République Algérienne Démocratique et Populaire
Ministère de L'enseignement Supérieur et de la Recherche Scientifique
Université Constantine 2 Mahri Abdelhamid

Faculté des nouvelles technologies de l'information et de la communication
Département des Technologies des Logiciels et des Systèmes d'Information

Thèse

Pour obtenir le diplôme de
Doctorat en Sciences

Approche basée Agents Mobiles et Composants pour développer des applications ouvertes et adaptables

Par
Abderrahim Siam
Soutenue le : 02/02/2015

Devant le Jury :

Pr Belala Faiza	Professeur à l'universitéMehri abdelhamid Constantine	Président
Pr Maamri Ramdane	Professeur à l'universitéMehri abdelhamid Constantine	Rapporteur
Pr Rahmoun Abdellatif	Professeur à l'université université Djillali Liabes Sidi Bel abbes	Examinateur
Pr Chikhi Salim	Professeur à l'universitéMehri abdelhamid Constantine	Examinateur
Pr Kazar Okba	Professeur à l'universitéMohamed Khider de Biskra	Examinateur
Pr Sahnoun Zaidi	Professeur à l'universitéMehri abdelhamid Constantine	Invité

République Algérienne Démocratique et Populaire
Ministère de L'enseignement Supérieur et de la Recherche Scientifique
Université Constantine 2 Mahri Abdelhamid

Faculté des nouvelles technologies de l'information et de la communication
Département des Technologies des Logiciels et des Systèmes d'Information

Thèse

Pour obtenir le diplôme de
Doctorat en Sciences

Approche basée Agents Mobiles et Composants pour développer des applications ouvertes et adaptables

Par
Abderrahim Siam

Remerciements

Je tiens tout d'abord à remercier Professeur Ramdane MAAMRI et Professeur Zaidi SAHNOUN qui m'ont fait l'honneur d'être les rapporteurs de cette thèse.

Mes sincères remerciements vont aux membres du jury qui m'ont fait l'honneur d'évaluer les travaux ici présentés. Je remercie Professeur Belala F, qui a présidé le jury. Je remercie Pr Maamri Ramdane et Pr Sahnoun zaidi qui ont accepté de rapporter cette thèse. Je remercie aussi Pr Rahmouni A et Pr Chikhi Salim qui ont pris part au jury. Je les remercie tous pour leur évaluation qui donne toute sa valeur à cette thèse ainsi que pour leurs remarques et questions..

Je tiens à remercier tout un ensemble de personnes grâce à qui, directement ou indirectement, j'ai pu mener ce travail à bien :

- *Tous les membres du laboratoire LIRE, particulièrement les membres de l'équipe GL&IA ;*
- *Toute ma famille : mes parents, ma femme, mes frères et ma sœur,...*
- *Mes amis, particulièrement : Benaboud Rouhellah, Badis Abdelhafid et le gentleman Ramoul Hichem.*
- *Je n'oublie pas les étudiants que j'ai eu le plaisir d'encadrer pour les richesses qu'ils m'ont apportés tant sur le plan scientifique que sur le plan humain.*
- *Je n'oublie pas de penser aux enseignants que j'ai eus par le passé depuis l'école primaire.*

Abstract

This research focuses on the development of, distributed, open and adaptive applications in which software components and agents (mobile and not mobile) are involved. Software components and agents present two technologies that have a great impact on the software development. The proprieties of distribution, openness and adaptability are imposed as essential characteristics in several systems by the wide spread of Internet and Web technologies as well as the presence of means of data processing in the majority of equipment around us. The development of distributed, open and adaptive applications needs special means, and which could provide solid solutions that meet all the characteristics related to distribution, openness and adaptability. In this thesis, the development of such applications is addressed from the way of the development of organizational multi agent systems. We proposed a combination between components and agents to define a flexible organizational model of MAS based on three concepts: roles, self-adaptive agents based on components and fuzzy groups. Roles are played by agents in fuzzy groups. A fuzzy group is a fuzzy set of agents characterized by a membership function reflecting the notion of partial membership and expressing the membership degree of each agent to the group. The membership degree expresses the degree of capacity of each agent to play a role. We proposed a fuzzy measure of the capacity of agents to play roles. We proposed a model of auto adaptive agents constructed and adapted by automatic assembly (reassembly) of software components that implement required capabilities to play roles. The proposed model and introduced concepts have been tested using the Madkit platform.

Keywords*:* *software components, self adaptive agents, adaptive applications, distributed and open applications, organization, fuzzy sets.*

