants vont pouvoir changer de comportement en fonction du contexte d'exécution. Pour cela on dispose d'une batterie d'implémentations chacune d'entre elles étant adaptée à un contexte d'exécution.

Les implémentations ont besoin d'un ensemble de ressources pour qu'elles puissent s'exécuter normalement comme la bande passante et l'espace mémoire. D'autre part, le réseau constitué d'un ensemble de nœuds et de liens entre ces nœuds offre un ensemble de ressources. Le problème du placement des composants sur les nœuds consiste, d'abord, à sélectionner les implémentations à déployer et les nœuds d'instanciation, et à répartir ensuite ces implémentations sur les différents nœuds tout en optimisant l'affectation des ressources.

Dans ce rapport, nous allons développer ces deux principes fondamentaux du déploiement adaptatif, à savoir le choix des implémentations et le choix des nœuds de déploiement et proposer un algorithme pour la résolution de ce problème.

Le document est organisé de la manière suivante :
Dans le deuxième chapitre nous présentons le modèle de composants CCM (CORBA Component Model). Les chapitres 3 et 4 présentent, respectivement, les langages de descriptions des ressources requises par les composants, et les langages de descriptions des ressources offertes par les nœuds. Dans le chapitre 5, nous présentons l'algorithme de placement des composants sur les nœuds : *Sekitei*, et enfin nous présentons dans le chapitre 6 la modélisation OWL des contextes fournis par les nœuds et requis par les composants, notre algorithme de choix du placement des composants sur les nœuds et la réalisation de la proposition.

Chapitre 2

Le modèle de composants de CORBA :

1.1. Introduction :

Afin de faciliter le développement des applications, certains industriels, comme Microsoft, Sun et l'organisation de standardisation OMG ont fait évoluer leurs modèles à objets vers les composants. Le *CORBA Component Model* (CCM), proposé par l'OMG, est le modèle de base du standard CORBA3, il représente une extension du modèle objet de CORBA2.

Un modèle de composant doit permettre de concevoir les applications multi-composants en proposant aux utilisateurs les outils permettant de concevoir les composants, de les implémenter, de les assembler pour constituer des applications et de les déployer sur les différents sites.

Plusieurs architectures ont été proposées pour permettre aux modèles de composants de déployer les applications suivant le contexte d'exécution en proposant de rajouter les modules permettant de récupérer les informations de contexte, et de choisir les implémentations de composants les mieux adaptées à ce contexte. Le modèle de composant CCM donne plus de flexibilité et de contrôle de composants en offrant les API permettant d'ajouter et supprimer les composants et de modifier leurs assemblages. Pour cette raison nous avons choisi une architecture au dessus de CCM, sur laquelle nous avons conçu un algorithme de choix et d'affectation des implémentations aux nœuds.
Nous allons dans ce chapitre parler des principaux concepts du CCM en présentant les quatre modèles proposés dans la spécification CCM. Dans notre travail nous nous intéressons au déploiement des applications multi-composants sur le réseau, pour cela nous allons présenter en détail le modèle d'assemblage et de déploiement.

La figure *FIG. 2.1* présente la chaîne de production d'une application CCM, de sa conception à son déploiement :

2. La conception : Le niveau conceptuel permettant aux concepteurs de définir les différentes interfaces et propriétés du type de composant en utilisant la nouvelle version du langage OMG IDL, dans la suite de ce chapitre nous utiliserons le terme IDL3 pour désigner cette nouvelle version et IDL2 pour la version associé à CORBA2.

3. La programmation :

Un composant comporte une partie fonctionnelle (programmée) et une partie non fonctionnelle (décrites en IDL/CIDL et générées). CIDL (*Component Implementation Definition Language*) est le langage qui permet de définir la structure d'implantation d'un type de composant. Après avoir défini le type de composant, le fichier IDL3 va être projeté en un fichier IDL2 pour en générer le squelette de l'implantation de composant. Le développeur implémente ensuite la partie fonctionnelle du composant en utilisant le langage de programmation.

4. Le déploiement : L'implantation de chaque composant et son descripteur vont être agrégés dans des paquetages de composant. A leur tour, associés à leurs propriétés ainsi que les propriétés de leurs maisons, ils vont être agrégés pour constituer des paquetages d'assemblages de composant définis par des descripteurs d'assemblage.

   Les paquetages d'assemblages et de composants vont ensuite être déployés sur les différents sites.

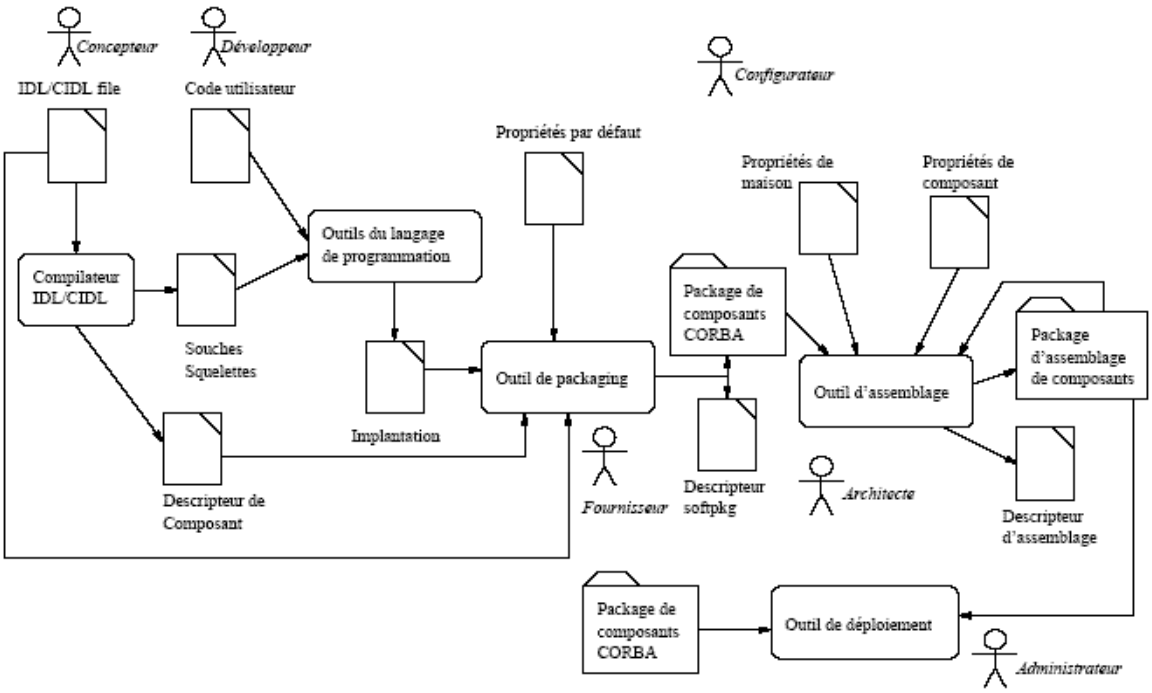


FIG. 2.1 - La chaîne de production d'une application CCM

La spécification CCM [CCM02] est découpée en quatre modèles :

4.1.Le modèle abstrait de composants :

Offre aux concepteurs le moyen d'exprimer les propriétés et les interfaces fournies et utilisées par le type de composant en introduisant une version enrichie du langage OMG IDL. Ce modèle permet aussi de définir les maisons de composants.

6.6.4.Type de composant :

Un type de composant est un méta-type de base dans CORBA, qui représente une extension du méta-type objet. Il regroupe la définition d'attributs et de ports qui représentent, respectivement, ses propriétés configurables et ses interfaces fournies et utilisées. Il peut définir quatre types de ports :

5. Facette : interface fournie par un type de composant et utilisée par d'autres types de composants en mode synchrone.
6. Réceptacle : interface utilisée par un type de composant en mode synchrone.
7. Puit d'événement : interface fournie par un type de composant et utilisée par ses clients en mode asynchrone.
8. Source d'événement : interface utilisée par un type de composant en mode asynchrone.

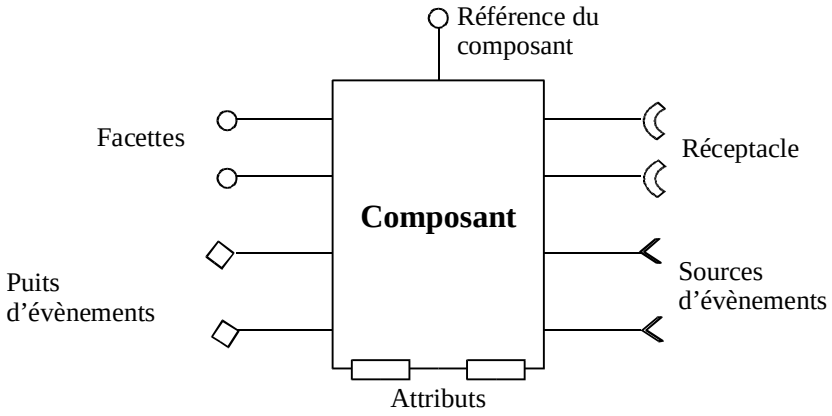


FIG. 2.2 - Le modèle abstrait d'un type de composant

6.6.4.Les maisons de composants :

Tout comme le composant, la maison de composants est un méta-type qui représente un gestionnaire pour des instances d'un même type de composant.

Elle gère le cycle de vie des instances, éventuellement en leur associant des clés primaires dans le cas des instances de composants persistantes. Pour cela, elle offre une fabrique d'instances de composants et des opérations de recherche utilisant ces clés. [ACC02]

Un type de maison doit spécifier le type de composant à gérer dont les instances peuvent être gérées par plusieurs types de composants.

8.2.Le modèle de programmation :

Un composant comporte une partie fonctionnelle qui représente le code du programmeur et une partie non fonctionnelle générée automatiquement à partir des descripteurs IDL et CIDL.
Le langage CIDL (*Component Implementation Definition Language*) permet de décrire la structure de l'implantation d'un composant.
Afin d'intégrer l'implantation fonctionnelle du composant dans la partie non fonctionnelle, le CCM inclut le framework CIF(*Component Implementation Framework*) qui décrit les interactions entre les parties fonctionnelle et non fonctionnelle de l'implantation du composant et génère les squelettes de composants en se basant sur le langage CIDL.

8.3.Le modèle d'assemblage et de déploiement :

Le modèle de déploiement est le second apport majeur des composants CORBA. Il offre des moyens pour automatiser la diffusion et la mise en place d'une application répartie. Il contribue à accroître la réutilisabilité d'entités logicielles en facilitant l'utilisation et l'intégration de composants existants [ACC02].

6.6.4.Paquetages de déploiement :

Un paquetage est l'unité de déploiement des composants CORBA qui regroupe un ensemble d'implantation de composants associés à un ou plusieurs descripteurs, ces descripteurs sont écrits en OSD (*Open Software Descriptor*). Ces différents éléments sont regroupés dans un fichier au format «ZIP».
Un descripteur décrit le contenu d'un paquetage en spécifiant ses informations générales (auteur, description, interface OMG IDL, … etc.), les descriptions de ses implantations (nom, système d'exploitation cible, langage d'implantation, … etc.) et les dépendances des composants par rapport à l'environnement.
Il existe deux types de paquetages :

Paquetage de composant :

Contient un seul composant dont le descripteur, qui est généré par le compilateur CIDL, décrit sa structure : l'héritage, les interfaces supportées et les ports, et permet de connecter les composants lors du déploiement.

Paquetage d'assemblage de composant :

Permet de déployer un ensemble de composants en offrant un patron (*template*) pour les instancier et les interconnecter. Il regroupe l'ensemble des paquetages de composants, le descripteur d'assemblage de ses composants et un ensemble de fichiers de propriétés.

Le descripteur d'assemblage de composants décrit les composants et les connexions entre les composants en utilisant des assertions telles que *connectinterface* ou *connectevent*. Le processus de déploiement se base sur ce descripteur et le projette sur des sites physiques.

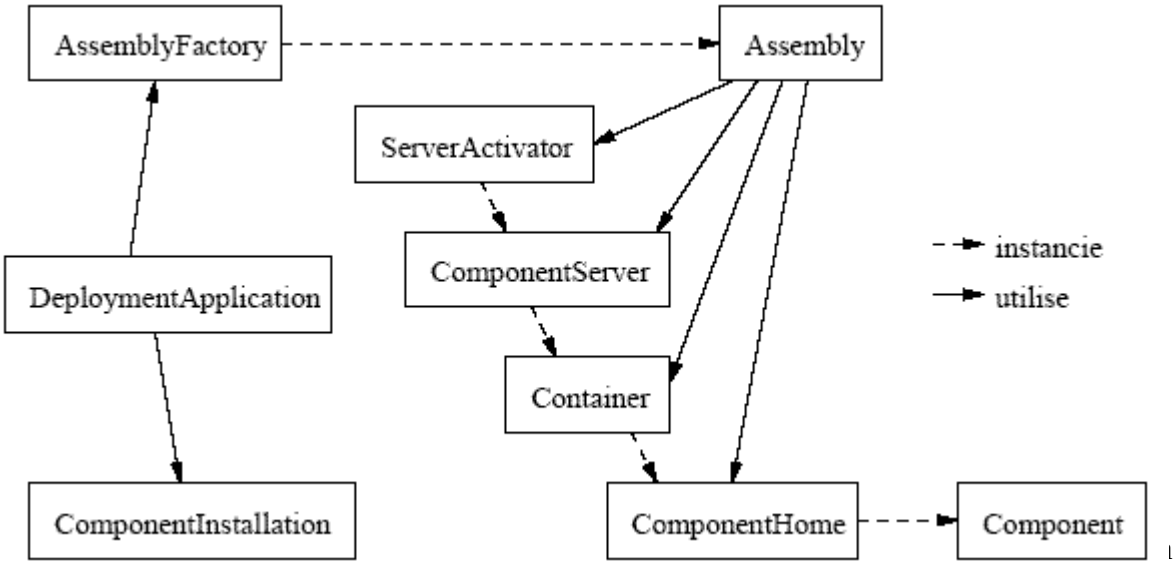
Les fichiers de propriétés permettent de configurer les composants et les maisons de composants en affectant leurs attributs par des valeurs par défaut, ces attributs peuvent être modifiés par l'utilisateur.

6.6.5.Le déploiement :

Consiste à déployer les paquetages de composants et d'assemblages de composants sur les sites cibles, en installant, instanciant et configurant les composants et les connexions suivant les quatre étapes suivantes :

9. Définition et choix des sites cibles et installation des implantations.
10. Instanciation des maisons puis des composants.
11. Connexion des composants.

Le schéma suivant présente le processus de déploiement :



utilisé par l'instance d'*Assembly* et détermine les sites sibles.

FIG. 2.3 - L'architecture du processus déploiement.

13. L'outil de déploiement utilise *ComponentInstallation* de chaque site pour installer les implantations de composants requises par ce site.
14. Une seule instance d'*Assembly* sera créée, pour toute l'application, sur un site déterminé en utilisant la fabrique d'assemblage (*Assemblyfactory*).

15. L'instance d'*Assembly* crée les maisons et les instances de composants sur leurs sites en se basant sur le descripteur d'assemblage.
16. La création d'un serveur de conteneur en utilisant l'instance de *ServerActivator* du site cible.
17. Instanciation d'un conteneur en utilisant l'instance du serveur de conteneur.
18. L'installation d'une implantation de la maison et la création de son instance par le conteneur.
19. L'instance d'*Assembly* crée une instance de composant en utilisant la maison de composant.
20. Configuration de la nouvelle instance de composant si c'est nécessaire.
21. Après avoir installé tous les composants, l'instance d'*Assembly* connecte leurs instances en se basant sur le descripteur d'assemblage.
22. L'instance d'*Assembly* notifie aux instances de composants que leurs connexions initiales ont été effectuées en invoquant leurs opérations *Configuration_Complete*.

22.1.Le modèle d'exécution :

Un conteneur est un environnement d'exécution pour une instance de composant CORBA. Cet environnement est implanté comme un serveur d'applications et/ou comme une plate-forme de production de telles applications [ACC02].

Un serveur de composant contient un nombre arbitraire de conteneurs dont chacun gère les instances d'un seul type de composant. Chaque conteneur comporte un POA (*Portable Object Adapter*) qui est utilisé pour créer des références qui seront exportées vers les clients et gérer l'activation d'instances de composants à la réception des requêtes.

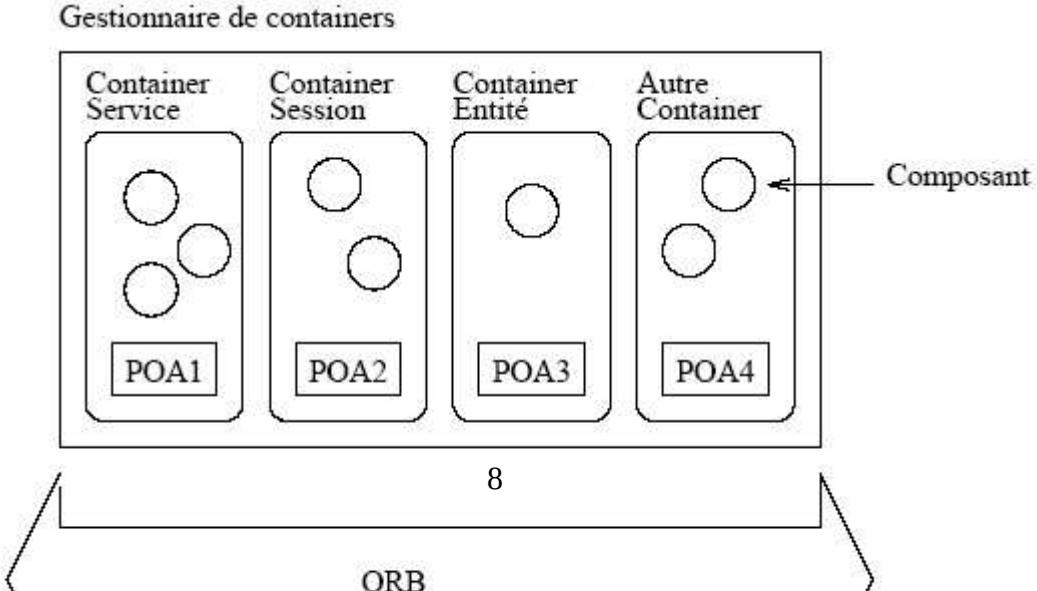


FIG. 2.4 - L’architecture d’un serveur de composants

22.2.Conclusion :

Le modèle de composant CCM permet de concevoir les composants en spécifiant leurs interfaces (fournies et utilisées) et leurs attributs, d’utiliser des outils d’assemblages pour créer des applications en les interconnectant, et de déployer les applications sur un ensemble de sites. Le déploiement standard CCM n’autorise pas encore la prise en compte automatique du contexte d’exécution lors du déploiement. Mais, il offre des API permettant d’ajouter, de supprimer et d’assembler les composants sans avoir besoin d’outils d’assemblage, ce qui donne à l’utilisateur la possibilité de contrôler les composants et de changer leurs assemblages en fonction du changement du contexte d’exécution.

Afin de compléter les fonctionnalités du CCM en lui permettant de déployer les composants suivant le contexte d’exécution, on propose dans l’architecture présentée dans le chapitre 5 de rajouter les modules permettant de collecter les informations de contexte, de choisir les implémentations à déployer et les nœuds d’instanciations et de générer un plan de déploiement qui indique que telle implémentation sera déployée sur tel noeud, ce plan sera déployé par CCM sur les sites physiques.

Pour que ce processus de déploiement fonctionne correctement, nous devons, d’abord, décrire les informations de contexte pour pouvoir les stocker, les échanger et les analyser, et ce en utilisant les langages de descriptions qui seront présentés dans le chapitre suivant.