Résumé

Ce travail de recherche porte sur le développement d'applications distribuées, ouvertes et adaptables en faisant intervenir les composants logiciels et les agents (mobiles et non mobiles) qui présentent deux technologies ayant un grand impact sur le développement d'applications informatiques. L'ouverture, la distribution et l'adaptabilité s'imposent de plus en plus comme caractéristiques importantes et essentielles dans plusieurs systèmes informatiques vu la popularisation de l'internet et des technologies Web ainsi que la présence des moyens de traitement de l'information dans plupart des dispositifs qui nous entourent. Le développement d'applications distribuées, ouvertes, et adaptables nécessite des moyens qui peuvent fournir des solutions solides et qui sont en mesure de répondre aux exigences ainsi qu'aux caractères spécifiques imposés sur les applications par les propriétés d'ouverture, de distribution et d'adaptation. Dans cette thèse, on a adressé le développement de telles applications sous l'angle du développement de systèmes multi agents organisationnels. On a proposé une combinaison entre les composants logiciels et les agents pour définir un modèle d'organisation flexible basé sur les concepts de rôles, d'agents auto adaptatifs à base de composants et les groupes flous d'agents. Les rôles sont joués par les agents dans des groupes flous. Un groupe flou est un ensemble flou d'agents caractérisé par une fonction d'appartenance reflétant la notion d'appartenance partielle et exprimant le degré d'appartenance de chaque agent au groupe. Le degré d'appartenance d'un agent à un groupe exprime le degré de sa capacité pour jouer le rôle joué dans le groupe. On a proposé une mesure floue pour la mesure des capacités des agents pour jouer les rôles et un modèle d'agents auto adaptatifs construits et adaptés par assemblage automatique de composants qui implémentent les compétences requises pour jouer les rôles. Les modèles et les concepts proposés ont été testés à travers le développement d'une application multi agents sous la plateforme MadKit.

Mots clé : *composants logiciels, agents auto adaptables, applications adaptables, applications ouvertes et distribuées, organisation, ensembles flous.*

ملخص

هذا البحث يتناول بالدراسة تطوير البرامج و التطبيقات التي تتصف بكونها موزعة, مفتوحة والقابلة للتأقلم من خلال استغلال تكنولوجية البرمجيات الجزئية و تكنولوجية الوكلاء(متنقلة وغير متنقلة) والتي لهما تأثير هام على صعيد تطوير وهندسة البرامج. إن الانتشار الكبير للانترنت و تكنولوجياتها على غرار انتشار وسائل معالجة البيانات في اغلب الأجهزة المحيطة بنا يحتم يوما بعد الآخر على التطبيقات أن تكون موزعة, مفتوحة وقابلة للتأقلم. إن تطوير تطبيقات بهذه المواصفات يتطلب وسائل من شأنها أن تقدم حلولا ناجعة و التي تتماشى مع ما تفرضه خصائص التوزيع, الافتتاح و قابلية التأقلم من مميزات. في هذه الأطروحة نقترح تطوير البرامج و التطبيقات الموزعة, المفتوحة والقابلة للتأقلم من خلال تطوير نظم مظمة متعددة الوكلاء أين نقترح صيغة نستخدم فيها البرمجيات الجزئية و الوكلاء جنبا إلى جنب. الاقتراح المقدم يتمثل في اقتراح نموذج تنظيم يستند على المفاهيم التالية: الأدوار, وكلاء ذاتية التأقلم يتم بناؤها عبر تجميع البرمجيات الجزئية و أخيرا تجمعات مبهمة للوكلاء. هذه التجمعات هي عبارة عن مجموعات مبهمة يتم بداخلها لعب الأدوار من طرف الوكلاء. كل تجمع يتميز بدالة انتماء تعكس مفهوم الانتماء الجزئي إلى مجموعة وتعبر عن مدى انتماء كل وكيل إلى التجمع. هذا المقدار يعبر بدوره عن مدى قدرة الوكيل على لعب الدور المتعلق بالتجمع الذي ينتمي إليه الوكيل. من أجل استكمال هذا النموذج قمنا باقتراح قياس مبهم لتقييم مدى قدرة الوكلاء على إتمام انجاز الأدوار. في الوقت ذاته, تم اقتراح نموذج وكلاء تبنى و تتأقلم من خلال تجميع و إعادة تجميع البرمجيات الجزئية الداخلة في تشكيل الوكيل و التي تضم الكفاءات التي تلزم الوكلاء حتى يتسنى لهم لعب الأدوار داخل التنظيم. النماذج و المفاهيم المقترحة تم تجريبها من خلال تطبيق تم انجازه على أرضية تطوير النظم متعددة الوكلاء ماد كيت.

كلمات مفتاحية: البرمجيات الجزئية, الوكلاء ذاتية التأقلم, التطبيقات القابلة للتأقلم, التطبيقات المفتوحة و الموزعة, التنظيمات, المجموعات المبهمة.

Tables des matières

Table des figures

Introduction générale

Depuis quelques années la conception et le déploiement d'applications ouvertes, distribuées et adaptables présentent de grands enjeux pour les chercheurs dans plusieurs branche de l'informatique. L'ouverture, la distribution et l'adaptabilité s'imposent de plus en plus comme caractéristiques importantes et essentielles dans plusieurs systèmes informatiques vu la popularisation de l'internet et des technologies Web ainsi qu'avec la progression de l'informatique ubiquiste du fait que la plupart des dispositifs qui nous entourent comme les téléphones mobiles sont équipés de moyens de traitement de l'information et de plus en plus dotés d'équipements de communication.

Dans cette thèse nous abordons le développement d'applications ouvertes, distribuées et adaptables particulièrement en exploitants les technologies de composants logiciels, d'agents et d'agents mobiles. Nous allons exposer ci-dessous le contexte et les motivations nous ayant poussé à aborder cette problématique. Ensuite, nous détaillerons les objectifs ainsi que la démarche suivie et nous terminerons par la présentation de la structure du manuscrit.