Chapitre

```
<SOFTPKG NAME=" Solitaire" VERSION="1,0,0,0">
        <TITLE>Solitaire</TITLE>
        <ABSTRACT>description du solitaire </ABSTRACT>
        <IMPLEMENTATION>
                <OS VALUE="WinNT"><OSVERSION VALUE="4,0,0,0"/></OS>
                <PROCESSOR VALUE="P3" />
                <LANGUAGE VALUE="en" />
                <CODEBASE HREF="/ solitaire/implementation/solitaire.cab" />
        </IMPLEMENTATION>

        <IMPLEMENTATION>
                <IMPLTYPE VALUE="Java" />
                <CODEBASE HREF="/solitaire/implementation/solitaire.jar" />
                <!— Cette implémentation a besoin de l'objet carte -->
                <DEPENDENCY>
                        <CODEBASE HREF="/ solitaire/ cartes/cartes.osd" />
                </DEPENDENCY>
        </IMPLEMENTATION>
</SOFTPKG>
```

Langages [illegible] cution d'un com[illegible]

22.3.Int[illegible]

Les applicati[illegible] ngements du contexte [illegible] ensemble d'implémentations, chacune d'entre elles étant conçue pour être déployée dans un contexte bien déterminé en spécifiant ses besoins en ressources. Ces ressources doivent être prises en compte lors du déploiement initiale de l'application pour placer au mieux les composants sur les nœuds. Pour pouvoir stocker et analyser ces ressources, on doit les modéliser en utilisant un langage de description des ressources.

Plusieurs langages de descriptions ont été proposés pour exprimer les besoins des implémentations en ressources et qui diffèrent par leurs capacités de décrire toute sorte de contrainte.

Nous allons dans ce chapitre présenter les principaux langages de description des ressources nécessaires à l'exécution d'un composants à savoir OSD (*Open Software Description*) et DSD (*Deployable Software Description*), avec une comparaison entre leurs capacités de modélisation en terme de vocabulaires proposés, extensibilité et description des contraintes et des dépendances.

22.4.Open Software Description (OSD) :

6.6.4.Description :

OSD est un langage de description des paquetages de logiciels, la spécification d'OSD propose une DTD XML qui fournit le vocabulaire permettant de décrire : les composants, les relations de dépendances entre les composants et leurs ressources nécessaires en terme de système d'exploitation, processeur et l'espace disque.

La DTD OSD présentée dans l'annexe A permet de fournir les descriptions suivantes :

Informations générales :

Les informations concernant le logiciel à savoir : le nom, la version, le titre, la description et la licence.

Les implémentations :

Une implémentation du logiciel est une configuration qui est définie par un certains nombre de propriétés comme le système d'exploitation, le processeur et le fichier contenant cette configuration.

Les dépendances :

La DTD de DSD permet de décrire les dépendances logicielles de sorte que l'implémentation en question ne puisse être choisie que si l'objet correspondant soit présent sur le site concerné.

6.6.5.L'utilisation d'OSD :

Le vocabulaire OSD est extensible en utilisant les *namespaces* XML et peut être utilisé pour plusieurs buts :

23. L'utilisation autonome : Déclaration des ressources nécessaires pour les composants afin d'installer seulement ceux qui sont adaptés à la configuration de la machine cible.
24. Utilisation pour les fichiers d'archives : les fichiers d'archives comme *Java Archive* (JAR) et *Cabinet* (CAB) peuvent utiliser le vocabulaire OSD pour décrire les dépendances requises pour l'installation des différentes pièces du logiciel.
25. Référence dans les pages HTML : Le vocabulaire OSD peut être utilisé pour définir les différents modules requis par les pages WEB en indiquant comme référence le fichier OSD.
26. Distribution automatique : Les applications «*push*» peuvent utiliser le vocabulaire OSD pour lancer l'installation automatique des logiciels, dans ce cas, le fichier OSD fournit les informations nécessaires à l'installation des composants.

Dans ce rapport nous nous intéressons à l'utilisation autonome du vocabulaire OSD, l'exemple suivant présente une instance de la DTD OSD et décrit le paquetage du logiciel *solitaire* en spécifiant les ressources requises par la première implémentation (système d'exploitation, processeur et langue) et en indiquant la dépendance de la deuxième implémentation du composant *cartes* de sorte qu'il doit être installé pour que cette implémentation puisse être sélectionnée.

Après avoir analysé le fichier OSD, le client détermine l'implémentation qui s'adapte à sa configuration.

26.1.Deployable Software Description (DSD) :

6.6.4.Description :

Le DSD, proposé dans le projet Software Dock [HAL99], définit un schéma standard permettant de décrire les besoins des logiciels pour l'automatisation du cycle de vie de déploiement qui représente toutes les opérations permettant d'installer, reconfigurer, adapter, mettre à jour et désinstaller l'application.

La DTD de DSD présentée dans l'annexe B permet de fournir les descriptions suivantes :

Informations générales :

L'élément *ID* fournit la description du logiciel en spécifiant son nom, son producteur, sa licence, son logo et sa signature.

Les propriétés :

Deux types de propriétés sont définis par cette DTD :

27. Les propriétés externes du logiciel qui sont spécifiées par *ExternalProperties* et peuvent représenter : le système d'exploitation, l'architecture du site, …etc. Ces informations ne seront connues que pendant le déploiement de l'application et servent à garder la trace de ces propriétés pour reconfigurer ou annuler l'opération de déploiement dans le cas du changement de leurs valeurs.
28. Les propriétés internes du logiciel qui sont définies par *Properties*, l'exemple suivant définit la propriété implémentation qui peut prendre les valeurs : java (la valeur par défaut) et native, il définit également ses valeurs par défauts d'activation ou de désactivation en utilisant les balises : *DefaultEnabled* et *DefaultDisabled*.

Une valeur d'activation (désactivation) sert à spécifier la valeur d'une propriété dans les règles de compositions.

```
<Properties>
        <Property>
                <Name>Implementation</Name>
                <VarType Value="string"></VarType>
                <Description>
                        Selects implementation
                </Description>
                <DefaultValue>Java</DefaultValue>
                <DefaultEnabled>Native</DefaultEnabled>
                <DefaultDisabled>Java</DefaultDisabled>
                <TopLevel Value="true"></TopLevel>
                <Values>
                        <Value>Native</Value>
                        <Value>Java</Value>
                </Values>
        </Property>
</Properties>
```

Les règles de composition :

Définissent les relations entre les propriétés internes d'une famille de logiciel suivant une condition donnée, il existe quatre type de relation :

29. *any-of* : Indique la sélection de n'importe qu'elle propriété mentionnée dans *RuleProperties*.
30. *one-of* : La séléction d'une seule propriété.
31. *excludes* : Désactive la proprité mentionnée dans *RuleProperties*, c-à-d elle va prendre la valeur de désactivation qui est définie dans *properties.*
32. *includes* : Active la proprité mentionnée dans *RuleProperties.*

L'exemple suivant définit la relation *excludes* sur l'*implementation* de sorte qu'elle soit *Java* sous un système d'exploitation différent de Win2000 et LINUX.

```
<Composition>
        <CompositionRule>
                <Condition> (($OS$ != «Win2000») AND ($OS$ != «LINUX»))
        </Condition>
                <ControlProperty> </ControlProperty>
                <Relation Value=«excludes»> </Relation>
                <RuleProperties>
                        <Name> Implementation </Name>
                </RuleProperties>
        </CompositionRule>
</Composition>
```

Les assertions et les dépendances :

Définit les contraintes sur les valeurs de propriétés internes et externes du logiciel.

Une contrainte est définie par une condition qui doit être vérifiée pour que le déploiement ait lieu, au contraire d'une dépendance qui définit une procédure de résolution de conflit. L'exemple suivant spécifie des contraintes sur le système d'exploitation et indique que le

composant *cartes* doit être installé pour que l'on puisse utiliser la version Java, dans le cas contraire, il l'installe.

```
<Assertions>
        <Assertion>
                <Condition>
                        (($OS$==«Win98») || ($OS$==«WinNt») ||
                         ($OS$==«Solaris») || ($OS$==«Linux»))
                </ Condition >
                 <Description>
                        Seulement sous Win98, WinNT, Slorais, and Linux
                </Description>
         <Assertion>
</Assertions>
<Dependencies>
            <Guard></Guard>
            <Dependency>
                    <Guard>($Implementation$ == "Java")</Guard>
                    <Condition>
                            (!installed("cartes"))
                    </Condition>
                    <Description>
                            La version Java dépend du composant cartes.
                    </Description>
                    <Resolution>Install</Resolution>
            </Dependency>
</Dependencies>
```

Les fichiers de configurations :

Définis dans l'élément *Artifacts* qui spécifie les fichiers contenant les différentes configurations du logiciel, l'exemple suivant indique l'emplacement des classes Java de l'application *Solitaire*.

```
<Artifacts>
        <Artifacts>
                 <Guard> ($Implementation$ == «Java») </Guard>
                <Signature> 96429</Signature>
                 <ArtifactType> CLASSEFILE </ArtifactType>
                <SourceName> solitaire.class </ SourceName >
                <Source> /solitaire/java/classes </ SourceName >
                <DestinationName> solitaire.class </ DestinationName >
                <Destination> classes </Destination>
        </Artifact>
</Artifacts>
```

6.6.4.Le processus de déploiement :

Le DSD permet de décrire les logiciels en terme de propriétés, dont l'ensemble des valeurs valides représente une configuration valide du logiciel. Le choix d'une telle configuration se fait en trois étapes :

33. Les valeurs de propriétés sont choisies suivant les règles de compositions.
34. Ces valeurs seront ensuite utilisées pour sélectionner l'ensemble des fichiers (*artifacts*) et ce suivant les différentes contraintes et dépendances.
35. Enfin, ces fichiers génèreront la configuration désirée.

Le déploiement représente l'ensemble des opérations permettant d'installer, adapter, mettre à jour, reconfigurer et désinstaller l'application, il peut être considéré comme une transformation de la configuration du logiciel en passant d'une configuration valide à une autre.

35.1.Comparaison entre les langages de descriptions :

Le cycle de vie du déploiement représente l'ensemble des opérations qui permettent d'installer, mettre à jour, adapter et désinstaller les applications.

Le schéma suivant, proposé dans [HAL97], présente le cycle de vie du déploiement.

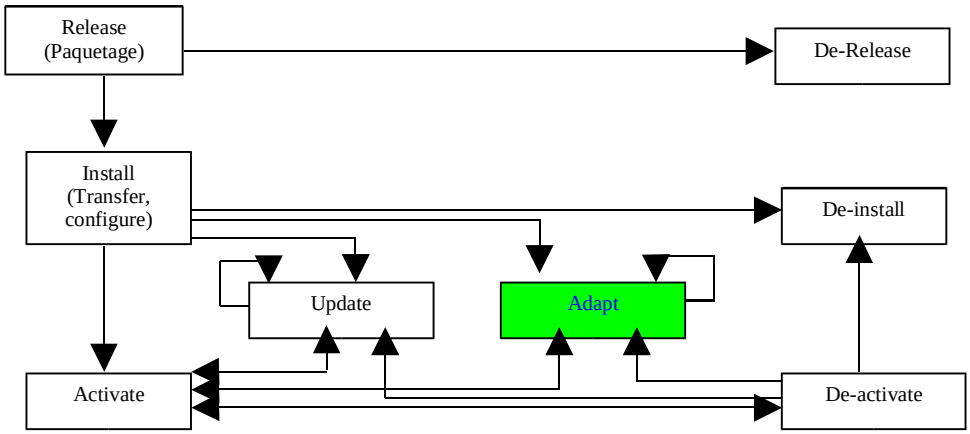


FIG. 3.1 - Le cycle de vie de déploiement

Afin d'assurer ce cycle de vie, on doit fournir les descriptions permettant d'exécuter les différentes opérations, à savoir : description de contraintes et de dépendances.

Les langages de descriptions diffèrent par leurs capacités de fournir ces descriptions.

Le tableau suivant présente une comparaison entre les langages OSD et DSD en terme de vocabulaire proposé, les capacités de descriptions et l'extensibilité.

Langage	**Description**
OSD	36. Schéma réduit (moins de vocabulaire). 37. Les dépendances entre les composants. 38. Les contraintes sur les propriétés des composants. 39. Extensible.
DSD	40. Plus de vocabulaires. 41. Les dépendances entre les propriétés du logiciel. 42. Les contraintes sur les propriétés. 43. Extensible

43.1.Conclusion :

Nous avons présenté dans ce chapitre les langages de descriptions des ressources nécessaires à l'exécution des composants à savoir OSD et DSD ainsi qu'une comparaison entre leurs capacités de description.

Ces deux langages permettent de décrire les contraintes et les dépendances entre les propriétés des composants. Mais ils ne permettent pas de concevoir des schémas où on spécifie les classes de ressources et leurs relations ni de définir des relations d'héritages entre ces classes.

Dans le chapitre suivant, nous allons présenter d'autres langages de descriptions qui donnent la possibilité de décrire ce genre d'informations et même de modéliser la sémantique des ressources.

Chapitre 4 :

Langages de descriptions des ressources offertes par les nœuds

43.2.Introduction :

Notre travail consiste à choisir et affecter les implémentations aux nœuds d'instanciation suivant le contexte d'exécution. Le contexte d'exécution est l'ensemble de tous les attributs détectables caractérisant l'environnement d'exécution, il peut représenter les capacités du terminal, sa localisation, son profil, …etc. Ces informations doivent être envoyées au serveur de déploiement pour que l'on puisse déployer les implémentations de composants les mieux adaptées au contexte d'exécution. Pour cela, on doit décrire ces informations en spécifiant leurs valeurs, propriétés et relations.

Plusieurs langages de descriptions ont été proposés dans la littérature, les sections suivantes en présentent les plus utilisés pour la modélisation du contexte.

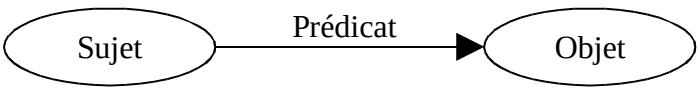
43.3.Resource Description Framework (RDF) :

6.6.4.Description :

Les applications web ont besoin, de plus en plus, des méta-données pour les ressources web. Le moteur de recherche, par exemple, utilise une base de méta-données pour chercher efficacement des documents. Pour cette raison, un modèle de méta-données est nécessaire pour que les applications puissent les utiliser et les échanger. RDF (*Resource Description Framework*), recommandé par W3C (*World Wide Web Consortium*), est un framework de représentation des ressources dans le web. Ces ressources sont identifiées par des URI (*Uniform Resource Identifiers*).

6.6.5.Le modèle de données :

L'élément de base du modèle RDF est le triplet : sujet, prédicat et objet. Il peut être représenté par le graphe suivant :

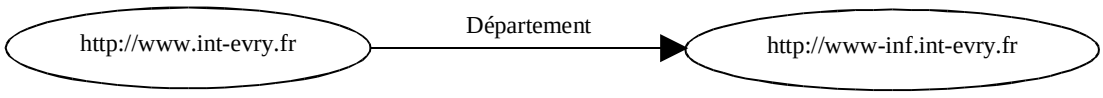


Sujet : la ressource à décrire qui est représentée par un URI.

Objet : représente une autre ressource ou une valeur littérale.

Prédicat : définit une propriété du sujet dont la valeur est l'objet.

L'exemple suivant indique que INF est un département de l'INT.



6.6.6.Apporter la sémantique pour la modélisation :

La modélisation RDF donne la capacité d'échanger le modèle, mais elle ne permet pas à l'utilisateur de définir des vocabulaires qui portent la sémantique de description de ressources [TUA]. RDFS (RDF Schéma) permet d'étendre le vocabulaire RDF en définissant le schémas correspondant au contexte d'utilisation. il se base sur le concept de triplet (sujet, prédicat et objet) et permet de définir des classes, des sous-classes, des propriétés et des sous-propriétés.

On peut distinguer trois types de concepts dans RDF [PIE01] : concepts élémentaires, concepts pour la définition de schéma et concepts utilitaires.

Concepts élémentaires :

Ces concepts correspondent aux éléments du modèle RDF, ils comportent : *rdf :ressource* qui permet de décrire les ressources, *rdf :property* pour les prédicats et *rdf :statement.*

Concepts pour la définition de schéma :

RDFS permet de définir de nouveaux termes en se basant sur les concepts suivants :

rdf :Type pour spécifier le type du nouveau terme, ce type peut être une classe de ressource ou une propriété.

rdfs :Class représente le type classe de ressource.

rdfs:subClassOf permet de définir une sous classe d'une classe de ressource.

rdfs:subPropertyOf Un terme de type propriété peut décrire une sous classe d'une autre propriété.

rdfs:domain spécifie la classe de ressource sur laquelle la propriété va s'appliquer.

rdfs:range spécifie la classe de ressource dans laquelle la priorité va prendre de valeur.

rdfs:Literal Une propriété peut prendre une valeur de type littéral.

Concepts utilitaires

RDF schéma définis aussi d'autres concepts qui sont utilisés dans la plupart des applications comme les containers (*rdfs:Container*), les commentaires (*rdfs:comment*), …etc.

6.6.7.Représentation en XML :

L'élément de base du modèle RDF peut être représenté par la description suivante :

<rdf:Description about=La sujet à décrire >

<prédicat>L'objet qui peut être un autre sujet </prédicat>

<rdf:Description/>

L'exemple précédent peut être représenté par l'écriture suivante :

<rdf:Description about= «http://www.int-evry.fr» >

<Departement> «http://www-inf.int-evry.fr » </ Departement >

<rdf:Description/>

Pour définir des schémas RDF, on doit ajouter des nouveaux termes qui représentent les nouvelles classes, sous classes et propriétés en utilisant les concepts de définitions de schémas présentés dans la section précédentes. La description suivante permet de déclarer une classe :

<classe id="Le nom de la classe">
...
</classe>

Dans l'exemple suivant, nous allons définir le schéma permettant de décrire le terminal utilisateur en terme de système d'exploitation et du navigateur web. Les deux classes *Navigateur* et *OS* ont deux propriété communes : *Nom* et *Version*, pour cela on a rajouté la classe *Logiciel* dont *Navigateur* et *OS* sont des sous classes, cette classe prend comme propriété *Nom* et *Version*.

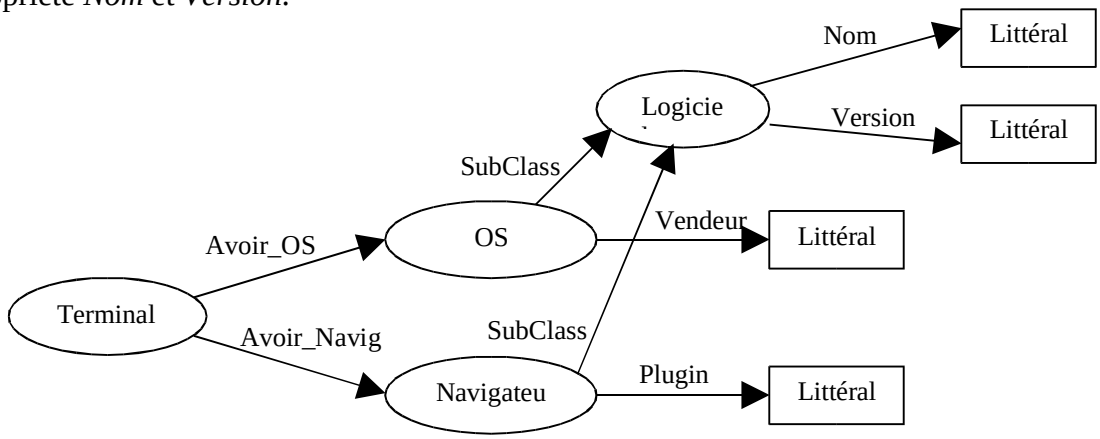


FIG. 4.1 - Exemple de schéma RDFS

La représentation XML de ce schéma est la suivante :

```
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">
<rdfs:Class rdf:ID="Terminal" />
<rdfs:Class rdf:ID="Logiciel" />
<rdfs:Class rdf:ID="Navigateur">
    <rdfs:subClasseOf rdf:resource="#Logiciel" />
</rdfs:Class>
<rdfs:Class rdf:ID="OS">
    <rdfs:subClasseOf rdf:resource="#Logiciel" />
</rdfs:Class>

<rdf:Property rdf:ID="Avoir_Navig">
    <rdfs:domain rdf:resource="#Terminal" />
    <rdfs:range rdf:resource="#Navigateur" />
</rdf:Property>
```

```
    <rdf:Property rdf:ID="Avoir_OS">
        <rdfs:domain rdf:resource="#Terminal" />
        <rdfs:range rdf:resource="#OS" />
    </rdf:Property>
    <rdf:Property rdf:ID="Nom">
        <rdfs:domain rdf:resource="#Logiciel" />
        <rdfs:range rdf:resource="rdfs:Literal" />
        </rdf:Property>
    <rdf:Property rdf:ID="Version">
        <rdfs:domain rdf:resource="#Logiciel" />
        <rdfs:range rdf:resource=" rdfs:Literal" />
    </rdf:Property>
    <rdf:Property rdf:ID="Plugin">
        <rdfs:domain rdf:resource="#Navigateur" />
        <rdfs:range rdf:resource=" rdfs:Literal" />
    </rdf:Property>
    <rdf:Property rdf:ID="Vendeur">
        <rdfs:domain rdf:resource="#OS" />
        <rdfs:range rdf:resource="Literal" />
    </rdf:Property>
</rdf:RDF>
```

La figure nent RDF qui décrit le termina b :

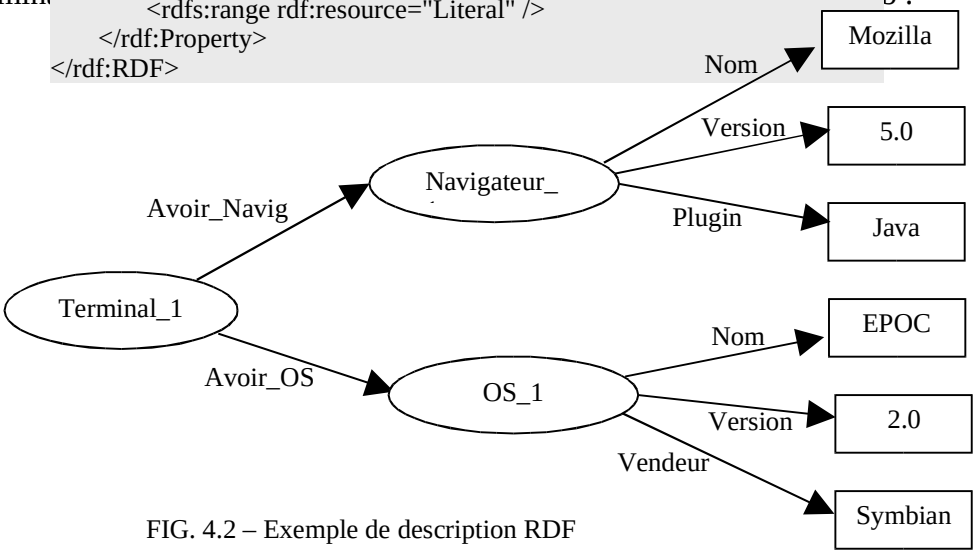


FIG. 4.2 – Exemple de description RDF

L'écriture XML de cette description est la suivante :

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:ex="L'exemple précédente#">
<ex :Terminal rdf :ID= «Terminal_1» >
    <ex :Avoir_Navig>
        <ex :Navigateur rdf :ID= «Navigateur_1» >
            <ex :Nom > « Mozilla» </ ex :Nom >
            < ex :Version > «5.0» </ ex :Version>
            < ex :Plugin> « Java» </ ex :Plugin >
        </ ex :Navigateur >
    </ex :Navigateur >
    <ex :Avoir_OS>
        <ex :OS rdf :ID= «OS_1» >
            <ex :Nom > « EPOC» </ ex :Nom >
            < ex :Version > «2.0» </ ex :Version>
            < ex :Vendeur> «Symbian» </ ex :Vendeur >
        </ ex :OS>
    </ex :Avoir_OS>
</ ex :Terminal>
```

43.4.Composite Capabilities / Preference Profiles (CC/PP) :

6.6.4.Description :

Le CC/PP, proposé par W3C, est un standard de description des capacités du terminal et des préférences utilisateurs. Il se base sur RDF et permet de fournir une description sur deux niveaux : composant et attribut comme présenté dans le schéma suivant :

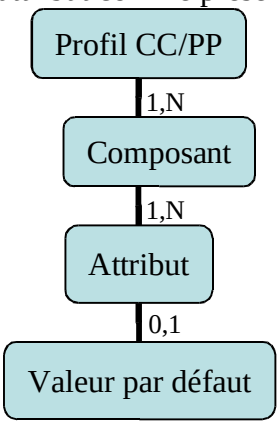


FIG. 4.3 – Les niveaux de descriptions de CC/PP

Un profil CC/PP contient un ou plusieurs composants, qui contiennent à leurs tours un ou plusieurs attributs. Chaque composant est représenté par une ressource de type *ccpp:Component*, par exemple le navigateur. Un attribut est une propriété du composant comme la version du navigateur.

Les attributs peuvent être incorporés avec les composants, ou être déclarés en faisant référence à un profil par défaut qui peut être stocké séparément. Ce qui améliore énormément le transfert du profil en ne transférant qu'une partie de la description.

Nous rajoutons à l'exemple précédent la taille de l'écran et nous utilisons CC/PP pour décrire les capacités du terminal utilisateur en définissant trois composants : la plate-forme matérielle, la plate-forme logicielle et le navigateur :

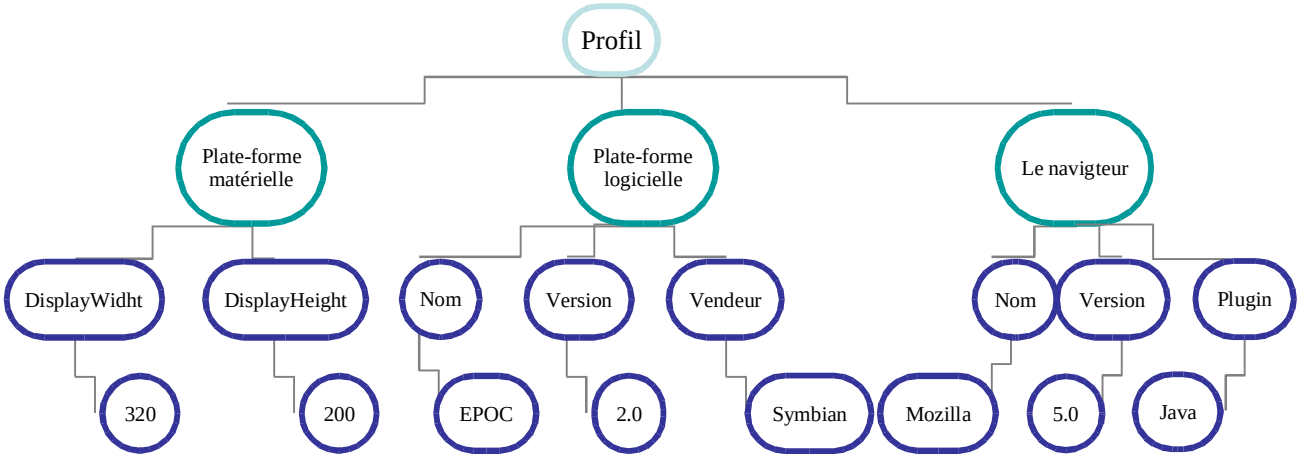


FIG. 4.4 – Exemple de description CC/PP

43.5.OWL :

6.6.4.Introduction :

Le Web sémantique due à Tim Berners-Lee, est une vision future du Web comme un espace d'échange de ressources entre êtres humains et machines facilitant le traitement automatique des informations disponible sur le Web par des machines ayant des capacités accrues à accéder aux contenus des ressources et effectuer des raisonnements sur ceux-ci grâce à une représentation formelle de leurs sémantiques. Par conséquent le langage de description doit dépasser les mots clés et spécifier la signification de ces ressources. OWL, développé par *W3C Web Ontology Working Group*, est un langage de description à base d'ontologie permettant de décrire les classes de ressources et leurs relations.

Le *Webster's Third New International Dictionary* définit l'ontologie comme:

«1. *A science or study of being: specifically, a branch of metaphysics relating to the nature and relations of being; a particular system according to which problems of the nature of being are investigated.*

2. A theory concerning the kinds of entities and, specifically, the kinds of abstract entities that are to be admitted to a language system. »

Historiquement, l'ontologie est la branche principale de la métaphysique et représente la science qui étudie l'existant et décrit les entités dans le monde et la façon dont elles sont reliées. Dans le domaine d'intelligence artificielle, nous trouvons plusieurs définitions similaires de l'ontologie dont nous citons : « *an ontology is a formal explicit description of concepts and relations in a domain of discourse, including properties of each concept and constraints expressed as axioms*».

Les ontologies permettent de décrire l'hiérarchie des classes comme RDF/RDF schéma, de représenter la sémantique des ressources et d'effectuer des raisonnements sur celles-ci pour déduire des nouvelles informations, le schéma suivant présente les principaux standards de descriptions des ressources :

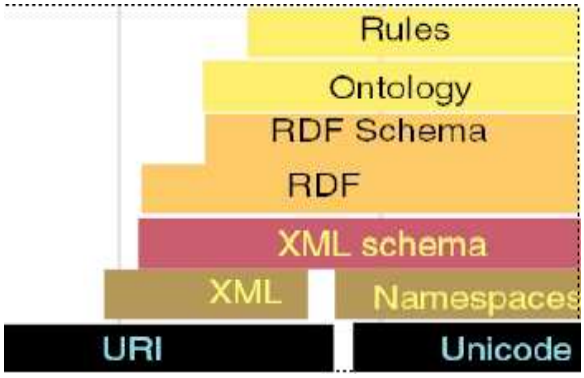


FIG. 4.5 – Les principaux standards de descriptions des ressources

OWL est un langage d'ontologie permettant de décrire le contenu des informations grâce à un vocabulaire supplémentaire et une sémantique formelle, il se base sur la capacité de XML à définir des systèmes de balisages personnalisés et sur la souplesse de RDF à représenter les données. L'ontologie OWL qui décrit le terminal utilisateur cité dans l'exemple précédent est la suivante :

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
              xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
              xmlns:owl="http://www.w3.org/2002/07/owl#">

<owl:Class rdf:ID="Terminal" />
<owl:Class rdf:ID="Logiciel" />
< owl:Class rdf:ID="Navigateur">
       <rdfs:subClasseOf rdf:resource="#Logiciel" />
</owl:Class>
< owl:Class rdf:ID="OS">
       <rdfs:subClasseOf rdf:resource="#Logiciel" />
</ owl:Class>

<owl:ObjectProperty rdf:ID="Avoir_Navig">
      <rdfs:domain rdf:resource="#Terminal" />
      <rdfs:range rdf:resource="#Navigateur" />
</ owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="Avoir_OS">
<rdfs:domain rdf:resource="#Terminal" />
<rdfs:range rdf:resource="#OS" />
</ owl:ObjectProperty>

<owl:DataProperty rdf:ID="Nom">
      <rdfs:domain rdf:resource="#Logiciel" />
      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#String" />
</ owl: DataProperty >
<owl: DataProperty rdf:ID="Version">
      <rdfs:domain rdf:resource="#Logiciel" />
      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#String" />
</owl: DataProperty >
<owl: DataProperty rdf:ID="Plugin">
      <rdfs:domain rdf:resource="#Navigateur" />
      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#String" />
</owl: DataProperty >
<owl: DataProperty rdf:ID="Vendeur">
      <rdfs:domain rdf:resource="#OS" />
      <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#String" />
</owl: DataProperty >
</rdf:RDF>
```

6.6.5.Les sous langages de OWL :

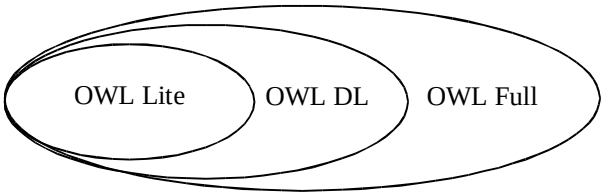
OWL fournit trois sous langages d'expressivité croissante [OWLO04], destinés à des communautés spécifiques de développeurs et d'utilisateurs :

44. OWL Lite : destiné aux utilisateurs ayant principalement besoin d'une hiérarchie de classification et de contraintes simples, par exemple, des valeurs de cardinalités 0 ou 1. Il est plus facile de mettre en œuvre des outils pour OWL Lite que pour ses parents qui sont plus expressifs.

45. OWL DL : destiné aux utilisateurs qui demandent une expressivité maximale tout en gardant la complétude du calcul (toutes les inférences sont calculables), et la décidabilité (tous les calculs se terminent dans un intervalle de temps fini).

46. OWL Full : destiné aux utilisateurs qui veulent une expressivité maximale et la liberté syntaxique de RDF sans garantie de calcul. L'avantage de OWL Full, est sa compatibilité avec RDF (syntaxiquement et lexicalement) de sorte que tout document RDF légal soit un document OWL Full légal aussi.

Chacun de ces sous langages représente une extension de son prédécesseur, comme présenté dans le schéma suivant :



46.1.COACH :

COACH, proposé dans *Information Society Technologies (IST) programm*[COA03], permet de décrire la topologie et les propriétés de l'environnement de déploiement, en définissant deux niveaux de descriptions :

6.6.4.La description des propriétés du réseau :

Consiste à décrire le réseau physique en terme de nœuds et liens de communications entre les nœuds, le schéma suivant présente la DTD de ce descripteur:

```
<!ELEMENT dci (
                        (description)?,  (authority)?,
                        ((dcimgrref)  |  ((dcimgrref)?,
                (nodes) ,       (nodelinks) ? ,
                        (services) ? ,    (assemblies)?) )
                        )
>
<!ATTLIST dci
        name            ID      #REQUIRED
        networkname     ID      #REQUIRED
        fileversion     ID      #REQUIRED
        date            ID      #REQUIRED
>
```

La topologie du réseau est définie essentiellement par les éléments *Nodes* et *NodeLinks* :

47. L'élement *Nodes* permet de déclarer les nœuds en faisant référence à leurs descripteurs de propriétés.
48. *NodeLinks* définit les liens de communication entre les nœuds (les deux nœuds de chaque lien, son nom et son type : ATM, Ethernet, …etc.) ainsi que leurs propriétés comme la bande passante maximale (en KByte) et l'utilisation maximale (%).
49. *Services* déclare les services offerts par le réseau et le middleware (nom, type : *objectservice* ou *networkservice*, disponibilité, description, et référence sur l'objet CORBA qui offre ce service), par exemple le service Telnet.
50. *Assemblies* déclare l'assemblages des composants déjà déployés sur le site (l'identifiant d'assemblage, une description, et l'état : *installed*, *running*, *unknown*).

6.6.4.La description des capacités et propriétés des nœuds :

Décrit les capacités et propriétés du nœud, indépendamment du réseau, en terme de matériel disponible : le processeur, la mémoire, …etc, et de logiciel installé : système d'exploitation, L'ORB, le compilateur,…etc.

Le schéma suivant présente la DTD de ce descripteur :

```
<!ELEMENT node (
```

```
                        (description)?,    (authority) ?,
                        (os),              (networkadress)*,
                        (processor) *,     (memory) ?,
                        (storage) *,       (installedhardware) ?,
                        (installedsoftware) ?, (properties) ?,
                       )
>
<!ATTLIST node
        id              ID      #REQUIRED
>
```

Cette DTD permet de fournir les descriptions suivantes :

- **Description du Software :**

  Spécifie le système d'exploitation (nom, version, vendeur, répertoire, description) et le software installé sur le nœud comme le langage de programmation, l'ORB et les protocoles réseau.

- **Description du Hardware :**

  Permet de décrire :

  - Le processeur : le type (Athlon, PowerPC, …), la fréquence, le vendeur, …
  - La mémoire : La taille de la mémoire en KByte
  - Le matériel et son driver installé sur le nœud : nom (clavier, souris, …), vendeur, driver, version et description.

- **Description de la capacité de stockage :** La capacité de stockage en KByte, le type de stockage (disque dur, …).

50.2.Comparaison entre les langages de descriptions :

Le tableau suivant montre une comparaison entre les différents langages de descriptions des ressources offertes par le réseau en terme de vocabulaire proposé, niveaux de descriptions et extensibilité :

Langages	**Descriptions**
RDF	51. Schéma généralisé. 52. Plusieurs niveaux de descriptions 53. Extensible (RDFS).
CC/PP	54. Schéma réduit (moins de vocabulaire). 55. Deux niveaux de descriptions (composant et attribut). 56. Extensible.
OWL	57. Basé sur RDF. 58. Langage à base d'ontologie. 59. La description de la sémantique.
COACH	60. Description de la topologie du réseau. 61. Extensible.

61.1.Conclusion :

Les ressources offertes par le réseau et celles requises par les implémentations doivent être modélisées pour que l'on puisse les échanger, les stocker et les analyser en utilisant un des langages de descriptions cités précédemment. Ces langages diffèrent par leurs capacités de description et sont classés en deux catégories :

62. les langages de modélisation du besoin des implémentations en ressource comme DSD et OSD
63. les langages de modélisations du contexte d'exécution comme RDF/RDFS et OWL qui sont utilisés dans des domaines variés (l'adaptation des application au contexte, Web sémantique, base de données, …etc.), et qui permettent de définir des schémas en spécifiant les classes de ressources et leurs relations, par conséquent ils peuvent être utilisés pour modéliser les deux types de ressources.

OWL se base sur RDF et permet de décrire la sémantique des ressources et de raisonner sur celles-ci pour déduire des nouvelles informations. Dans notre modèle, nous avons besoin de décrire les classes de ressources et leurs relations pour pouvoir adapter les applications aux changements du contexte d'exécution. Pour une éventuelle extension du modèle où l'aspect sémantique sera pris en compte pour déduire des informations du genre «si le nœud offre la

ressource dont la valeur satisfait l'implémentation alors cette implémentation pourra être déployée sur ce nœud», nous avons conçu une ontologie OWL permettant de décrire les ressources disponibles sur le réseau, et celles requises par les implémentation. Cette ontologie sera présentée dans le chapitre 6.

Chapitre 5

Algorithme de placement des composants sur les nœuds : *Sekitei*

63.1.Introduction :

Le déploiement adaptatif d'une application multi-composant sur le réseau consiste à placer les composants sur les nœuds suivant leurs contextes requis et fournis.

Plusieurs algorithmes ont été proposés dans la littérature pour le problème de placement des composants sur les nœuds comme GARA (*Globus Architecture for Reservation and Allocation*) [FOS00], PSF [IVA02], Conductor [REI00], Ninja [GRI01], CANS [XFU01], et *Sekitei* [TAT04]. Ce dernier regroupe les autres algorithmes en proposant un modèle global prenant en compte les différents aspects du problème et une méthode de résolution basée sur la planification. Dans ce chapitre, nous étudierons en détail l'algorithme *Sekitei*.

63.2.L'algorithme *Sekitei* :

6.6.4.Modèle du CPP (*Component Placement Problem*) :

Le problème du placement des composants sur les nœuds est défini par les éléments suivants : la topologie du réseau, l'application, le déploiement des composants, l'envoi des interfaces via le réseau et le but du CPP.

64. La topologie du réseau :

Elle est définie par l'ensemble des nœuds et des liens entre ces nœuds, chacun ayant des propriétés statiques et dynamiques.

Les propriétés dynamiques sont des propriétés qui peuvent être changées durant le cycle de vie de l'application et sont représentées par des valeurs réelles non négatives comme la bande passante et le CPU, contrairement aux propriétés statiques qui sont fixes durant le cycle de vie de l'application et peuvent être représentées par des variables booléennes ou des intervalles réels comme la sécurité du lien.

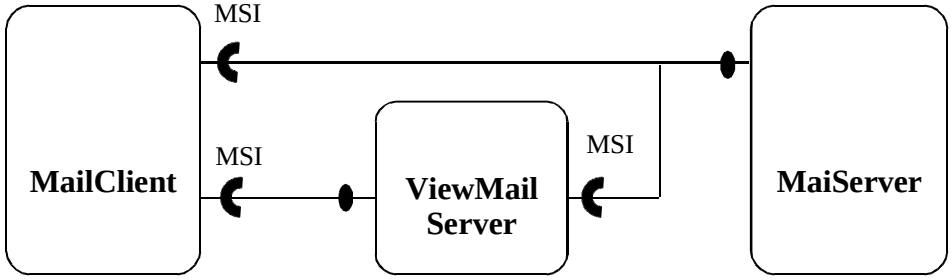
65. L'application :

Définie par l'ensemble des types de composants, chacun spécifie un ensemble d'interfaces requises et fournies qui sont caractérisées par un ensemble de propriétés.