Le développement d'applications distribuées, ouvertes, et adaptables nécessite des moyens qui peuvent fournir des solutions solides et qui sont en mesure de répondre aux exigences ainsi qu'aux caractères spécifiques imposés sur les applications par les propriétés d'ouverture, de distribution et d'adaptation. La propriété de distribution impose qu'une application soit implémentée sous forme d'un ensemble d'entités logicielles et qui s'exécutent sur des machines distantes avec une distribution des computations et une décentralisation des ressources et des connaissances. Derrière cette distribution, se cachent plusieurs problèmes de taille à gérer. La propriété d'ouverture impose de son côté que les applications doivent obéir à des règles d'interopérabilité, de pouvoir changer de frontières par l'ajout (intégration) de nouvelles entités ou par la suppression (retrait) de quelques unes parmi celles qui la composent tout en assurant un contrôle sur les échanges entre les applications et leurs environnements. Quant à l'adaptation, cette propriété impose que les applications soient capables de réagir en s'adaptant en réponse à des besoins applicatifs ou pour être plus conformes avec les environnements d'exécution qui ne cessent de changer constamment. Les propriétés de distribution, d'ouverture et de d'adaptation ne sont pas totalement indépendantes les unes des autres. Par contre, les aspects couverts par chacune des propriétés sont fortement liés les uns aux autres.

Le développement des applications informatiques s'inscrivant dans ce contexte mobilise différentes technologies. Parmi celles-ci, nous pouvons notamment citer la technologie des composants logiciels et celles des systèmes multi-agents. Ces deux technologies définissent le cadre de nos travaux. Les composants logiciels et les systèmes multi agents présentent

actuellement deux approches de développement de logiciels très importantes et qui sont toutes les deux bien équipées de moyens et d'outils leurs permettant de contribuer considérablement dans le développement d'applications distribuées, ouvertes et adaptables. Les approches de développement à base de composants ainsi que celles basées agents proposent des abstractions pour organiser le logiciel sujet de développement en un ensemble d'entités logicielles dans le but de réduire la complexité, d'assurer une meilleure structuration du logiciel ainsi que pour mieux gérer son évolution.

Dans le contexte décrit ci-dessus, nous sommes confrontés à des problèmes qui nous poussent à utiliser conjointement les composants et les d'agents (mobiles et non mobiles) à deux niveaux d'abstraction différents, programmation et conception. Malgré que les systèmes multi agents présentent des avancées technologiques par rapport aux composants, ces derniers préservent toujours quelques atouts incontournables. Particulièrement leurs capacités de structuration et de reconfiguration dynamique ainsi que la grande mise en valeur de la réutilisation. Deux questions fondamentales aiguillent nos travaux dans cette thèse : quelles sont les meilleures formes de combinaisons des composants logiciels, des agents et des agents mobiles pour développer des applications qui sont distribuées, ouvertes et adaptables ? Quelles sont les combinaisons qui permettent de tirer profit au maximum des avantages des deux paradigmes à savoir la structuration et la réutilisation du côté des composants et de tous les avantages découlant de l'utilisation des agents et des agents mobiles ?

La communauté des chercheurs travaillant sur des problématiques liées au développement d'applications et des systèmes informatiques ou des problématiques du génie logiciel en général proposent des solutions dans lesquelles ils combinent souvent plusieurs éléments. Ces éléments peuvent être des techniques, des méthodes, des approches, des modèles et des paradigmes de conception et de programmation différents dans le but de tirer profit de plusieurs points forts caractérisant chacun de ces éléments d'un coté, et d'un autre côté faire compenser les limites et les faiblesses d'un élément par les caractéristiques avantageuses des autres éléments.

Dans cette même tendance de combinaison s'inscrivent nos travaux de thèse. Ces travaux entreprennent l'étude et l'analyse des différentes possibilités de combinaison et du potentiel de fertilisation croisé entre les composants et les agents entant que deux approches de conception et de développement de logiciel ayant actuellement un grand impact. Ces deux approches proposent des abstractions pour organiser le logiciel comme une combinaison d'éléments logiciels, avec pour objectifs communs : réduire la complexité ; assurer une meilleure structuration du logiciel ; et de faciliter son évolution. Les systèmes multi-agents à l'aide de leurs capacités d'auto-organisation et d'utilisation de connaissances repoussent encore plus loin le niveau d'abstraction.

Dans ce travail, nous défendons la thèse qu'un élément de solution intéressant pour aborder le développement d'applications distribuées, ouvertes, et adaptables, consiste à fournir une infrastructure à composants pour *la construction et l'adaptation d'agents par*