L'exemple suivant présente une application de service de messagerie qui fournit les fonctionnalités suivantes : la gestion des comptes utilisateurs, des répertoires, des listes de contacts et la possibilité d'envoyer et recevoir des messages. En plus, elle permet à

l'utilisateur de spécifier un niveau de sécurité pour chaque message. Pour cela, elle comporte les composants suivants :

66. *MailServer(MS)* : gère les comptes clients.
67. *MailClient(MC)* : déployé sur le terminal utilisateur et gère ses opérations comme l'envoi d'un message.
68. *ViewMailServer(VMS)* : représente une copie de *MailServer*.
69. l'interface *MailServerInterface(MSI)* a été ajoutée pour que les composants puissent communiquer entre eux, comme présenté dans le schéma suivant :



L'exemple suivant, proposé dans [TAT04], présente une description du composant VMS qui spécifie :

70. Les propriétés de son interface fournie MSI^i et requise MSI^r comme le niveau maximum de sécurité du message (*Trust*), l'interface préserve ou non la sécurité du message (*sec*), le nombre de requêtes à traiter (*NumReq*), la taille maximale d'une réponse (*ReqSize*), le nombre de requêtes envoyées aux interfaces requises en réponse aux requêtes des interfaces fournies RRF(*Request Resuction Factor*), la capacité de CPU nécessaire pour le traitement d'une requête (*ReqCPU*), et le nombre maximum de requêtes pouvant être traitées par le composant (*MaxReq*).
71. Les besoins en ressources (*conditions*) qui doivent être satisfaits pour qu'il puisse être déployé sur un nœud donné.
72. Les effets (*Effects*) sur les différentes propriétés des nœuds et des interfaces après le déploiement.

<***Component name***= *VMS*>

<***Linkages***>

<*Implements*>

<*Interface* *name = MSI^r*>

<*Properties*>

MSI^i.Trust …….. fournie

MSI^i.Sec …….. fournie

MSI^i.NumReq …….. fournie

MSI^i.ReqSize …….. fournie

MSI^i.RRF := 10

MSI^i.ReqCPU := 2

MSI^i.MaxReq := 100

<*Requires*>

<*Interface name* = *MSI^r*>

<*Conditions*>

*Node.NodeCPU >= (MSI^i.NumReq * MSI^i.ReqCPU)*

*MSI^r.NumReq >= (MSI^i.NumReq * MSI^i.RRF)*

MSI^i.NumReq <= MSI^i.MaxReq

MSI^r.Sec = True

MSI^r.Trust >= 5

<*Effects*>

MSI^i.Sec := True

MSI^i.Trust := Node.Trust

MSI^i.ReqSize := 1000

MSI^i.NumReq := MIN(MSI^r.NumReq / MSI^i.RRF , MSI^i.MaxReq , Node.NodeCPU / MSI^i.ReqCPU)

*Node.NodeCPU := Node.NodeCPU – MSI^i.NumReq * MSI^i.ReqCPU*

<*Interface name* = *MSI*>

<*Crosslink*>

MSI^d.Sec := MSI^o.Sec AND Link.Sec

*Link.BW := Link.BW – MIN(Link.BW , MSI^o.NumReq * MSI^o.ReqSize)*

MSI^d.NumReq := MIN(MSI^o.NumReq , Link.BW / MSI^o.ReqSize)

MSI^d.ReqSize := MSI^o.ReqSize

VMS : ViewMailServer.

MSI : MailServerInterface.

r et i indiquent, respectivement, les interfaces requises et fournies (Implemented).

o et d indiquent les interfaces du lien d'origine et destination.

73. Le déploiement :

Pour déployer un composant sur un nœud donné, il faut que ses interfaces requises soient disponibles sur ce nœud et les contraintes sur ses propriétés et ses ressources (indiquées dans <*Conditions*>) soient satisfaites, dans l'exemple précédent, le composant VMS spécifie les conditions suivantes : le nœud doit être capable de traiter ses requêtes (*NumReq*), le nombre de requêtes (*NumReq*) ne doit pas dépasser un certain maximum, et le composant doit être capable d'envoyer RRF requêtes à ses interfaces requises.

Après le déploiement, les interfaces fournies par ce composant deviennent disponibles sur le nœud dont les propriétés dynamiques vont changer suivant les affectations indiquées dans <*Effects*>, dans l'exemple, la capacité de CPU diminuera et le nombre de requête (*NumReq*) sera recalculé.

74. L'envoi des interfaces via le réseau :

Afin de placer les composants sur les nœuds, les interfaces sont envoyées sur les liens du réseaux pour que l'on vérifie leurs satisfactions en ressources. Après l'envoi d'une interface sur un lien donné, leurs propriétés dynamiques (interface et lien) vont changer suivant les affectations spécifiées dans <*crosslink*>.

75. Le but de CPP :

L'algorithme *Sekitei* se base sur le but de CPP qui consiste à placer un composant donné sur un nœud bien déterminé. A partir de ce but, l'algorithme détermine le placement des autres composants. Par exemple, le but de CPP est de placer le composant *MailClient* (présenté précédemment) sur le nœud 0.

6.6.4.CPP comme un problème de planification :

Le problème du placement des composants sur les nœuds peut être considéré comme un problème de planification (*AI Planning Problem*) avec des contraintes sur les ressources :

76. L'état du système est défini par la disponibilité des interfaces et le placement des composants sur les nœuds. Ces informations sont représentées par des variables propositionnelles (booléennes).
77. Les propriétés des nœuds, liens et interfaces sont représentées par des variables de ressources qui prennent des valeurs réelles.
78. Le placement d'un composant sur un nœud donné et l'envoi d'une interface sur un lien sont exprimés par des opérateurs.
79. Le but de CPP est transformé en un but propositionnel.

La figure suivante présente la transformation d'un problème de placement des composants en un problème de planification (tâche réalisée par le compilateur), le planificateur génère ensuite un plan pour ce problème, qui sera utilisé par le décompilateur pour générer un plan de déploiement :

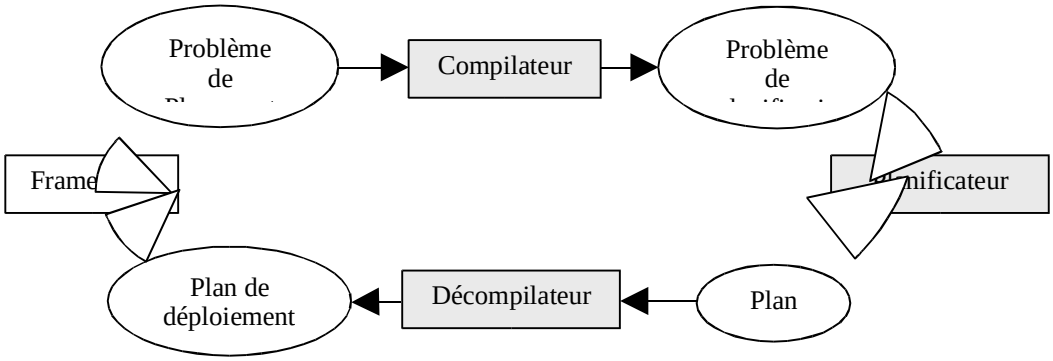


FIG. 5.1 – La transformation d'un problème de placement des composants en un problème de planification

6.6.4.La transformation :

L'état du système est défini par la topologie du réseau, l'existence des interfaces sur les nœuds et la disponibilité des ressources. Ces informations sont transformées par le compilateur en variables de ressources et de propositions.

Par exemple : aVMSI (0) représente la disponibilité de l'interface MSI sur le nœud 0, la variable de ressource CPU (1) indique la capacité de CPU sur le nœud 1.

La transformation d'un problème CPP en un problème de planification génère deux opérateurs *PL<Composant> (?n)* pour placer le composant sur le nœud *n*, et *CR<interface> (?n1, ?n2)* pour envoyer l'interface du nœud *n1* vers *n2*.

Le schéma d'un opérateur est défini par :

80. Précondition logique : représente l'ensemble des propositions qui doivent être vrais pour que l'opérateur puisse être exécuté (ligne 2).
81. Précondition sur les ressources : définie par des fonctions arbitraires qui retournent des valeurs booléennes (ligne 3-6).
82. Les effets logiques : représentent l'ensemble des propositions qui prennent la valeur vraie après l'exécution de l'opérateur (ligne 7).
83. Les effets sur les ressources : représentent l'ensemble des affectations de variables de ressources (ligne 8-12).
84. Le code suivant décrit l'opérateur de placement du composant *ViewMailServer*(*VMS*) sur le nœud *n*.

1 plVMS(?n : node)
2 PRE : avMSI(?n)

3	*cpu(?n) > MSIMaxReq*MSIReqCPU*
4	*numReq(MSI, ?n) > MSIMaxReq*MSIRRF*
5	*sec(MSI, ?n) = True*
6	*trust(MSI, ?n) > 5*
7 EEF :	*avMSI(?n), plVMS(?n)*
8	*numReq(MSI, ?n) := MIN(numReq(MSI, ?n) / MSIRRF, MSIMaxReq, cpu(?n) / MSIReqCPU)*
9	*cpu(?n) := cpu(?n) – numReq(MSI, ?n) * MSIRRF / MSIReqCPU*
10	*sec(MSI, ?n) := True*
11	*trust(MSI, ?n) := ntrust(?n)*
12	*reqSize(MSI, ?n) := 1000*

Le compilateur génère pour chaque type de composant (type d'interface) un schéma d'opérateur pour le placement de ce composant sur un nœud donné (l'envoi de cette interface d'un nœud *n1* vers *n2*).

6.6.4.L'algorithme *Sekitei* :

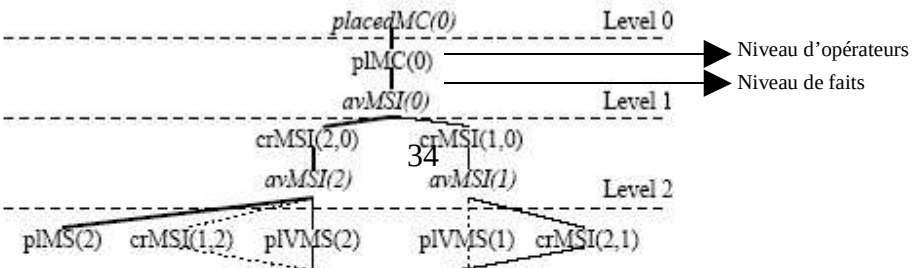
Le problème peut avoir des centaines de nœuds et des dizaines de composants qui seront transformés en opérateurs de placements de composants et d'envois d'interfaces, dont la plupart ne seront pas utilisés pour le résoudre. Pour cela l'algorithme détermine l'ensemble des opérateurs pertinents qui représentent ceux qui forment une séquence d'actions atteignant le but et élimine les opérateurs non pertinents en utilisant deux structures de données : RG (*Regression Graph*) et PG (*Progression Graph*).

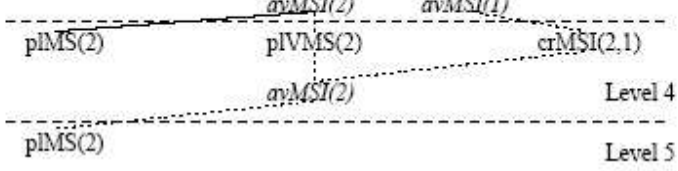
85. La phase de régression (RG) :

Détermine le plus petit ensemble d'opérateurs pertinents en prenant en compte leurs préconditions et effets logiques (les contraintes sur les ressources ne sont pas prises en compte).

Le graphe de régression (RG) contient plusieurs niveaux d'opérateurs et de faits définis de la façon suivante :

86. Le niveau de fait 0 représente le but.
87. Le niveau d'opérateur *i* contient tous les opérateurs réalisant certains faits du niveau *i-1*.
88. Le niveau de fait *i* contient toutes les préconditions logiques des opérateurs du niveau *i.*
89. Ce processus est répété jusqu'à ce que le but soit atteint. La figure suivante présente le graphe de régression du problème présenté précédemment, les lignes gras, normal et pointillée représentent, respectivement, les sous graphes possibles à 3, 4 et 5 étapes.





Sekitei *omposants sur les nœuds :*

FIG. 5.2 – Le graphe de régression

Nous pouvons décrire, par exemple, les niveaux 1 et 2 comme suit :

Pour placer le composant MC sur le nœud 0 (le but), il faut que l'interface MSI soit disponible sur ce nœud, pour cela on envoie MSI de 1 vers 0 ou de 2 vers 0. Mais pour envoyer MSI de 1 vers 0, il faut que cette interface soit disponible sur le nœud 1 …

90. La phase de progression (PG) :

Cette phase utilise l'ensemble des opérateurs pertinents déterminé par la phase de régression pour vérifier la possibilité d'atteindre le but du CPP en prenant en compte les contraintes sur les ressources et les propriétés, si celui-ci n'est pas atteint l'algorithme réexécute la phase de régression pour avoir plus d'opérateurs. Le graphe de progression contient aussi des niveaux d'opérateurs et de faits et des informations sur les relations d'exclusion mutuelle (*mutex*),

(c-à-d : le placement d'un composant sur un nœud donné peut empêcher le placement d'un autre composant sur le même nœud à cause des ressources non suffisantes dans la figure suivante une relation *mutex* est indiquée par des lignes pointillées). Le graphe de progression est construit de la façon suivante :

91. le niveau de fait 0 représente le but.
92. pour toutes les propositions du niveau *i-1*, un opérateur [*non-op*] est ajouté au niveau *i* ayant comme fait sa précondition et son effet logique, donc il ne consomme pas de ressources (cet opérateur est indiqué par des crochets dans le graphe).
93. pour chaque opérateur de la couche correspondante dans RG, un opérateur est ajouté à PG si aucune de ses préconditions n'est en exclusion mutuelle avec les propositions du niveau précédent.
94. deux opérateurs de même niveau son en *mutex* si :
    94.1. Certaines de leurs préconditions sont en *mutex*.
    94.2. Un opérateur change la variable d'une ressource utilisée dans les préconditions ou les effets de l'autre opérateur.

94.3.Leur consommation totale de ressources dépasse la valeur disponible.

95. deux faits d'un même niveau sont en *mutex* si tous les opérateurs qui peuvent les produire le sont aussi.

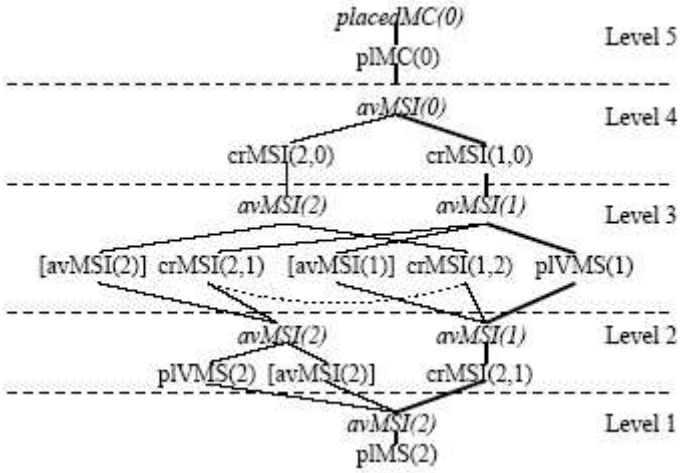


FIG. 5.3 - Le graphe de progression

6.6.4.Conclusion :

Sekitei propose un modèle global pouvant exprimer les différents éléments des modèles proposés dans les autres algorithmes, et une méthode basée sur la planification pour la résolution du problème de placement des composants sur les nœuds, mais il ne prend pas en compte tous les aspects d'adaptation des applications au contexte d'exécution où plusieurs implémentations doivent être associées à chaque composant et une étape du choix de ces implémentations est nécessaire pour s'adapter aux changements du contexte. Ces implémentations peuvent spécifier des préférences sur les ressources qui doivent être prises en compte pour mieux optimiser leurs placements sur les nœuds, et définir des poids sur les ressources indiquant l'importance qu'elles leurs accordent. Ces éléments ne sont pas pris en compte par l'algorithme *Sekitei* et leurs intégrations nécessitent le changement de certaines étapes. En plus la méthode de résolution proposée se base sur le but du CPP qui consiste à placer un composant donné sur un nœud bien déterminé, ce qui peut être considéré comme une restriction du problème si nous voulons concevoir un modèle ouvert où l'algorithme est le seul à décider du placement de tous les composants.

Pour cela, nous avons proposé un nouveau modèle qui prend en compte les différents aspects du problème d'adaptation en définissant une ontologie OWL permettant de décrire les ressources offertes par le réseau et celles requises par les implémentations, et un nouveau algorithme du choix et d'affectation des implémentations aux nœuds. Ce modèle et cet algorithme sont présentés dans le chapitre suivant.

Chapitre 6

Le choix du placement des composants sur les nœuds

Les applications multi-composants sensibles au contexte sont des applications qui s'adaptent aux changements du contexte d'exécution (l'ensemble des paramètres caractérisant l'environnement d'exécution comme les capacités du terminal utilisateur en terme de CPU et espace mémoire). Pour cela on dispose pour chaque composant d'un ensemble d'implémentations chacune étant conçue pour être déployée dans un contexte bien déterminé en spécifiant ses besoins en ressources. Par conséquent, l'adaptation consiste, essentiellement, à choisir parmi les implémentations de chaque composant celle qui sera déployée sur le réseau et placer les implémentations choisies sur les différents nœuds.
Dans ce chapitre, nous présentons la modélisation OWL des ressources fournies par le réseau et celles requises par les implémentations de composants, notre algorithme du choix du placement des composants sur les nœuds qui se base sur l'architecture présentée dans le paragraphe suivant et la réalisation de la proposition.

95.2.Architecture de déploiement adaptatif :

L'architecture dans laquelle va s'intégrer notre travail, proposée dans [AYD04], définit une plate-forme de déploiement adaptatif basée sur CCM, où chaque application est définie par l'ensemble des paquetages de composants, un descripteur d'assemblage et un descripteur du contexte auquel elle est sensible.
Les capacités limitées des terminaux font que ceux-ci ne peuvent pas toujours accueillir la totalité des composants de l'application, pour cela on a défini des serveurs d'exécution où les composants peuvent être instanciés.
Afin de compléter les fonctionnalités du CCM en lui permettant d'effectuer un déploiement adaptatif, [AYD04] a proposé d'ajouter un ensemble de modules permettant de détecter les changements de contexte pertinent, choisir les implémentations qui s'adaptent à ce contexte et les affecter aux différents nœuds du réseau.