assemblage de composants. Les agents ainsi construits font parties d'une société d'agents structurée selon *une organisation flexible* dans laquelle les agents appartiennent à des ensembles flous d'agents dans lesquels sont autorisés à opérer des tâches selon une mesure de leurs capacités effectives pour l'achèvement de ces tâches. La mesure des capacités des agents est une mesure floue multidimensionnelle, qui à partir des descriptions des composants constituant un agent, la mesure produit un indice de capacité de l'agent à l' égard d'une tâche donnée exprimant au même temps le degré d'appartenance de l'agent à l'ensemble flou auquel il appartient. Les ensembles flous sont utilisés comme moyen de partitionnement de l'organisation que nous avons appelé groupes flous ou *F-groupes* permettant d'exploiter la notion d'appartenance partielle à un ensemble pour exprimer un passage graduel entre la situation de non capacité d'un agent à l'égard d'une tâche à la situation d'une capacité totale. Ces idées sont développées le long des chapitres IV et V dans lesquels nous présentons l'approche proposée pour développer des applications distribuées, ouvertes et adaptables dans la quelle nous combinons les composants logiciels et agents (mobiles et non mobiles). Mais bien avant ces deux chapitres, les trois premiers chapitres de ce manuscrit constituent un état de l'art selon la manière suivante : le premier chapitre est consacré pour présenter et définir les propriétés de distribution, d'ouverture et d'adaptabilité des applications en mettant l'accent sur les contraintes imposées et les difficultés à surmonter lors du développement d'applications caractérisées par de telles propriétés. Le deuxième chapitre fait l'objet d'une présentation de l'approche de développement par composants *component-based development (CBD)* et le génie logiciel à base de composants *Component-based software engineering* (*CBSE*) ainsi que la majorité des concepts en relation avec cette approche. Une attention particulière est prêtée pour l'adaptation et les possibilités de reconfiguration dynamique des applications à base de composants. Dans le chapitre trois, après avoir présenté d'une manière sommaire le paradigme agent, une analyse comparative entre les approches orientées composant et celles orientées agent est présentée. Ce chapitre dresse un état de l'art des approches dans les quelles les paradigmes composant et agent sont combinés tout en discutant les apports mutuels des composants et des agents. Pour la mise en valeur est l'évaluation de notre approche, deux études de cas ont été conduites et présentées dans le chapitre VI dans lequel nous présentons vers sa fin un bilan dans lequel nous évaluons l'approche proposée. Nous terminons ce manuscrit avec une conclusion dans laquelle nous résumons les contributions de cette thèse ainsi que ses limites et ses perspectives.

Chapitre I :
Applications distribuées, ouvertes et adaptables

Chapitre I :

Applications distribuées, ouvertes et adaptables

1.1 Introduction

Pour plusieurs raisons, comme la grande propagation de l'internet et des technologies Web ainsi que la présence de processeurs et de moyens de traitement de l'informations dans la majorité des équipements qui nous entourent, l'ouverture, la distribution et l'adaptation s'imposent comme caractéristiques ayants d'extrêmes importances dans les applications informatiques de nos jours. Noun nous intéressons au développement d'applications distribuées, ouvertes et capables de s'adapter en réponse à des besoins applicatifs ou pour être plus conformes avec les environnements d'exécution qui ne cessent de changer constamment. Dans ce premier chapitre nous présentons les caractéristiques liées à la distribution, l'ouverture et l'adaptation des applications d'une manière générale. Cette présentation servira de cadre pour étudier et situer plus particulièrement dans les deux chapitres qui suivent les apports dans le développement de telles applications des composants logiciels [164] et des agents [52], notamment mobiles [18], en tant que deux approches de développement ayant leurs impacts importants sur le développent de logiciels. Nous présentons en premier lieu un aperçu sur le développement d'applications distribuées; en deuxième lieu nous traitons la propriété d'ouverture ainsi que la manière dont elle est supportée dans différentes applications ; et en troisième position nous présentons l'adaptation des applications d'une manière générale et nous reviendrons sur l'adaptation des applications à base composants logiciels et à base d'agents dans le deuxième et le troisième chapitres en faisant une projection

des concepts liés à l'adaptation présentés dans ce chapitre sur les applications à base de composants, celles basées agents et les applications dans lesquelles les deux approches sont combinées, tout en insistant également sur les caractéristiques de distribution et d'ouvertures.

1.2 Développement d'applications distribuées

Le développement d'applications distribuées [159] partage beaucoup de points en commun avec le développement de n'importe quelles autres applications informatiques. La particularité à prendre en compte lors du développement de telles applications c'est que les composantes de ces dernières sont à exécuter sur des machines distantes les une des autres et qui communiquent via des réseaux (figure 1.1). Cette particularité fait des applications distribuées des applications plus complexes que les applications centralisée ; par conséquence, elles sont plus difficiles à concevoir, à implémenter et à tester. En outre, il est très difficile de maitriser les propriétés émergeantes de la distribution d'une application ou d'un système à cause de la complexité des interactions entre les composants et l'infrastructure du système. Malgré les difficultés liées à la distribution, cette propriété a ses impacts positifs sur les applications informatiques en mettant en avant des propriétés importantes tel que le partage de ressources, la concurrence, l'ouverture, l'extensibilité et la tolérance aux pannes.

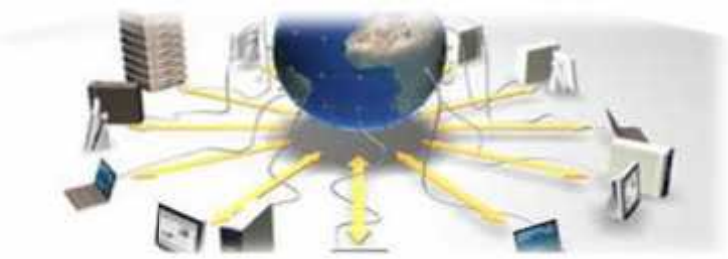

Figure 1.1 : *Machines distantes communiquant via des réseaux*

1.2.1 Problèmes liés au développement des applications distribuées.

Les applications et les systèmes distribués sont caractérisés par leurs grandes complexités et par le caractère inhérent du non prédictibilité des services qu'ils fournissent. Cette complexité est due à l'impossibilité de la mise en œuvre de contrôles *Top-Down* du fait que les nœuds qui fournissent les services sont des systèmes indépendants qui n'obéissent pas à une seule autorité ainsi que les réseaux qui assurent les connexions entre les nœuds présentent eux même des systèmes indépendants gérés séparément des nœuds du système. Le développement des applications distribuées et plus particulièrement leurs conceptions s'articulent sur: la transparence, l'ouverture, l'extensibilité, la sécurité, les qualités de service et la gestion des pannes.