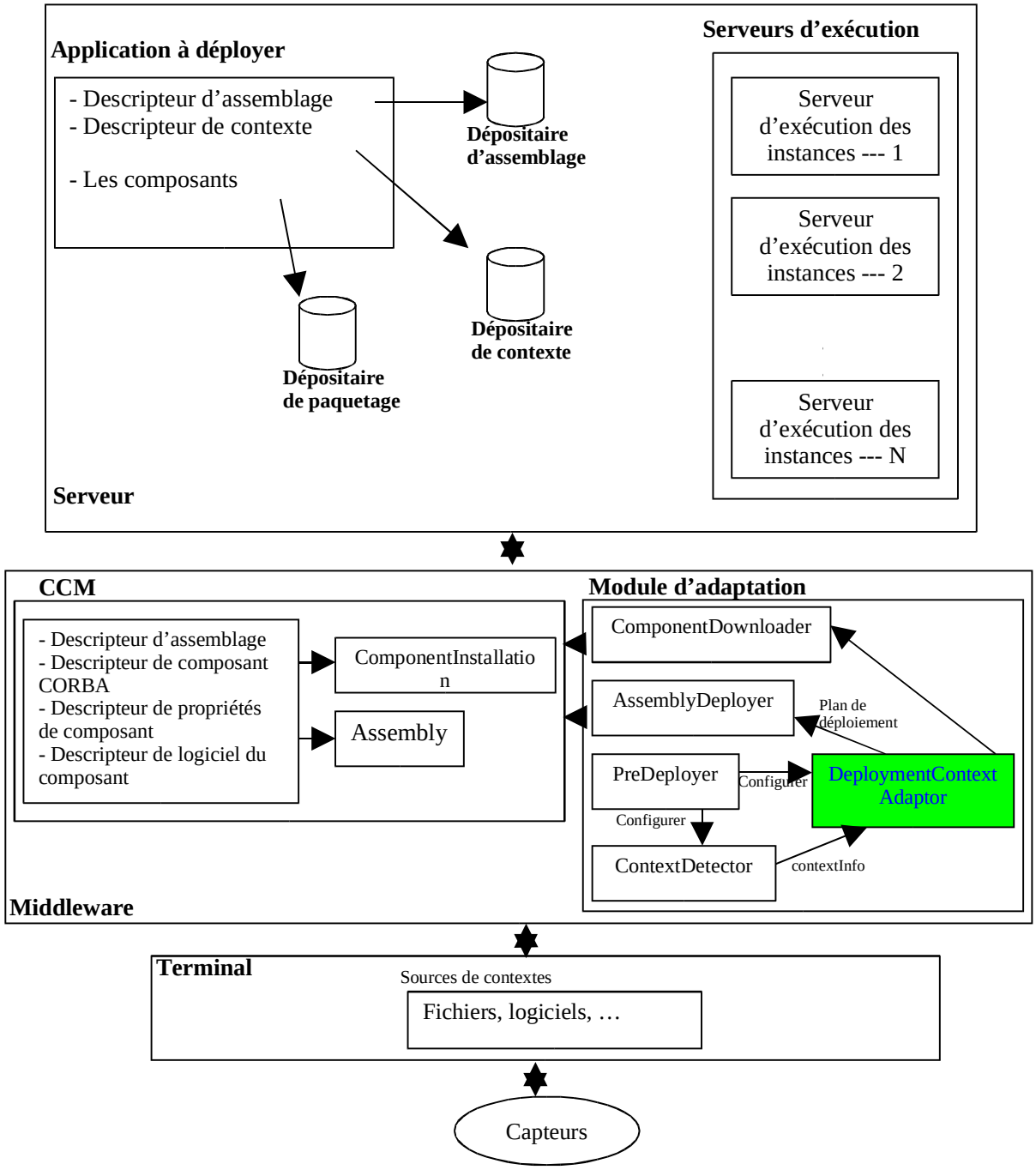
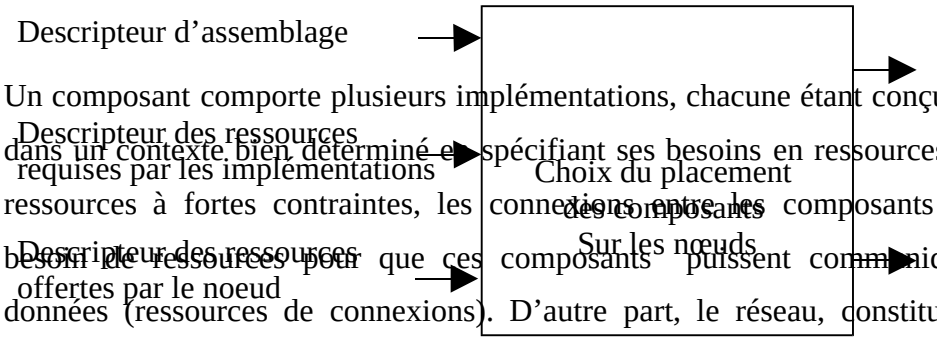


FIG. 6.1 – Architecture de déploiement adaptatif

ContextDetector détecte le contexte et le récupère à partir des sources de contextes (fichiers, logiciels, …), après filtrage, ce module envoie le message *ContextInfo* à *DeploymentContextAdaptor* qui détermine les implémentations à déployer et les nœuds d'instanciations, et génère un plan de déploiement qui sera projeté par CCM sur les sites physiques.

95.3.Architecture globale du choix de placement des composants sur les nœuds :

Notre travail se situe dans le module *DeploymentContextAdaptor* et consiste à choisir et affecter les implémentations aux nœuds en prenant en compte les ressources nécessaires à l'exécution des implémentations et celles offertes par les nœuds.



Un composant comporte plusieurs implémentations, chacune étant conçue pour être déployée dans un contexte bien déterminé en spécifiant ses besoins en ressources qui représentent les ressources à fortes contraintes, les connexions entre les composants peuvent aussi avoir besoin de ressources pour que ces composants puissent communiquer et échanger les données (ressources de connexions). D'autre part, le réseau, constitué d'un ensemble de nœuds et de liens entre les nœuds, offre un ensemble de ressources.

Les implémentations peuvent définir aussi des préférences sur les ressources (ressources de préférences) qui permettent de mieux affecter les implémentations aux nœuds, par exemple une implémentation peut être déployée sur un nœud qui n'a pas d'espace disque libre mais elle préfère, si possible, celui en ayant la plus grande valeur.

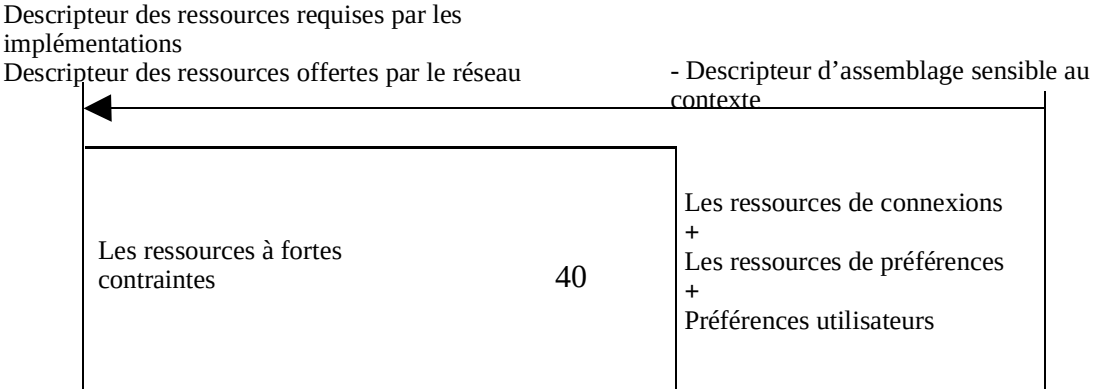
De même pour l'utilisateur qui peut définir des préférences, par exemple il préfère ne pas payer la connexion réseau, dans ce cas il faut placer les composants de sorte que leurs connexions ne consomment pas de ressources, une solution possible à cela est de les placer tous sur son terminal.

Le problème de placement des composants sur les nœuds consiste à choisir les implémentations à déployer et les nœuds d'instanciation en se basant sur les ressources à fortes contraintes et affecter ces implémentations aux nœuds tout en maximisant le nombre de préférences satisfaites et optimisant le partage de ressources.

Pour pouvoir déployer réellement ces implémentations sur le réseau, on doit générer un plan de déploiement de l'affectation choisie en prenant en compte le descripteur d'assemblage sensible au contexte.

Le schéma suivant en présente les étapes nécessaires :

-



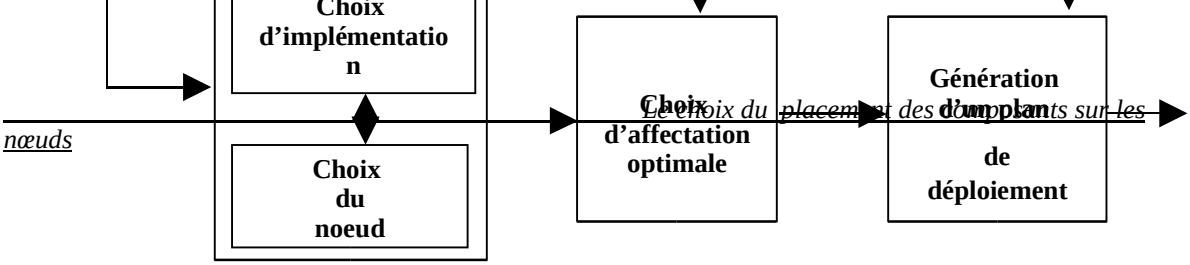


FIG. 6.2 – Architecture globale du choix du placement des composants sur les nœuds

95.4.Le modèle de ressource :

Pour pouvoir stocker, échanger et analyser les ressources offertes par le réseau et celles requises par les implémentations, nous devons d'abord modéliser ces ressources en utilisant un des langages de description présentés précédemment. OWL semble le mieux adapté pour la modélisation du contexte car il permet de concevoir des ontologies en spécifiant des classes et leurs relations et de modéliser la sémantique des ressources de sorte que l'on puisse déduire des nouvelles informations.

Dans notre algorithme du choix de placement des composants sur les nœuds, nous avons besoin de deux niveaux de modélisation : réseau et application.

6.6.4.Réseau :

Le réseau est un ensemble de nœuds et de liens entre ces nœuds. Ces composants peuvent offrir des ressources qui représentent le contexte d'exécution des applications.

Le schéma suivant présente la modélisation de ce contexte :

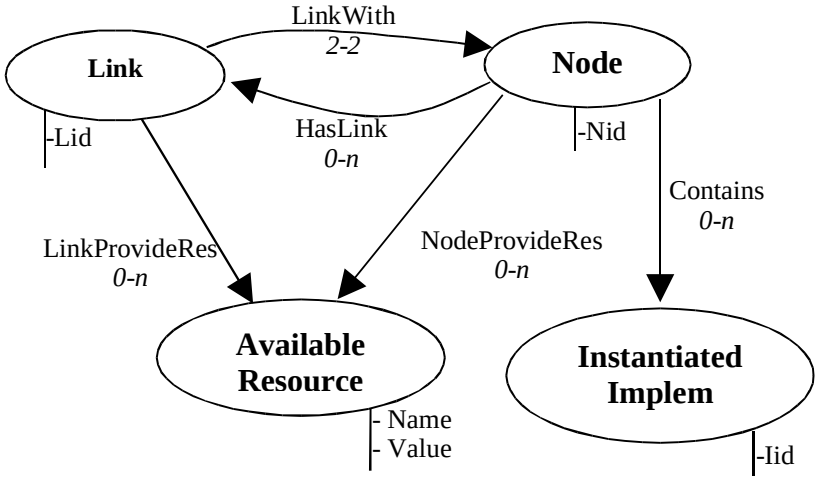


FIG. 6.3 – Le modèle des ressources offertes par le réseau

Chaque nœud est identifié par son *Nid* (*Node Identifier*), un lien est identifié par les deux *Nid* des nœuds le constituant, plusieurs liens peuvent exister entre deux nœuds par exemple un lien fibre optique et un lien coaxiale, pour cela nous avons ajoutés l'identifiant du lien *Lid* (*Link Identifier*).

Une ressource offerte par le nœud (le lien) est caractérisée par son nom comme *BW* pour la bande passante et sa valeur.

Afin de ne pas réinstancier les mêmes implémentations sur le même nœud, nous avons ajouté l'élément *InstantiatedImplem* à chaque nœud pour garder la trace de toutes les implémentations déployées sur ce nœud. Ces implémentations sont identifiées par leurs *Iid* (*Implementation Identifier*).

6.6.5.Application :

96. Composant :

Une application est un ensemble de composants identifiés par leurs *Cid* (*Component Identifier*) et interconnectés entre eux, chacun pouvant avoir plusieurs implémentations, chacune d'entre elles est conçue pour être déployée dans un contexte bien déterminé.

Une implémentation peut avoir besoin d'un certain nombre de ressources pour qu'elle puisse s'exécuter normalement, par conséquent, elle peut définir des contraintes sur ces ressources (l'attribut *Optional* de *Constraint* prend la valeur *False*) , on parle alors de ressources à fortes contraintes. Une implémentation peut définir aussi des préférences sur les ressources (l'attribut *Optional* de *Constraint* prend la valeur *True*) qui représentent les ressources de préférences. De même pour les connexions entre les composants qui peuvent définir des contraintes et des préférences sur les ressources.

Une affectation représente l'opération d'assignation des implémentations aux différents nœuds, elle peut créer une situation d'insuffisance de ressource où la ressource offerte par un nœud donné ne satisfait pas simultanément toutes les implémentations qui lui sont affectées. Les affectations valides sont toutes les affectations où les ressources offertes par les nœuds sont toutes suffisantes, pour ne pas pénaliser les implémentations dans le cas où toutes les affectations créent des situations d'insuffisances de ressources, nous essayons d'élargir l'ensemble des affectations valides de sorte que celles où les implémentations peuvent avoir un ordre d'exécution permettant de consommer les ressources à tour de rôle puissent en faire partie. Cet ordre est défini par le graphe de précédence qui permet de définir une relation de

précédence entre les composants de sorte que leurs implémentations déployées sur les nœuds n'utilisent pas simultanément les ressources, ce qui optimise l'utilisation de ces ressources. Pour cela nous avons ajouté pour chaque composant l'attribut *order* qui indique son ordre d'exécution, par conséquent les composants s'exécutant simultanément auront le même ordre.

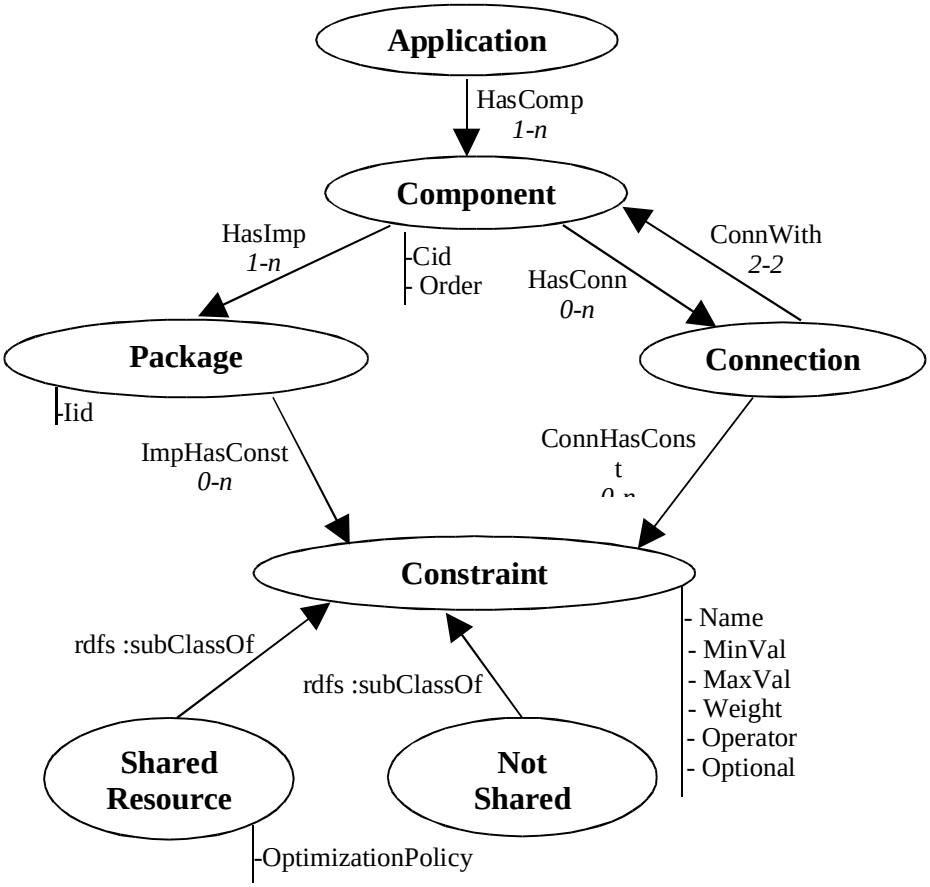


FIG. 6.4 – Le modèle des ressources requises par les implémentations

97. Ressources :

L'exécution simultanée des composants peut consommer plus de ressources que celle en série. Certaines ressources ne sont pas consommables comme la taille de l'écran et leurs valeurs ne changent pas pendant l'exécution, par conséquent nous pouvons différencier deux types de ressources : ressources partagées et non partagées.

Une ressource **partagée** est une ressource consommée par les implémentations comme la bande passante, auquel nous associons une politique d'optimisation (l'attribut *OptimizationPolicy*) définissant une fonction économique sur cette ressource, cette fonction est proportionnelle au sens de consommation de la ressource et peut être une fonction *max* pour le sens croissant comme la bande passante et *min* dans le cas contraire comme l'utilisation de CPU.

Dans notre algorithme, nous nous basons sur la notion de ressource **supplémentaire** qui représente la valeur au-delà du besoin minimum de l'implémentation, par exemple pour une

implémentation ayant besoin de plus de 100 Kbps de bande passante sachant que le réseau en offre 180 Kbps, la valeur supplémentaire est 180 – 100 = 80 Kbps.

La politique d'optimisation permet de calculer les ressources supplémentaires de sorte qu'elle soit : *ressource offerte – ressource requise* pour le cas de *max* et l'inverse pour *min*. Par exemple pour l'utilisation de CPU où la politique est *min*, supposont que l'implémentation a besoin de moins de 40% et le nœud offre 15%, donc la valeur supplémentaire sera 40-15=25.

Les implémentations (connexions) peuvent définir des contraintes et des préférences sur les ressources en spécifiant, pour chacune d'entre elles, l'intervalle de valeurs requis (les attributs *MinVal* et *MaxVal*) et le poids qui détermine l'importance que l'implémentation (connexion) lui accorde.

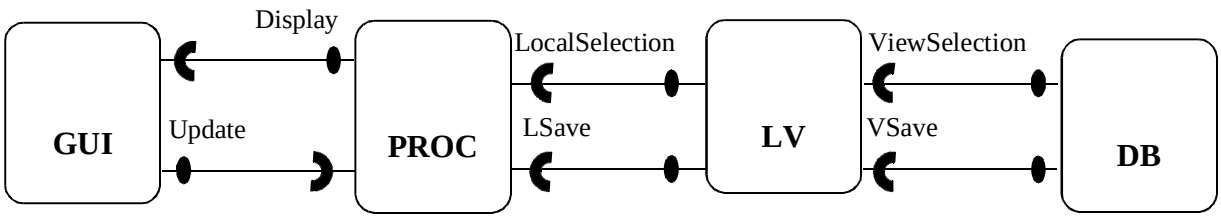
Les implémentations peuvent s'exécuter simultanément sur le même nœud et partager ses ressources offertes en fonction de leurs poids de sorte que l'implémentation ayant le plus grand poids d'une ressource donnée bénéficiera de la plus grande quantité supplémentaire de cette ressource. Par conséquent, ces poids doivent être normalisés et prennent leurs valeurs dans un intervalle déterminé, par exemple [0, 100].

Nous expliquerons en détail l'utilisation des poids pour le partage des ressources dans le paragraphe **5.7.2.b**.

Exemple :

Inspiré de [AYE04], cet exemple présente le déploiement d'une application ayant 4 composants : GUI : (***G**raphical **U**ser **I**nterface)*, PROC (Processing Component), LV : (***L**ocal **V**iew)*, et DB : (***D**ata **B**ase component*) sur le réseau, elle permet à l'utilisateur de consulter une base de données, faire des traitements sur ces données et sauvegarder les résultats. L'utilisateur peut travailler aussi en mode déconnecté en utilisant LV qui représente une vue partielle de la base de donnée.

L'assemblage des composants :



Les contraintes :

LV	USEDCPU(%)		FR_DISK_SP(GB)	
	Valeur	Poids	Valeur	Poids
L	<= 1O	1	>= 0.2	1

FR_DISK_SP : Free Disk Space.

GUI	SCREEN (pouces)		OS		Plugin	
	Valeur	Poids	Valeur	Poids	Valeur	Poids
G1	=3	1	EPOC	1	JAVA	1
G2	[14, 15]	1	UNIX	1	JAVA	1
G3	17	1	WIN2000	1	JAVA	1

PROC	USEDCPU(%)		FR_MEM_SP	
	Valeur	Poids	Valeur	Poids
P1	<= 10	3	>= 50	5
P2	<= 80	4	>= 200	1

FR_MEM_SP : Free Memory Space.

DB	FR_DISK_SP (GB)		FR_MEM_SP(MB)	
	Valeur	Poids	Valeur	Poids
D1	>= 1	5	>= 100	2
D 2	>= 5	7	>= 20	1

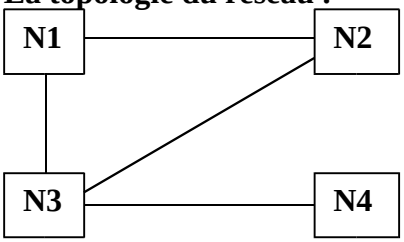
Les préférences :

PROC	CPU (GHZ)		FR_DISK_SP (GB)	
	Valeur	Poids	Valeur	Poids
P1	>= 1.8	2	>= 1	1
P2	>= 2	2	>= 0.5	1

Les connexions :

PROC-LV	BW (KBPS)		DIST (m)	
	Valeur	Poids	Valeur	Poids
P-L	>= 150	2	<= 200	1

La topologie du réseau :



Les ressources offertes par les nœuds :

	SCREEN	CPU	USEDCPU	FR_DISK_SP	FR_MEM_SP	OS	PLUGIN
N1	15	1	60	1	50	UNIX	/
N2	14	2	20	4	300	WIN2000	JAVA
N3	17	1.5	30	2	250	UNIX	/
N4	3	0.4	10	0.2	10	EPOC	/

Les ressources offertes par les connexions :

	BW	DIST
N1-N2	200	50
N1-N3	180	100
N2-N3	80	80
N3-N4	160	70

97.1.Formalisation du problème :

Le problème du placement des composants sur les nœuds consiste à choisir et affecter les implémentations aux nœuds d'instanciations, il peut se présenter sous deux dimensions :

98. Espace (réseau) : L'affectation des implémentations aux nœuds tout en satisfaisant :
    98.1.Les besoins de chaque implémentation en ressources.
    98.2.Les besoins de chaque connexion en ressources.
99. Temps : Le partage des ressources entre les implémentations dans le temps, donc les contraintes suivantes doivent être vérifiées :
    99.1.Satisfaire les besoins des implémentations pouvant partager les ressources offertes par les nœuds.
    99.2.Satisfaire les besoins des connexions pouvant partager les ressources offertes par les liens.

Pour mieux affecter les implémentations aux nœuds, nous devons optimiser l'affectation des ressources et prendre en compte leurs préférences ainsi que celles des connexions, donc les critères suivants doivent être pris en compte :

100.Maximiser le nombre préférence vérifiées pour les implémentations.

101.Maximiser le nombre de préférence vérifiées pour les connexions.

102.Maximiser le nombre de préférences utilisateur vérifiées.

103.Maximiser les ressources offertes par les nœuds.

104.Maximiser les ressources offertes par les liens entre les nœuds.

105.Ne pas réinstancier les implémentations déjà installées sur le même nœud.

Pour la co-localisation qui représente le placement de plusieurs composants sur le même nœud, nous avons laissé le choix au concepteur de spécifier les distances entre les composants

en ajoutant les contraintes appropriées pour les connexions correspondantes. Dans l'exemple précédent, pour co-localiser les composants PROC et LV, il suffit de spécifier la contrainte DIST=0 pour la connexion P-L.

Ce problème n'est pas NP-Complet dans le cas où tous les composants s'exécutent en série car il suffit d'affecter chaque implémentation (individuellement) au nœud vérifiant les critères précédents, de même dans le cas où toutes les ressources ne sont pas partagées.

Dans le cas contraire, il est équivalent au problème de *bin packing* qui consiste à mettre un ensemble d'éléments de tailles différentes dans des sacs de même capacité en utilisant le moins possible de sacs.

Nous prenons la plus petite taille du problème qui consiste à placer des implémentations ayant besoin d'une seule ressource sur des nœuds qui représentent les sacs dont les tailles sont les valeurs offertes. Les implémentations représentent les objets qui ont comme tailles les valeurs requises. Par conséquent, le problème de placement des composants sur les nœuds est NP-Complet.

105.1.La solution proposée :

Comme nous avons présenté dans le schéma FIG. 5.5, le placement des composants sur les nœuds se fait en deux étapes :

106.Le choix des implémentations et des nœuds qui consiste à sélectionner parmi les implémentations de chaque composant celle qui doit être déployée, et parmi les nœuds du réseau ceux qui vont être utilisés pour instancier ces implémentations et ce en se basant sur les ressources requises par les implémentations et celles offertes par les nœuds.

107.Après avoir choisi les implémentations et les nœuds, plusieurs affectations valides peuvent exister. Le choix d'affectation optimale consiste à sélectionner celle qui maximise le nombre de préférences satisfaites (implémentations, connexions et utilisateur) et optimise l'utilisation des ressources offertes par le réseau (nœuds et liens) de sorte que les implémentations et les connexions puissent en bénéficier de la quasi totalité, par conséquent, les ressources restées après le déploiement des implémentations doivent être maximales. Ces ressources représentent les ressources supplémentaires offertes par le réseau pour les implémentations et les connexions entre les composants.

La ressource **supplémentaire** offerte par un nœud (lien) pour une implémentation (connexion) donnée est la valeur au-delà de son besoin minimum offerte par ce nœud (lien).

La solution basique du choix d'affectation optimale est d'effectuer une recherche systématique sur l'ensemble des affectations valides en prenant en compte :

108.Les ressources de préférences des implémentations et des connexions.

109.Les préférences utilisateurs.

110.Les ressources supplémentaires (implémentations et connexions).

Cette solution est coûteuse en temps de traitement et espace mémoire car elle peut engendrer une explosion combinatoire. Pour cette raison, nous avons opté pour d'autres méthodes d'optimisations. Les heuristiques sont des méthodes de recherche puissantes et plus utilisées dans le domaine de l'optimisation combinatoire. Due à la complexité de ce problème où plusieurs critères d'optimisations doivent être pris en compte, nous nous basons sur une recherche heuristique permettant de trouver une affectation dans un temps raisonnable, pour cela nous devons d'abord trouver les métriques qui permettent de mesurer les critères du choix d'affectation cités précédemment.

Pour mieux gérer ces critères, nous essayons de minimiser le nombre de métriques en regroupant ceux qui sont de mêmes natures en un seul critère tout en gardant leurs influences sur le choix d'affectation de la façon suivante :

111.Maximiser le nombre de préférences satisfaites pour les implémentations, les connexions et l'utilisateur : en prenant en compte leurs poids, donc ça revient à maximiser **le pourcentage de préférences** satisfaites.

112.Maximiser les ressources supplémentaires offertes par les nœuds, et les liens entre les nœuds : c-à-d les ressources supplémentaires offertes par le réseau, ça revient à maximiser la **moyenne** des ressources supplémentaires offertes par le réseau.

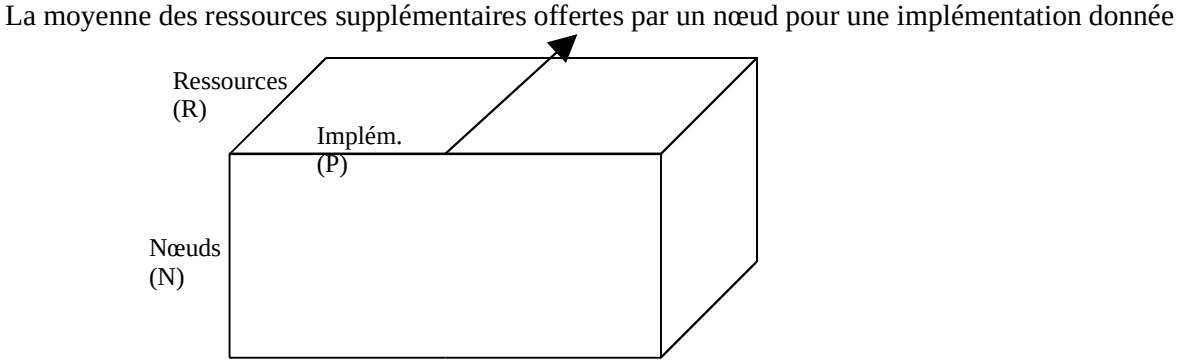
Donc cette heuristique permet de maximiser le pourcentage de préférence et la moyenne des ressources supplémentaires offertes par le réseau.

Comme nous avons expliqué précédemment, le placement des composants commence d'abord par le choix des implémentations et des nœuds. Pour minimiser le temps de traitement, nous voulons que cette étape soit une phase préparatoire pour la deuxième étape (l'optimisation) de sorte que cette dernière puisse réutiliser ses résultats de calcul. Pour cela, nous nous basons sur les ressources supplémentaires de sorte que les implémentations n'ayant aucune ressource supplémentaire sur tous les nœuds vont être éliminées de la façon présentée dans la section suivante.

6.6.4.Le choix des implémentations et des nœuds :

Un type de composant est réalisé par un ensemble d'implémentations chacune d'entre elles étant adaptée à un certain type de contexte d'exécution. Le choix des implémentations consiste à sélectionner pour chaque composant l'implémentation qui sera déployée sur le

réseau. Cette sélection dépend essentiellement de la disponibilité d'au moins un nœud pouvant lui offrir les ressources dont elle a besoin (c-à-d toutes ses contraintes sur les ressources doivent être satisfaites). La validité de la sélection peut être vérifiée en se basant sur les ressources supplémentaires offertes par les nœuds qui peuvent être présentées sous forme du tableau 3D suivant :



Le calcul de la ressource supplémentaire dépend de la politique d'optimisation et peut être : *ressource offerte - ressource requise* pour une politique *max*, et *ressource requise - ressource offerte* pour *min*.

Prenons le triplet (**I**, **N, R**) qui indique la case du tableau correspondante à la ressource **R** offerte par le nœud **N** pour l'implémentation **I**, la ressource supplémentaire correspondante est obtenue de la façon suivante :

113.**R** est partagée :

113.1.La contrainte de **I** sur **R** est vérifiée : appliquer la formule précédente.

113.2.La contrainte de **I** sur **R** n'est pas vérifiée : -1.

114.**R** n'est pas partagée :

114.1.La contrainte de **I** sur **R** est vérifiée : 0.

114.2.La contrainte de **I** sur **R** n'est pas vérifiée : -1.

donc une implémentation est sélectionnée s'il existe au moins un nœud où toutes les ressources supplémentaires offertes pour cette implémentation sont supérieures ou égales à zéro.

Pour mieux analyser le tableau et diminuer le temps de traitement en préparant la deuxième étape, il vaut mieux le réduire en un tableau 2D en calculant les moyennes des ressources supplémentaires pour chaque couple (implémentation : **I**, nœud : **N**). La formule de calcul des moyennes des ressources supplémentaires est présentée dans le paragraphe **5.7.2.b**. Ce tableau est construit de la façon suivante :

115. Toutes les contraintes de **I** sont vérifiées : on met la moyenne des ressources supplémentaires.

116. Au moins une contrainte n'est pas vérifiée : on met –1.

donc une implémentation est sélectionnée s'il existe au moins un nœud lui offrant une moyenne supérieur ou égale à zéro, de la même façon un nœud est sélectionné s'il existe au moins une implémentation pour laquelle il offre une moyenne positive ou nulle, comme présenté dans l'exemple suivant :

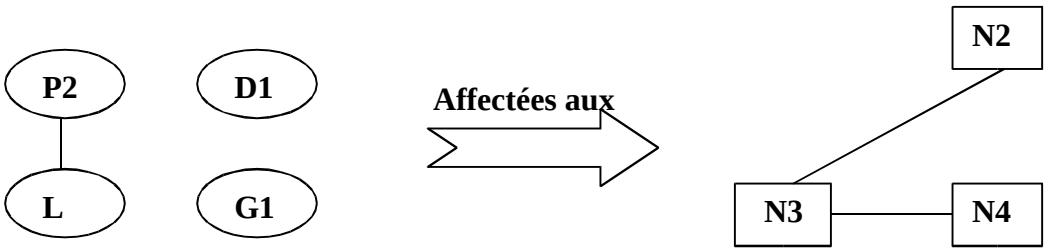
Le tableau des moyennes des ressources supplémentaires pour notre exemple :

	L	G1	G2	G3	P1	P2	D1	D2
N1	-1	-1	-1	-1	-1	-1	-1	-1
N2	-1	-1	-1	-1	-1	68	59.28	-1
N3	-1	-1	-1	-1	-1	50	43.57	-1
N4	0	0	-1	-1	-1	-1	-1	-1

Ce tableau sera réduit une deuxième fois en éliminant certaines implémentations et certains nœuds. Donc les implémentations et les nœuds sélectionnés pour notre exemple sont :

	L	G1	P2	D1
N2	-1	-1	68	59.28
N3	-1	-1	50	43.57
N4	0	0	-1	-1

Après avoir réduit l'espace de recherche en sélectionnant les implémentations à déployer et les nœuds d'instanciations, nous affectons ces implémentations à ces nœuds en prenant en compte les différents critères d'optimisations cités précédemment.



Application Réseau

6.6.4.L'affectation des implémentations aux nœuds :

Comme nous avons expliqué précédemment, l'affectation des implémentations aux nœuds s'effectue suivant ces deux critères d'optimisations :

117.Maximiser le pourcentage de préférence.

118.Maximiser la moyenne des ressources supplémentaires.

Pour pouvoir gérer ces deux critères, nous devons trouver les métriques qui permettent de les mesurer.

119.Le pourcentage de préférence :

Les préférences représentent l'ensemble de toutes les ressources de préférences des implémentations choisies et des connexions entre les composants ainsi que les préférences utilisateurs. Chaque préférence a un poids indiquant son importance par rapport aux autres préférences.

Le pourcentage de préférence est le rapport entre le nombre de préférences satisfaites sur le nombre de toutes les préférences et ce en prenant en compte leurs poids, il est donné par la formule suivante :

$$P(\mathrm{D,E}) = \frac{\sum_{i=1}^{card(E \cup C \cup U)} \sum_{j=1}^{Nb''Pref} P_{ij} * Verifie(i,j)}{\sum_{k=1}^{card(E \cup U)} \sum_{l=1}^{Nb''Pref} P_{kl}}$$

D : l'ensemble des nœuds.

E : l'ensemble des implémentations.

C : l'ensemble des connexions.

U : l'utilisateur

P (D,E) : Le pourcentage de préférence.

P_{ij} : Le poids de la préférence j défini par l'implémentation (connexion ou l'utilisateur) i.

NbPref : Le nombre de préférences.

Verifie(i,j) = 1 si la préférence est satisfaite.

Verifie(i,j) = 0 si la préférence n'est pas satisfaite.

120.La moyenne des ressources supplémentaires :

Les ressources supplémentaires offertes par le réseau représentent les ressources supplémentaires offertes par les nœuds et celles offertes par leurs liens.

Plusieurs implémentations peuvent s'exécuter simultanément sur le même nœud et partagent ainsi ses ressources supplémentaires suivant les poids qu'elles leurs accordent.

Supposons que les implémentations I1 et I2 s'exécutent en parallèle sur le nœud N, et ont besoin, respectivement, de plus de 100 Kbps et 140 Kbps de bande passante dont N offre 300 Kbps. Les poids de cette ressource sont, respectivement, 6 et 4 pour I1 et I2.

La ressource supplémentaire offerte par N pour (I1 et I2) est : 300- (100+140) = 60 dont I1 va bénéficier de : (6*60) / (6+4) = 36 et I2 de : (4*60) / (6+4) = 24.

Pour placer ces deux implémentations en commençant par exemple par I1, nous devons prendre en compte I2. Par conséquent, les implémentations s'exécutant en parallèle doivent être placées ensemble, pour cela nous les regroupons dans une seule entité qui peut être considérée comme une nouvelle implémentation dont les contraintes et les préférences sont l'union de celles des implémentations la constituant. Nous appelons cette entité **granule**.

De même pour les connexions entre les composants qui peuvent partager les ressources offertes par les liens entre les nœuds.

Nous définissons ainsi le granule d'implémentation (connexion) comme étant l'ensemble des implémentations (connexions) s'exécutant simultanément sur le même nœud (partageant le même lien), il représente l'unité de placement des implémentations (connexion) et peut être une seule implémentation (connexion) ou un ensemble d'implémentations (connexions).

Dans l'exemple précédent, le granule sera défini par la contrainte : bande passante >= 240.

La moyenne des ressources supplémentaires offertes par un nœud (lien) donné pour un granule est obtenue en calculant celles de ses implémentations (connexions) tout en prenant en compte le partage des ressources de la façon suivante :

$$M_{\text{supp}}(\text{N,I}) = \frac{\sum_{j=1}^{NbResPart} P_j * R_{\text{sup}\,p}(R_j, N, I)}{\sum_{l=1}^{NbResPart} P_l} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

M_{supp}(N,I) : La moyenne des ressources supplémentaires offertes par le nœud (lien) N à l'implémentation (connexion) I.

NbResPart : Le nombre de ressources partagées. pour les ressources non partagées la ressources supplémentaires = 0.

P_j : Le poids de la ressource R_j défini par l'implémentation I (connexion).

R_{supp}(R_j,N,I) : La valeur de la ressource supplémentaire R_j offerte par le nœud (lien) N à l'implémentation (connexion) I qui est donnée par la formule suivante :

$$R_{supp}(R_j, N, I) = \frac{P_j * R_{supp}(R_j, N, G)}{\sum_{k=1}^{m} P_{kj}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

m : Le nombre d'implémentations (connexions) partageant la ressource R_j.

P_{kj} : Le poids de la ressource R_j défini par l'implémentation (connexion) I_k.

$R_{supp}(R_j,N,G)$: La valeur de la ressource supplémentaire R_j offerte par le nœud (lien) N au granule G qui contient l'implémentation I (connexion), donnée par la formule suivante :

$$R_{supp}(R_j, N, G) = R_{off} - \sum^{G} R_{min} \quad \text{Pour la politique } max \ldots\ldots\ldots\ldots\ldots\ldots (3)$$

$$R_{supp}(R_j, N, G) = \sum^{G} R_{max} - R_{off} \quad \text{Pour la politique } min. \ldots\ldots\ldots\ldots\ldots\ldots (4)$$

Roff : La ressource offerte par N.

Rmin : La valeur minimale de la ressource requise par G.

Rmax : La valeur maximale de la ressource requise par G.

La formule suivante donne la moyenne des ressources supplémentaires offertes par un ensemble de nœuds (liens) D pour un ensemble d'implémentations (connexions) E en calculant, pour chaque nœud (lien), les moyennes des ressources supplémentaires offertes pour les implémentations (connexions) qui lui sont affectées :

$$M_{supp}(D, E) = \frac{\left[\sum_{k=1}^{card(D)} \sum_{j=1}^{card(E_k)} M\sup p(N_k, I_j)\right]}{card(E)} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5)$$

Où :

D : l'ensemble des nœuds (liens).

E : l'ensemble des implémentations (connexions).

Msupp(D,E) : La moyenne des ressources supplémentaires offertes par D pour les E.

E_k : l'ensemble des implémentations (connexions) affectées au nœud N_k.

$Msupp(N_k,I_j)$: La moyenne des ressources supplémentaires offertes par le nœud (lien) N_k à l'implémentation (connexion) I_j donnée par la formule (1).

Donc :

$$M_{supp}(D,E) = \left[\sum_{k=1}^{card(D)} \sum_{j=1}^{card(E_k)} \frac{\sum_{l=1}^{NbResPart} PG_l * R_{supp}(R_l, N_k, G_j)}{\sum_{i=1}^{NbResPart} P_i} \right] ¿\, card(E) \quad \ldots\ldots\ldots\ldots\ldots\ldots (6)$$

Où :

$$PG_l = \frac{p_l^2}{\sum_{i=1}^{m} p_i}$$

P_l : Le poids de la ressource R_l pour l'implem. (connexion) j.

m : Le nombre d'implémentations (connexions) partageant la ressource R_l.