La propriété de transparence est en relation avec la visibilité d'un système par ses utilisateurs. Un système distribué est dit transparent que s'il est vu par ses utilisateurs

comme étant un seul système dont le comportement n'est pas affecté par la manière dont il est distribué. Cette propriété est très importante et fortement recommandée pour les applications distribuées. Mais, en pratique et pour plusieurs raisons il est impossible d'avoir des systèmes distribués complètement transparents. La principale raison consiste en l'impossibilité de la mise en œuvre d'un contrôle centralisé pour un système distribué et donc les différentes entités peuvent se comporter de manières différentes à différents moments. En outre, les contraintes liées aux réseaux d'interconnexion tel que les délais d'attentes et les localisations de certaines ressources sont inévitables et même par fois incontournables. Vu la difficulté, voire l'impossibilité, de développer des systèmes qualifiées de totalement transparents, il est utile de mettre les utilisateurs au courant de quelques aspects de distribution du système sur lequel ils travaillent pour qu'ils puissent comprendre quelques conséquences de la distribution comme les délais d'attentes et la perte de nœuds.

Le développement d'applications et de systèmes distribués prend en charge généralement comme l'une des préoccupations essentielles la possibilité d'intégrer et de permettre l'interopérabilité de différentes entités ou composantes arrivants de différentes sources. Les aspects qui tournent au tour de cette préoccupation sont couverts par la propriété d'ouverture que nous présentons avec plus de détails dans la section 1.3.

Concernant l'extensibilité, [121] cette propriété souvent liée à la distribution désigne les capacités d'une application pour ajouter plus de ressources suivant le nombre croissants des utilisateurs ; les capacités d'une application de répartir géographiquement ses composantes sans que cette répartition ait des influences négatives sur ses performances ; et finalement, les capacités d'une application de pouvoir gérer son extensibilité.

La sécurité présente l'une des plus importantes problématiques liées au développement des applications distribuées où il est extrêmement difficile d'établir des politiques de sécurité sérieusement applicables à toutes les composantes d'une application vu que ces dernières peuvent faire parties de différentes organisations où des politiques et des mécanismes de sécurité non compatibles les uns avec les autres sont utilisés. Par rapport aux applications centralisées, les applications distribuées peuvent être attaquées par un nombre plus important de points et avec plusieurs manières. Les principaux types d'attaques que les applications distribuées peuvent être victimes sont [159] : l'interception des communications entre différentes composantes d'une application provoquant une perte de confidentialité ; le bombardement d'un nœud par des fausses requêtes empêchant ce dernier de traiter les requêtes valides et fournir les services attendus ; le changement des données et des services d'une application ; et finalement, les attaques par la génération de fausses informations utilisables pour atteindre quelques privilèges comme la génération de mots de passe permettant l'accès à des données, des services et des ressources inaccessible sans avoir d'autorisations.

La qualité des services fournis par une application distribuée reflète sa capacité de fournir d'une manière fiable et dans des délais de réponse acceptables ses services [159]. Lors du développement d'une application distribuée les besoins en termes de qualités de services

doivent être spécifiés avant sa conception pour que cette dernière soit conçue de sorte à prendre en compte les qualités quelle doit assurer quand elle offre ses services. Un point plus important à prendre en compte lors du développement d'une application distribuée c'est que les dégradations et les pannes sont inévitables et que l'application doit résister dans les situations pareilles. En pratique, on ne peut pas assurer une bonne qualité de service à tout moment, il y a des situations où le nombre de requêtes atteint un pic et pour assurer une bonne qualité de service l'application nécessite plus de ressources dont elle n'exploite pas la plupart du temps. Dans ce cas, on accepte une dégradation de la qualité de service que d'investir dans des ressources qui seront sous exploitées. La qualité de service devient critique dans le cas où l'application à développer traite des données de type vidéo ou audio ou toutes autres données critiques. Dans ces circonstances, l'application doit préserver une qualité de service haut delà du seuil toléré sinon, les vidéos ou les sons seront dégradés.

1.3 Ouverture et applications ouvertes

La propriété d'ouverture dans les systèmes informatiques acquit de plus en plus d'importance en présentant une problématique fortement abordée dans plusieurs travaux de recherches. La notion d'ouverture dépend étroitement de la nature de la relation d'un système informatique avec son environnement. Les systèmes informatiques regroupent un ensemble de moyens humains (concepteurs, programmeurs, utilisateurs, ...) et un ensemble de moyens techniques qui regroupent des ressources matérielles (serveur, routeur, ...) et des logiciels (système d'exploitation, SGBD,…). Ces éléments sont en interactions permanentes, et leurs principales fonctionnalités sont de traiter, stocker, acheminer ou présenter de l'information. L'environnement d'un système informatique est constitué d'autres systèmes qui l'entourent et qui regroupent à leurs tours des éléments de types matériels, logiciels et humains.