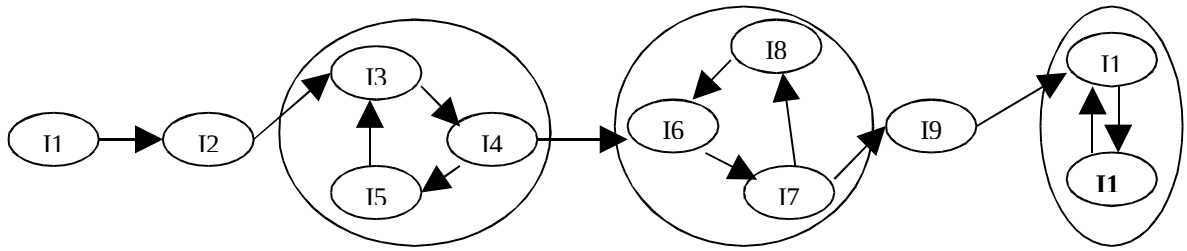
Rsupp(Rl,Nk, Gj) : La ressource supplémentaire Rl offerte par Nk pour le granule Gj, et qui est donnée par les formules donnée par les formules (3) et (4).

121.L'algorithme :

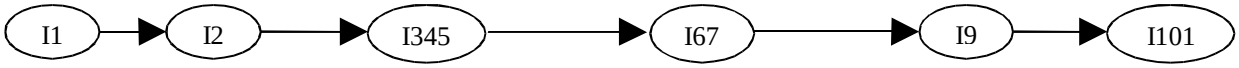
Les implémentations s'exécutant en série ne partagent pas de ressource, par conséquent cet algorithme peut être appliqué parallèlement sur les ensembles des implémentations pouvant s'exécuter simultanément, ce qui réduit considérablement l'espace de recherche :

Exemple :

Le graphe de précédence :



Le graphe réduit :



L'algorithme sera appliqué indépendamment sur :

(I1), (I2), (I3, I4, I5), (I6, I7, I8), (I9), (I10, I11).

Dans ce rapport nous supposons que les besoins des connexions en ressources sont toutes satisfaites et nous nous intéressons aux ressources supplémentaires offertes par les nœuds.

L'affectation des implémentations s'effectue suivant le pourcentage de préférences et la moyenne des ressources supplémentaires. La fonction d'évaluation sera établie suivant le choix du déployeur qui peut favoriser un critère sur l'autre, dans le cas contraire elle sera sous la forme suivante :

122.

$$\mathbf{f} = \begin{cases} P \text{ si l'implémentation a des préférences.} \\ M_{supp} \text{ si l'implémentation n'a pas de préférence} \\ \quad \text{ou} \\ \quad \text{Plusieurs implémentations ont le même P.} \end{cases}$$

L'algorithme, basé sur A* [DRE03] consiste à explorer l'espace de recherche en passant d'une configuration (affectation) à une autre et en essayant à chaque fois de se rapprocher de l'affectation optimale en choisissant celle où f est maximale (ligne 11).
Pour chaque itération (ligne 8) où certaines implémentations ont été déjà placées, une configuration (affectation) (ligne 10) est obtenue en affectant les autres implémentations (E) aux nœuds qui leurs donnent un f maximal (estimation basée sur les résultats de la première étape) (ligne 9).
Les itérations de (2) peuvent s'exécuter simultanément, pour cela nous leur assoçions des *Threads* (dans la réalisation de cet algorithme).

Cet algorithme est présenté comme suivant :
1. Placer les implémentations s'exécutant en série en choisissant le nœud où f est maximale (minimiser l'espace de recherche).
2. Pour chaque ensemble d'implémentations pouvant s'exécuter en parallèle faire :
3. f = 0

4. -Appliquer A^* :
5. *IMP1* = n'importe quelle implémentation choisie dans la première étape.
6. Tant que au moins une implémentation n'est pas encore placée faire :
7. *IMPL_PLACER* = *IMP1* ou l'implémentation suivante de la configuration sélectionnée (l'implémentation qui n'est pas encore placée)
8. Pour chaque nœud (où l'implémentation *IMPL_PLACER* pourra être déployée) faire :
9. E = estimation (placer les implémentations qui ne sont pas encore placées sur les nœuds où leurs f est max. et Msupp != -1).
10. f = f (implémentations placées + E).
11. Sélectionner l'affectation (configuration) où f est max.
12. Fin.

Exemple :

Les implémentations L et G peuvent être déployées sur un seul nœud (N4), par contre P2 et D1 peuvent être placées sur les deux nœuds N2 et N4 qui leur offrent les ressources suivantes:

P2 :

	P	M
N2	1	68
N3	0.33	50

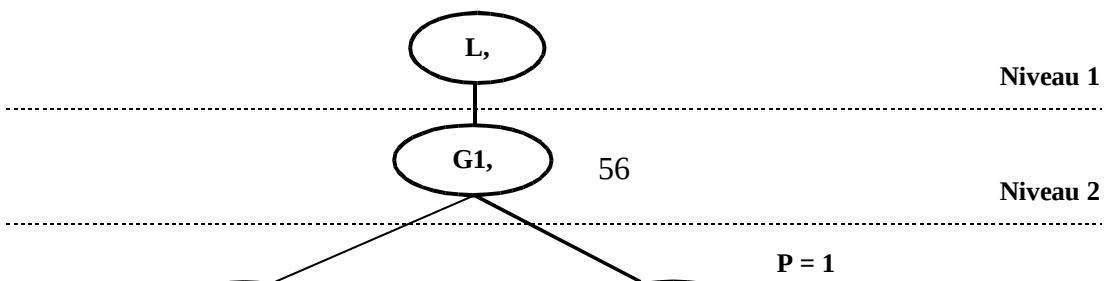
D1 :

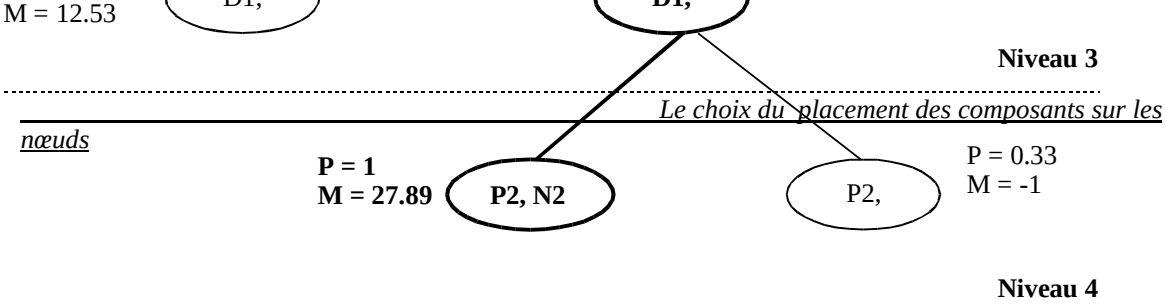
	M
N2	59.28
N3	43.57

Donc N2 est le meilleur nœud pour les deux implémentations car il offre un pourcentage de préférence (P=1) pour P2 supérieur à celui de N3 (P=0.33), et une moyenne des ressources supplémentaires (M=59.28) pour D1 supérieure à celle offerte par N3 (M=43.57).

Le schéma suivant présente l'arbre de recherche d'affectation optimale.

Dans le niveau 3 où L et G ont été déjà affectées, nous voulons placer l'implémentation D1 qui peut être déployée sur N2 ou N3. Pour compléter les deux configurations, nous affectons (estimation) P2 au nœud qui lui offre un f maximal (N2). Pour le niveau 4 nous prenons la meilleure configuration parmi les deux configurations précédentes et nous appliquons le même processus sur l'implémentation P2 :

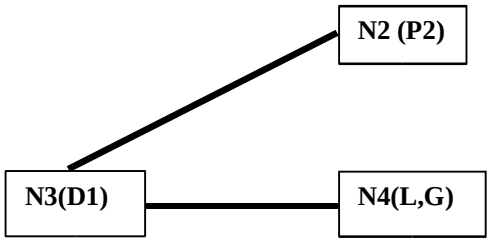




123.Le plan de déploiement :

Après avoir choisi l'affectation optimale, l'algorithme génère un plan de déploiement qui indique que telle implémentation sera déployée sur tel nœud, ce plan sera projeté par le modèle de composant sur les sites physiques.

La solution du plan de déploiement obtenu sur l'exemple précédent est donné par le schéma suivant :



123.1.La réalisation:

Afin d'évaluer l'algorithme en déterminant les différents paramètres à savoir : la montée en charge et le temps de traitement, nous avons mis en œuvre cette proposition en utilisant comme langage de description OWL qui devient de plus en plus le langage maître de la modélisation du contexte par sa capacité de décrire toute sorte d'information, et comme langage de programmation JAVA.

Notre application se compose de trois parties : modélisation, base de données objet et traitement, comme présenté dans le schéma suivant :

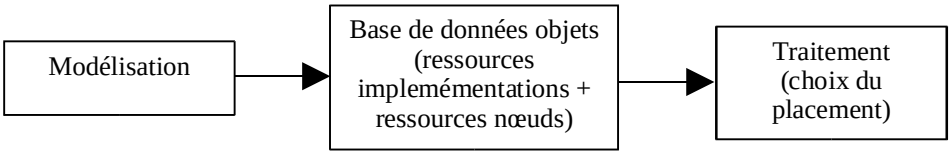


FIG. 6.5 – Les modules de l'applications

6.6.4.Modélisation :

Les ressources offertes par le réseau et celles requises par les implémentations doivent être modélisées pour que l'on puisse les échanger et les analyser. Dans le chapitre précédent nous avons défini les modèles correspondants qui permettent de définir deux schéma OWL : Réseau et application dont une instance représente une description des ressources, le code suivant présente un exemple de ces descriptions :

```
<componentinstance Cid="PROC" ord="1">
  <componentfileref idref="fileref2"/>

  <componentimplementations>

    <implementation  Iid="p1">
                <sharedresourceconstraint resourcename="usedcpu" weight="3" optimizationpolicy="toMinimize">
                        <operator>
                                <lowerorequal>
                                        <value>10</value>
                        </lowerorequal>
                        </operator>
                </sharedresourceconstraint>
                <sharedresourceconstraint resourcename="fr_mem_sp" weight="5" optimizationpolicy="toMaximize">
                        <operator>
                                <greaterorequal>
                                        <value>50</value>
                        </greaterorequal>
                        </operator>
                </sharedresourceconstraint>
                <notsharedresourceconstraint resourcename="cpu" weight="2" optional="true">
                        <operator>
                                <greaterorequal>
                                        <value>1.8</value>
                                </greaterorequal>
                        </operator>
                </notsharedresourceconstraint>
<sharedresourceconstraint resourcename="disk" weight="1" optional="true">
```

```
                    <operator>
                        <greaterorequal>
                            <value>1</value>
                    </greaterorequal>
                    </operator>
                </sharedresourceconstraint>
        </implementation>

    <implementation Iid="p2">
                <sharedresourceconstraint resourcename="usedcpu" weight="4" optimizationpolicy="toMinimize">
                    <operator>
                        <lowerorequal>
                            <value>80</value>
                    </lowerorequal>
                    </operator>
                </sharedresourceconstraint>
                <sharedresourceconstraint resourcename="fr_mem_sp" weight="1" optimizationpolicy="toMaximize">
                    <operator>
                        <greaterorequal>
                            <value>200</value>
                        </greaterorequal>
                    </operator>
                </sharedresourceconstraint>
                <notsharedresourceconstraint resourcename="cpu" weight="2" optional="true">
                    <operator>
                        <greaterorequal>
                            <value>2</value>
                    </greaterorequal>
                    </operator>
                </notsharedresourceconstraint>
                <sharedresourceconstraint resourcename="disk" weight="1" optional="true">
                    <operator>
                        <greaterorequal>
                            <value>0.5</value>
                        </greaterorequal>
                    </operator>
                </sharedresourceconstraint>
        </implementation>

  </componentimplementations>
        …
```

La description suivante présente les ressources offertes par les nœuds dont les valeurs varient dans le temps :

```
<nodesdescriptor>
        <nodes>
                <node Nid="n1">
                    <resource name="screen" value="15"/>
                    <resource name="cpu" value="1"/>
                    <resource name="usedcpu" value="60"/>
                    <resource name="fr_disk_sp" value="1"/>
                    <resource name="fr_mem_sp" value="50"/>
                    <resource name=”OS” value="UNIX"/>
                </node>
                <node Nid="n2">
                    <resource name="screen" value="14"/>
                    <resource name="cpu" value="2"/>
                    <resource name="usedcpu" value="20"/>
                    <resource name="fr_disk_sp" value="4"/>
                    <resource name="fr_mem_sp" value="300"/>
                    <resource name=”OS" value="WIN2000"/>
                    <resource name=plugin" value="JAVA"/>
            </node>
        …
</nodes>
```

6.6.5.Base de données objet :

Pour placer les composants sur les nœuds, l'algorithme peut avoir besoin d'une ressource à plusieurs reprises et récupère sa valeur autant de fois que nécessaire. Les accès aux ressources qui sont stockées sur le disque peuvent ralentir considérablement le traitement et augmentent ainsi le temps de réponse. Pour cela nous avons crée une base de données objet constituée principalement de tables d'hachages chacune d'entre elles contenant les objets d'un type de ressources, par conséquent trois tables sont définies :

124. Contrainte : contient les objets de la classe *Implementation* qui définissent les ressources à fortes contraintes des implémentations correspondantes dont les identifiants (Iid) représentent les clés de ces objets.
125. Préférence : La même chose que la table *Contrainte* en définissant les ressources de préférences des implémentations.
126. Nœud : stocke les objets de la classe *Node* qui définissent les ressources offertes par les nœuds correspondants dont les identifiants (Nid) sont les clés de ces objets.

L'accès aux objets de la base de données s'effectue suivant leurs clés, ce qui diminue énormément le temps de réponse, pour cela nous avons définis pour chaque table les méthodes d'accès permettant d'ajouter, récupérer et supprimer des contraintes, préférences ou des nœuds.

Pour cette raison nous avons crée les classes suivantes :

127. *resource* est la classe de base qui définit une ressource en spécifiant son nom, son poids l'intervalle de valeurs dont l'implémentation a besoin, l'opérateur qui peut être *Greaterorequal*, *Lowerorequal* ou *Between*, partagée ou non et sa politique d'optimisation qui peut être *ToMaximize* ou *ToMinimize*, comme présenté dans le code suivant :

```
public class Resource {

        String name=null;               /*      Le nom de la ressource                          */
        String op=null;                 /*      L'operateur                                     */
        String valmin=null;             /*      La valeur minimale                              */
        String valmax=null;             /*      La valeur maximale                              */
        int weight=0;                   /*      le poids de la contrainte ou de la preference   */
        boolean shared=true;            /*      partagee ou non partagee                        */
        String policy=null;             /*      la politique d'optimisation : ToMaximize,ToMinimize  */

        /*----------------------  constructeur pour les ressources offertes par les nœuds ----------------------------*/

        public Ressource(String nameC, String valueC){
                this.name=nameC;
                this.valmin=valueC;
        }
        /*------------------------------ constructeur pour les ressources requisespar ----------------------------------*/

        public Ressource(String nameC, String opC, String valminC, String valmaxC, int weightC, boolean sharedC, String policyC) {
                this.name=nameC;
                this.op=opC;
                this.valmin=valminC;
```

```
                this.valmax=valmaxC;
                this.weight=weightC;
                this.shared=sharedC;
                this.policy=policyC;
        }
]
```

128.La classe *Implementation* définit une implémentation en spécifiant ses ressources à fortes contraintes (de préférences) stockées dans la table d'hachage *RessImpl*, ce qui accélère l'accès à ces ressources, le code suivant en présente les principales propriétés :

```
public class Implementation {
        public Hashtable RessImplem = new Hashtable();   /*  contient des objets de type Resource pour spécifier les
                                                            contraintes (préférences) de l'implémentation  */

        String id;                                       /*  L'identifiant de l'implemantation  */
        String idComp;                                   /*  L'identifiant du composant  */
        …
}
```

129.La classe *Node* qui implémente *Thread* permet de créer pour chaque nœud le processus qui récupère ses ressources et calcule ses moyennes de ressources supplémentaires, ces processus peuvent s'exécuter en parallèle ce qui améliore énormément le temps de traitement, le code suivant présente les propriétés de cette classe :

```
public class Node extends Thread {
        public Hashtable RessNode;      /* contient des objets de type Resource pour spécifier les ressources offertes par le noeud */

        String id;                      /* L'identifiant du noeud  */

        public Hashtable MoyenRess;     /*le tableau des moyennes des ressources supplementaires offertes par le un nœud
                                            pour les implementations  */

        public Vector ImplemNode;       /* Les implmentations a dployer sur ce noeud (dans l'arbre) et qui constituent le granule */

…
}
```

La base de données objet est définie dans la classe *Placement* qui représente la classe principale, le code suivant en définit les principales propriétés et les méthodes d'accès à la base de données :

```
public class Placement {

	static Hashtable Tconst = new Hashtable();		/*	La table des contraintes	*/
	static Hashtable Tpref = new Hashtable();	/*	La table des preferences	*/
	static Hashtable Tnode = new Hashtable();	/*	La table des noeuds	*/

	static Hashtable PrecGraph = new Hashtable(); /* Le graphe de precedence */


/*------------------------------------  Les méthodes d'accès aux contraintes ----------------------------------------------*/

	/*-----  ajouter les contraintes d'une implementation -------*/

	public void addConst(Implementation imp){
		Tconst.put(imp.id,imp);
	}

	/*----- recuperer les contraintes d'une implementation -------*/

	public Implementation getConst(String key){
		return (Implementation)Tconst.get(key);
	}

	/*----- supprimer les contraintes d'une implementation ------*/

	public void removeConst(String key){
		Tconst.remove(key);
	}

/*--------------------------------------- Les méthodes d'accès aux preferences ---------------------------------------*/
	/*---- ajouter les préférences d'une implementation ----*/

	public void addPref(Implementation imp){
		Tpref.put(imp.id,imp);
	}

	/*-----  Récuperer les préférences d'une implementation -------*/

	public Implementation getPref(String key){
		return (Implementation)Tpref.get(key);
	}

	/*------ supprimer les préférences d'une implementation ------*/

	public void removePref(String key){
		Tpref.remove(key);
	}

/*--------------------------------------- Les méthodes d'accès aux  noeuds  ---------------------------------------*/

	/*--- ajouter les ressource d'un noeud -----*/

	public void addNode(Node n){
		Tnode.put(n.id,n);
	}

	/*----  récupérer les ressource d'un noeud ------*/

	public Node getNode(String key){
		return (Node)Tnode.get(key);
	}

	/---- supprimer les ressource d'un noeud -----*/

	public void removeNode(String key){
		Tnode.remove(key);
	}


}
```

Les contraintes (TCONT) :

Les deux implémentations p1 et p2 ont besoin des ressources “usedcpu” et “fr_mem_sp”, donc cette table d’hachage contient deux objets de type *Implementation* (les deux lignes) dont les clés d’accès sont P1 et P2. Chaque ligne contient deux objets de type *Resource* (les deux colonnes : usedcpu et fr_mem_sp de chaque ligne) :

Clès (*Iid*)	**Objets***(Implementation)*		
P1	**Clès**(name)	Usedcpu	fr_mem_sp
	Objet (*Resource*)	130. Name = “Usedcpu” 131. Weight = 3 132. Policy = toMinimize 133. Op = Lowerorequal 134. Value = 10 135. Shared = true	136. Name = “fr_mem_sp” 137. Weight = 3 138. Policy = toMaximize 139. Op = greaterorequal 140. Value = 50 141. Shared = true
P2	**Clès**(name)	usedcpu	fr_mem_sp
	Objet **(***Resource*)	142. Name = “Usedcpu” 143. Weight = 3 144. Policy = toMinimize 145. Op = Lowerorequal 146. Value = 10 147. Shared = true	148. Name = “fr_mem_sp” 149. Weight = 3 150. Policy = toMaximize 151. Op = greaterorequal 152. Value = 50 153. Shared = true