1.3.1 Environnement d'un système informatique

D'une manière générale, un environnement peut être vu comme étant ce qui entoure quelque chose. L'environnement d'un système informatique est définit par Van Gigch [168] comme tout autre système sur lequel les mécanismes de décision du système n'ont pas de contrôle. Selon cette définition, nous pouvons traiter un environnement d'un système comme étant tout ce qui est en dehors de celui-ci et n'est pas sous son contrôle. Les échanges entre un système informatique et son environnement sont principalement les transmissions d'informations à traiter ou traitées. Les influences réciproques entre un système et son environnement sont répertoriées en deux classes [172]. La première concerne les données échangées alors que la seconde concerne les modifications des frontières du système. Les données échangées entre un système et son environnement regroupent les données introduites dans le système par les éléments de l'environnement et celles que le système génère à destination de son environnement. Les modifications des frontières du système peuvent être

des extensions de frontières suite à l'intégration de certains éléments de son environnement, ou des restrictions de frontière par abandon de certains de ses éléments.

1.3.2 Systèmes informatiques ouverts

La définition de l'ouverture d'un système repose sur la notion de l'environnement. A l'opposé d'un système ouvert, un système fermé n'a pas d'environnement. Par conséquence, il n'a pas d'échanges avec des systèmes extérieurs et son état n'est influencé que par ses propriétés et ses conditions initiales [172]. L'étude de l'ouverture dans les systèmes consiste donc à définir les moyens d'établir et gérer les échanges que peut avoir un système avec les autres systèmes de son environnement. Il s'agit de définir la nature des échanges : échanges de messages (communication), les données échangées, les supports et flux d'échanges ainsi que les impacts des échanges (stabilité, adaptation, fiabilité). De cet effet, un système ouvert peut être défini comme un système situé dans un environnement et qui interagit, communique et faisant des échanges aves d'autres systèmes de son environnement. En d'autres termes, un système ouvert est un système qui fournit des avantages en interopérabilité, portabilité, et standards ouverts de logiciels ; ou configuré pour permettre des accès non restreints par des personnes et/ou des ordinateurs [93]. L'interopérabilité définit la capacité d'échange des données et services entre systèmes ; La portabilité concerne la capacité et la facilité d'intégration d'un système informatique dans un autre, ce qui est rapporté à la problématique d'extension d'un système par l'intégration d'autres systèmes de son environnement ; par l'utilisation de standards ouverts il est visé de faciliter la modification des logiciels utilisés de façon à ce qu'ils puissent mieux répondre aux besoins d'un système ; Les accès non restreints de personnes et/ou d'ordinateurs dépendent des applications et/ou des systèmes.

1.3.3 Propriétés des applications ouvertes

En analysant et faisant correspondance entre toutes les définitions de l'environnement et d'ouverture de systèmes présentées dans les deux sections précédentes, nous pouvons constater que les propriétés suivantes caractérisent un système informatique ouvert : l'interopérabilité, l'ajout / la suppression d'éléments, et le contrôle d'accès.

1.3.3.1 Interopérabilité

La propriété d'interopérabilité résume et traduit le besoin de faciliter les interactions entre un système et d'autres systèmes de son environnement. Ces interactions sont des accès aux fonctionnalités d'autres systèmes ainsi que des d'échanges d'informations. Un autre point qui est concerné par l'interopérabilité consiste à faciliter et pouvoir utiliser les informations échangées [37]. Pour arriver à répondre à ces besoins il est nécessaire d'avoir des structures

et des interfaces précises pour la représentation et l'échange de données. A cet égard, on distingue dans la littérature trois niveaux d'interopérabilité [42] : un niveau syntaxique qui concerne le format de représentation des données et des connaissances, un niveau structurel qui fait référence à l'organisation de l'information ; et un niveau sémantique qui prend en charge l'interprétation des représentations relativement à un domaine. Pour garantir l'interopérabilité entre différents systèmes développés par des firmes et des développeurs différents, les développeurs répondent à des consortiums de standardisation et de normalisation tel que ISO ou ANSI pour développer les interfaces de leurs systèmes ainsi que pour structurer les données échangées. Une norme est une règle fixant les conditions de la réalisation d'une opération, de l'exécution d'un objet ou de l'élaboration d'un produit dont on veut unifier l'emploi ou assurer l'interchangeabilité [93]. Les normes n'imposent pas la façon dont les fonctionnalités d'un système sont conçues et développées dans son architecture interne, elles ne définissent que les objectifs des fonctionnalités, leurs données respectives d'entrée et de sortie et les interfaces d'interopérabilités.

L'interopérabilité peut être mieux illustrée à travers Internet. Dans ce réseau géant les informations transitent d'un réseau à un autre via des routeurs fabriqués généralement par des constructeurs différents. L'utilisation des protocoles standards de transport et de transmission de données tel que TCP/IP permet à tout routeur quelque soit son constructeur et indépendamment de ses mécanismes de fonctionnement de pouvoir envoyer et recevoir des données provenant de tout autres routeurs. L'interopérabilité est garantie par des normes qui couvrent toutes les étapes du processus d'échange d'informations entre systèmes comme les normes de codage de caractères tel que UTF-8, les normes de représentation structurée des données tel que XML et RDFS et Les normes de transport internet et de transmission des données tel que TCP/IP et HTTPS.