Les préférences (TPREF) :

Les ressources de préférences des implémentations p1 et p2 sont “cpu” et “fr_disk_sp” :

Clès (*Iid*)	**Objets***(Implementation)*		
P1	**Clès**(name)	Usedcpu	fr_disk_sp
	Objet (resources)	154. Name = “usedcpu” 155. Weight = 2 156. Op = greaterorequal 157. Value = 1.8 158. Shared = false	159. Name = “fr_disk_sp” 160. Weight = 1 161. Op = greaterorequal 162. Value = 1 163. Shared = true
P2	**Clès**(name)	Usedcpu	fr_disk_sp
	Objet **(**Resources)	164. Name = “usedcpu” 165. Weight = 2 166. Op = greaterorequal 167. Value = 200 168. Shared = false	169. Name = “fr_disk_sp” 170. Weight = 1 171. Op = greaterorequal 172. Value = 0.5 173. Shared = true

Les nœuds (TNODE) :

Les nœuds N1 et N2 offrent les ressources “screen”, “cpu”, “usedcpu”, “fr_disk_sp” et ”fr_mem_sp”, donc cette table d’hachage contient deux objets de type *Node* (les deux lignes) dont les clès d’accès sont N1 et N2. Chaque ligne contient cinq objets de type *Resource* (les cinq colonnes : *screen* et *cpu*, *usedcpu, fr_disk_sp* et *fr_mem_sp* de chaque ligne) :

Clès (*Nid*)	**Objets***(Node)*							
N1	**Clès**(name)	Screen	cpu	usedcpu	fr_disk_sp	fr_mem_sp	OS	plugin
	Objet (*Node*)	Name = "screen" Value = 15	Name = "cpu" Value = 1	Name = "usedcpu" Value = 60	Name = "fr_disk_sp" Value = 1	Name = "fr_mem_sp" Value = 50	Name = "OS" Value = UNIX	Name = "plugin" Value = ""
N2	**Clès**(name)	Screen	cpu	usedcpu	fr_disk_sp	fr_mem_sp	OS	Plugin
	Objet (*Node*)	Name = "screen" Value = 14	Name = "cpu" Value = 2	Name = "usedcpu" Value = 20	Name = "fr_disk_sp" Value = 4	Name = "fr_mem_sp" Value = 300	Name = "OS" Value = WIN2000	Name = "plugin" Value = java

Le diagramme de classe :

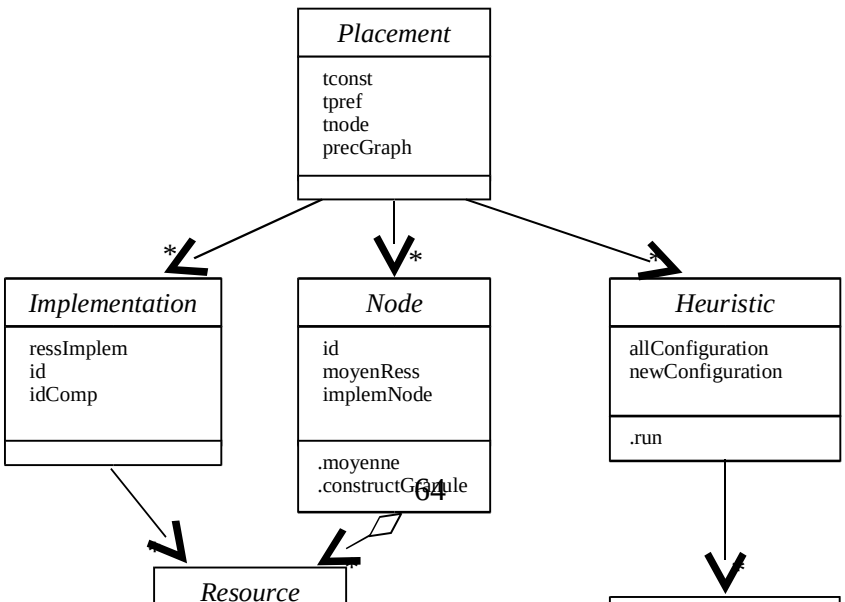
Pour déterminer la montée en charge de notre application nous devons d'abord construire son diagramme de classe qui comporte 6 classes :

174.*Placement, Implementaton, Resource,*

175.*Node* : La méthode *moyenne* calcule la moyenne des ressources supplémentaires d'une implémentation donnée ainsi que le pourcentage de préférence, de même pour la méthode *constructGranule* qui construit le granule en spécifiant ses contraintes et ses préférences et calculant sa moyenne des ressources supplémentaires et son pourcentage de préférence.

176.*Configuration* : représente une affectation des implémentations aux nœuds (la propriété *affectation*). Le choix d'affectation se fait suivant les critères d'optimisation suivants : le pourcentage de préférence(*percentage*) et la moyenne des ressources supplémentaires (*Msupp*), la propriété *setOfImplem* représente la liste des implémentations qui ont été placée par estimation. La prochaine implémentation à déployer fait partie de *setOfImplem.*

177.*Heuristic* représente un processus créé pour chaque ensemble d'implémentations s'exécutant en parallèle (*allConfiguration* : la liste de toutes les affectations déjà traitées, *newConfiguration* : l'affectation choisie et qui est en cours de traitement). Ce processus explore l'espace de recherche en passant d'une configuration à une autre et en essayant à chaque fois de se rapprocher de l'affectation optimale en choisissant celle où la fonction d'évaluation est maximale.



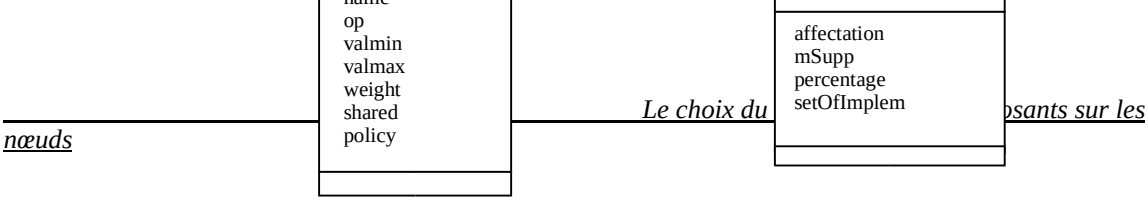


FIG. 6.6 – Le diagramme de classe

6.6.4.Traitement :

Le stockage dans la base de données s'effectue dans l'ordre suivant : les contraintes et les préférences, ensuite les nœuds dont chacun représente un processus permettant de récupérer les ressources et de calculer pour chaque implémentation la moyenne des ressources supplémentaires en se basant sur la formule présentée dans le paragraphe **5.7.2.b**. Ces processus s'exécutent en parallèle ce qui peut diminuer le temps de traitement, et leurs résultats sont stockés dans les objets *node* correspondants de la base de données, donc le tableau des moyennes des ressources supplémentaires sera distribué sur tout les objets *Node*. En se basant sur ce tableau, l'algorithme détermine les implémentations à déployer et les nœuds d'instanciation comme nous avons expliqué précédemment.

Après avoir choisi les implémentations et les nœuds, on applique l'heuristique permettant de maximiser le pourcentage de préférences et la moyenne des ressources supplémentaires, qui peut être appliquée parallèlement sur les ensembles des implémentations pouvant s'exécuter simultanément ce qui réduit l'espace de recherche et diminue le temps de traitement, pour cela nous avons défini la classe *Heuristic* qui implémente la classe *Thread*, par conséquent un processus sera créé pour chaque ensemble d'implémentations s'exécutant simultanément, ce processus se base sur l'algorithme cité dans le paragraphe **5.7.2.c** et cherche l'affectation optimale de l'ensemble des implémentations qui lui sont confiées.

Conclusion :

Nous avons présenté dans ce chapitre la modèlisation OWL des contextes requis et fournis ainsi que l'algorithme de résolution du problème de placement des composants sur les nœuds.

Nous avons implémenté cet algorithme en prenant en compte les différents types de ressources offertes par les nœuds. Nous avons également intégré cet algorithme dans l'architecture présentée dans le paragraphe 5.4.

Pour les ressources offretes par les liens, nous avons ajouté les tables de hachages nécessaires de la même façon que *Tconst* et *Tnode* et les procédures permettant de récupérer ces ressources. Il nous reste à faire l'intégration des ressources offertes par les liens dans l'affectation des implémentations aux nœuds ainsi que les testes concernant le temps de traitement et la montée en charge.

Chapitre 7

Conclusion :

Le déploiement d'une application sur un site donné, effectué par le modèle de composant, représente toutes les étapes nécessaires à son installation et son instanciation sur ce site, et permet à l'utilisateur d'accéder à l'application. L'utilisateur, étant mobile, peut changer de réseau ou même son terminal pour accéder au service, par conséquent l'application doit s'adapter aux caractéristiques du nouveau terminal (réseau).

Le déploiement adaptatif des applications multi-composants consiste essentiellement à choisir les implémentations de composants les mieux adaptées au contexte d'exécution et à les placer sur le réseau. Cette opération ne peut pas être effectuée par les modèles de composants existant car ils ne prennent pas en compte le contexte d'exécution lors du déploiement des applications. Les modèles de composants diffèrent, du point de vue adaptation, par les possibilités offertes aux utilisateurs de mieux contrôler les composants et d'adapter les applications à l'environnement d'exécution. Par rapport aux autres modèles, le modèle de composant CCM offre plus de flexibilité et de contrôle des composants et donne la possibilité de changer leurs assemblages, ce que peut être utilisé pour déployer les composants en fonction du changement du contexte d'exécution.

Plusieurs contributions ont été faites pour compléter les fonctionnalités des modèles de composants en leur permettant de prendre en compte le contexte d'exécution pendant l'installation et l'exécution des applications. Parmi celles-ci, l'architecture au dessus de CCM présentée dans le paragraphe 5.4 qui propose de rajouter les modules permettant de récupérer les informations de contextes et de déployer les composants les mieux adaptés à ce contexte.

Dans notre travail de DEA nous avons réalisé une partie du module *DeploymentContextAdaptor* : celle qui détermine les implémentations de composants à déployer sur le réseau et place ces implémentations sur les nœuds suivant le contexte d'exécution. Pour cela on a modélisé le contexte pour pouvoir le stocker et l'échanger en utilisant un des langages de descriptions présentés dans les chapitres 3 et 4.

OWL et RDF/RDFS sont des langages de descriptions puissant et plus utilisés pour la modélisation du contexte, ils permettent de décrire les classes de ressources, leurs relations et leurs sémantiques. Pour une éventuelle extension de notre modèle où l'aspect sémantique sera pris en compte pour déduire des nouvelles informations par exemple : «si le nœud offre la ressource dont la valeur satisfait l'implémentation alors cette implémentation peut être

déployée sur ce nœud», nous avons conçu une ontologie OWL permettant de décrire les ressources offertes par les nœuds et celles requises par les implémentations, et une méthode de choix et d'affectation des implémentations aux nœuds qui consiste à sélectionner parmi les implémentations de chaque composant celle qui sera déployée sur le réseau et affecter les implémentations choisies aux nœuds tout en maximisant le nombre de préférences satisfaites et optimisant le partage des ressources.

Afin d'évaluer cette méthode nous avons mis en œuvre la proposition en essayant pour chaque étape de l'algorithme de minimiser le temps de traitement et la montée en charge.

Pour cela nous avons associé à chaque nœud un *Thread* permettant de calculer ses pourcentages de préférences et ses moyennes de ressources supplémentaires. Nous avons aussi intégré cet algorithme dans l'architecture présentée dans le chapitre 5.4. Nous sommes en trains d'intégrer les ressources de connexions dans l'affectation des implémentations aux nœuds et d'effectuer les testes concernant le temps de traitement et la montée en charge.

Une perspective à court terme de cette approche serait la prise en compte de l'aspect sémantique des ressources pour anticiper le choix des implémentations et des nœuds (la première étape) en spécifiant les règles d'inférences nécessaires, ce qui peut diminuer le temps de traitement.

Annexe A :

La DTD OSD :

Le schéma suivant présente la DTD complet d’OSD :

```
<!ELEMENT ABSTRACT (#PCDATA)>

<!ELEMENT CODEBASE EMPTY>
<!ATTLIST CODEBASE FILENAME CDATA #IMPLIED>
<!ATTLIST CODEBASE HREF CDATA #REQUIRED>
<!ATTLIST CODEBASE SIZE CDATA #IMPLIED>

<!ELEMENT DEPENDENCY (CODEBASE|SOFTPKG) >
<!ATTLIST DEPENDENCY ACTION (Assert|Install) "Assert">

<!ELEMENT DISKSIZE EMPTY>
<!ATTLIST DISKSIZE VALUE CDATA #REQUIRED>

<!ELEMENT IMPLEMENTATION (CODEBASE | DEPENDENCY | DISKSIZE |
  IMPLTYPE | LANGUAGE | OS | PROCESSOR | VM)*>

<!ELEMENT IMPLTYPE EMPTY>
<!ATTLIST IMPLTYPE VALUE CDATA #REQUIRED>

<!ELEMENT LANGUAGE EMPTY>
<!ATTLIST LANGUAGE VALUE CDATA #REQUIRED>

<!ELEMENT LICENCE EMPTY>
<!ATTLIST LICENCE HREF CDATA #REQUIRED>

<!ELEMENT MEMSIZE EMPTY>
<!ATTLIST MEMSIZE VALUE CDATA #REQUIRED>

<!ELEMENT OS (OSVERSION)*>
<!ATTLIST OS VALUE CDATA #REQUIRED>

<!ELEMENT OSVERSION EMPTY>
<!ATTLIST OSVERSION VALUE CDATA #REQUIRED>

<!ELEMENT PROCESSOR EMPTY>
<!ATTLIST PROCESSOR VALUE CDATA #REQUIRED>

<!ELEMENT SOFTPKG (ABSTRACT | IMPLEMENTATION | DEPENDENCY | LICENSE |
  TITLE)*>
<!ATTLIST SOFTPKG NAME CDATA #REQUIRED>
<!ATTLIST SOFTPKG VERSION CDATA #IMPLIED>

<!ELEMENT TITLE (#PCDATA) >

<!ELEMENT VM EMPTY>
<!ATTLIST VM VALUE CDATA #REQUIRED>
```

Annexe B

La DTD de DSD :

Le schéma suivant présente la DTD principale de DSD :

```
<!ELEMENT Family (Id, ExternalProperties, Properties, Composition,
Assertions, Dependencies, Artifacts, Notifications, Interfaces,
Services, Activities)>
<!ELEMENT VarType EMPTY>
<!ATTLIST VarType Value (string | boolean | double) #REQUIRED>
<!ELEMENT Id (Name, Description, Producer, (License)?, Logo,
Signature)>
<!ELEMENT ExternalProperties (ExternalProperty)*>
<!ELEMENT ExternalProperty (Name, VarType, Description, Value)>
<!ELEMENT Properties (Property)*>
<!ELEMENT Property (Name, VarType, Description, DefaultValue,
DefaultEnabled, DefaultDisabled, TopLevel, Values)>
<!ELEMENT Composition (CompositionRule)*>
<!ELEMENT CompositionRule (Condition, ControlProperty, Relation,
RuleProperties)>
<!ELEMENT Relation EMPTY>
<!ATTLIST Relation Value (anyof | oneof | excludes | includes)
#REQUIRED>
<!ELEMENT Assertions (Guard, (Assertions | Assertion)*)>
<!ELEMENT Assertion (Guard, Condition, Description)>
<!ELEMENT Dependencies (Guard, (Dependencies | Dependency)*)>
<!ELEMENT Dependency (Guard, Condition, Description, Resolution,
Constraints)>
<!ELEMENT Artifacts (Guard, (Artifacts | Artifact)*)>
<!ELEMENT Artifact (Guard, Signature, ArtifactType, SourceName,
Source, DestinationName, Destination, EntryPoint, Mutable,
Permission, DiskFootPrint)>
<!ELEMENT Notifications (Guard, (Notifications | Notification)*)>
<!ELEMENT Notification (Guard, Name, Description)>
<!ELEMENT Interfaces (Guard, (Interfaces | Interface)*)>
<!ELEMENT Interface (Guard, Name, Description)>
<!ELEMENT Services (Guard, (Services | Service)*)>
<!ELEMENT Service (Guard, Name, Description)>
<!ELEMENT Activities (Guard, (Activities | Activity)*)>
<!ELEMENT Activity (Guard, Name, Action, When, Description)>
```

Bibliographie

[CCM02] CORBA Components Version 3.0 an Adopted Specification of the Object Management Group, June 2002.
[ACC02] Projet ACCORD, Le modèle de composant CORBA, Mai 2002.
[HAL97] Richard S.Hall, Dennis Heimbigner, Alexander L. Wolf. Software Deployment Languages and Schema, December 1997.
[HAL97] Richard S.Hall, Dennis Heimbigner, Alexander L. Wolf. Software Deployment Languages and Schema, December 1997.
[HAL99] Richard S.Hall, Dennis Heimbigner, Alexander L.Specifying the Deployable Software Description Format in XML, March 1999.
[TAT04] Tatiana Kichkaylo, Anca Ivan, Vijay Karamcheti. Sekitei : An AI planner for Constrained Component Deployment in Wide-Area Networks, Technical report TR2004-851 , March 2004.
[KICH]Tatiana Kichkaylo, Anca Ivan, Vijay Karamcheti. : Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques.
[TAKI]Tatiana Kichkaylo. : Timeless Planning and the Component Placement Problem.
[FOS00] I.Foster, A.Roy, and V.Sander. A quality of service architecture that combines resource reservation and application adaptation. In IWQOS, 2000.
[GRI01] S.Gribble, al. The Ninja architexture for robust Internet-scale systems and services. *Computer Networks*, 35(4) :473-497, 2001.
[XFU01] X.Fu, W.Shi, A.Akkerman, V.Karamcheti. CANS:Composable, Adaptive Network Services infrastructure. USITS-3, 2001.
[IVA02] A.Ivan, J.Harman, M.Allen, V.Karamcheti. Partitionable Services : A framework for seamlessly adapting distributed applications to heterogenous environments. In HPDC-11, 2002.
[REI00] P.Reiher, R.Guy, M.Yarvis, A.Rudenko. Automated planning for open architectures. *OPENARCH*, 2000.
[AYE04] D.Ayed, C.Taconet, G.Bernard. A Data Model for Context-aware Deployment of Component-based Applications onto Distributed Systems - Component-oriented approaches to context-aware systems Workshop ECOOP'04 - Oslo, Norway - June 2004

[AYD04]D.Ayed, C.Taconet, G.Bernard. Architecture à base de composants pour le déploiement adaptatif des applications multi-composant - Journées Composants 2004, Lille, France, Mars 2004.
[DHO04]D.Ayed, C.Taconet, G.Bernard. Context-Aware Deployment of multi-component applications, 2004.
[RDF04] W3C "Resource Description Framework (RDF): Concepts and Abstract Syntax"
Février 2004
[CCP04] W3C "Composite Capability/Preference Profiles (CC/PP): Structure and
Vocabularies 1.0" Janvier 2004
[COA03] Information Society Technologies (IST) Programme COACH
«Component based open source architecture for distributed Telecom
Applications. WP2 : Specification of the deployment and configuration». Juillet 2003
[PIE01] Pierre-Antoine Champin, RDF Tutorial. April 2001.
[NAB] N. Belhanafi «Enregistrement et propriétés non fonctionnelles» Rapport de stage. Septembre 2003
[ROB] R.Robinson «Context Management in Mobile Environments». Octobre 2000
[BRU04] E.Bruneton, T.Coupaye, J.B.Stefani «The Fractal Component Model», February 2004.
[OSD97] W3C. The Open Software Description Format (OSD). August 1997.
[OWLO04] OWL Web Ontology Language Overview. February 2004.
[OWLG04] OWL Web Ontology Language Guide. February 2004.
[TUA] TA Tuan Anh. Principe de RDF et RDFS.
[DRE03] J.Dréo, A.Pétrowski, P.Siarry, E.Taillard «Métaheuristiques pour l'optimisation difficile», 2003
[COL02] Y.collette, P.Siarry «Optimisation multiobjectif », 2002