En pratique, il n'existe pas de normes pour toutes les applications existantes, et qui sont sensées de régir toutes leurs interactions avec tous les systèmes avec lesquels elles sont susceptibles d'interagir. Ainsi, dans le cas d'absence de normes ou dans le cas de l'impossibilité de leurs utilisations, les constructeurs et développeurs de systèmes décrivent eux-mêmes et mettent à la disposition de leurs environnements les représentations des données échangeables ainsi que les interfaces d'interaction de leurs produits matériels et logiciels. Malgré son importance, l'interopérabilité ne peut suffire à elle seule pour répondre à tous les besoins de la problématique de l'ouverture. Spécifiquement, les besoins relatifs à l'intégration de nouveaux éléments dans un système ou la suppression d'une ou de plusieurs composantes d'un système.

1.3.3.2 Ajout et suppression d'éléments dans un système informatique ouvert

L'ajout et la suppression d'éléments dans un système informatique ouvert présentent les propriétés prépondérantes de l'ouverture. L'ajout d'une nouvelle entité dans un système informatique implique que ce dernier dispose de mécanismes d'entrée et d'intégration. L'intégration d'une entité dans un système référence l'aptitude des fonctionnalités de cette dernière d'être explicitement identifiables et que cette nouvelle entité peut interagir avec les autres entités du système. La suppression d'une entité dans un système informatique suppose que celui-ci dispose de mécanismes pour gérer la sortie d'entités. Le fait d'ajouter ou de retirer des entités d'un système peut remettre en cause son bon fonctionnement. Généralement les entités d'un système ont des fonctionnalités complémentaires, le retrait d'une entité peut provoquer une instabilité dans le fonctionnement d'autres entités et par conséquence dans tout le système. De même, l'arrivé d'une nouvelle entité dans un système nécessite une certaine coordination avec celles déjà existantes pour maintenir le bon fonctionnement de ce dernier.

La propriété d'ajout et de suppression d'éléments est présente dans plusieurs applications réelles comme ApacheFelix [10] qui présente une implémentation de la spécification OSGI [132]. ApacheFelix dispose d'une architecture modulaire de composants appelés *bundles* et qui sont des classes Java qui implémentent des services ; qui peuvent être ajoutés ou supprimés avant ou pendant l'exécution. Le *registry* d'une implémentation OSGI détecte automatiquement l'ajout de nouveaux bundles et donc de leurs services ainsi que la suppression de services suite à la suppression de(s) bundle(s) correspondants.

La propriété d'ouverture des systèmes et des applications présente un support naturel pour gérer l'adaptation de ses derniers, l'ajout et la suppression d'entités d'un système met en avant les possibilités d'extension de ce dernier et qui présente un volet important pour la distribution des applications déjà présentée ainsi que pour les possibilités d'adaptation et d'enrichissement des fonctionnalités d'un système. Nous reviendrons en détails sur l'adaptation dans une prochaine section de ce chapitre. Similairement à la propriété d'interopérabilité, les mécanismes liés aux ajouts et aux suppressions d'entités ont besoin de standardisation, sinon les systèmes se trouvent devant l'obligation de fournir explicitement aux systèmes de leurs environnements les types et les représentations des donnés ainsi que les interfaces qui permettent d'un côté d'ajouter et d'intégrer de nouvelles entités et d'un autre côté de supprimer et retirer des entités du système. L'interopérabilité ainsi que les ajouts et suppressions d'entités dans des systèmes informatiques ouverts nécessite encore l'intégration de mécanismes pour assurer la stabilité et la robustesse des systèmes que nous présentons dans la section suivante.

1.3.3.3 Contrôles d'accès dans un système informatique ouvert

Le contrôle présente le troisième volet couvert par la propriété d'ouverture. Ce volet présente une sorte de régulation pour s'assurer du bon déroulement d'échanges entre un système et son environnement. Le contrôle peut être étendu sur le fonctionnement du système pour s'assurer que tous les éléments constituant ce dernier accomplissent correctement leurs rôles, c'est-à-dire s'assurer que les ressources matérielles, les entités logicielles et les intervenants humains agissent correctement en réalisant les traitements, les stockages, l'acheminement et la présentation de l'information. Ce contrôle est très important vu que les interactions ainsi que l'ajout d'entités peuvent avoir des influences malveillantes aux systèmes ainsi qu'aux leurs environnements. Ces influences justifient pour quoi souvent ce point est abordé d'un point de vu sécurité où l'accent est mis sur l'intégrité et la fiabilité des informations échangées entre le système et son environnement ainsi que sur l'accessibilité ou la confidentialité des informations et des ressources. La mise en œuvre de mécanismes permettant d'atteindre ses finalités vont de la gestion des droits d'accès au cryptage des donnés en passant par la surveillance du système et tout autres moyens pouvant intervenir à cet égard.

Dans un système ouvert il est préférable d'avoir une gestion décentralisée du contrôle, c'est-à-dire, plusieurs entités de contrôle interviennent dans cette gestion. En effet, comme toutes solutions centralisées, une gestion de contrôle centralisée dans une seule entité met en cause la robustesse et la fiabilité du système particulièrement dans les cas de défaillance de l'entité de contrôle ou dans les cas de surcharge de cette dernière.

La gestion de l'ouverture dans les systèmes informatiques prend des formes différentes et varie selon les approches et les paradigmes de conception et de développement utilisés. Chacun des aspects de l'ouverture constitue un domaine de recherche dont nous ne pouvons pas faire une étude exhaustive dans cette thèse. Nous nous intéressons aux approches de développement à base de composants logiciels et à celles basées agents et agents mobiles. Nous reviendrons sur l'ouverture dans les approches à base de composants et les systèmes multi agents dans les deux prochains chapitres.

1.4 Adaptabilité et applications adaptables

La problématique de l'adaptation des applications et des systèmes informatiques n'est pas nouvelle, depuis très longtemps cette problématique est abordée et faisait jusqu'à nos jours sujets de plusieurs travaux de recherches. La propriété d'adaptation est aperçue sous plusieurs différents angles et traitées avec une grande diversité de manières et de méthodes qui se sont déployées autour de l'auto-organisation, les systèmes intégrant des facultés d'apprentissage, les systèmes dynamiques, la reconfiguration logicielle, les mécanismes et techniques bioinspirées, les métaphores des organisations collectives, …etc. cette diversité rend la comparaison entre les différents travaux sur l'adaptation une tache très difficile et ardue. Le

besoin d'adaptation des logiciels s'impose d'un côté du fait qu'il est très souvent qu'un logiciel ne corresponde pas totalement à ses spécifications ou les spécifications initiales d'un logiciel ne soient plus pertinentes après plusieurs mois d'utilisation, ou même pendant sa réalisation. D'un autre côté, les environnements dans lesquels les applications sont exécutées présentent, une hétérogénéité importante, une grande variabilité et de nombreuses possibilités d'évolution aussi bien au niveau des moyens d'exécution que des moyens de communication. Cette adaptation d'une application est vue comme l'aptitude de cette dernière de se conformer à des conditions nouvelles ou différentes. L'adaptation correspond au processus de modification d'un système, nécessaire pour permettre un fonctionnement adéquat de ce dernier dans un contexte donné [98]. Le terme adéquat signifie que le système correspond parfaitement à ce que l'on attend dans un contexte précis. Devant ce changement de conditions on distingue deux situations différentes, la première situation présente le cas où les changements sont déjà prévus alors une étape de reconfiguration du logiciel peut répondre aux besoins d'adaptation. Le deuxième cas où le changement des conditions n'a pas été prévu, alors il faut adapter le logiciel en modifiant une partie ou même sa totalité. L'intégration de la problématique de l'adaptation dans le cycle de vie du logiciel semble très utile [147] spécialement pour diminuer les couts de modifications à porter sur le logiciel pour l'adapter.

L'adaptabilité d'une application repose essentiellement sur le fait que cette dernière soit constituée d'entités séparables, c'est-à-dire l'application doit être décomposable et composable. Un système monolithique ne peut être modifié qu'en le remplaçant intégralement ; dans un système construit sous forme d'entités logicielles interconnectées, il est possible de modifier ou de remplacer certaines parties d'une application tout en minimisant les interférences avec le reste des parties. La décomposabilté reflète une mesure de la séparation des éléments qui constituent une application, alors que la composabilité reflète la mesure des capacités d'assemblage entre ces éléments. La granularité de la composition peut être plus ou moins fine et le couplage entre les différentes entités logicielles plus ou moins faible. Les entités constitutives d'une application peuvent prendre différentes formes selon le contexte technique de la réalisation de l'application et peuvent être des modules, des classes, des composants, des aspects, des services ainsi que plusieurs autres formes possibles. L'adaptabilité repose aussi sur une autre caractéristique très importante, il s'agit de la capacité d'une application à s'observer et donc à répondre à des questions sur son état. Cette propriété que l'on appelle *introspection* est essentielle particulièrement pour l'adaptation de systèmes complexes.

L'adaptabilité des applications concerne différentes dimensions, plusieurs études comme [67] et [152] ont permis d'identifier les axes autour de lesquels tourne cette notion d'adaptation qui ce déroule généralement en trois phases : le déclenchement, la décision et la réalisation. Chaque axe représente une réponse pour l'une des questions suivantes : Dans quels buts adapte-t-on une application ? Sur quels éléments de l'application peut porter une adaptation ? À quels moments peut-on adapter une application ? Qui décide d'adapter une application ? Comment réaliser l'adaptation d'une application ? Toutes ces dimensions

peuvent servir de critères pour mesurer le degré d'adaptabilité d'une application et par la suite donner un certain pouvoir pour comparer l'adaptabilité de différents systèmes.

1.4.1 Première dimension : Dans quels buts adapte-t-on une application ?

À partir de plusieurs travaux comme [89] et [67] nous pouvons définir et extraire les raisons pour les quelles une application est amenée à subir à des adaptations. Généralement la communauté des chercheurs dans ce domaine semble d'accord sur la définition de quatre types d'adaptations qui se distinguent par des motivations différentes tout en notant que ces différents types ne sont que des points de vue et qu'ils ne sont pas exclusifs : l'adaptation corrective, l'adaptation adaptative, l'adaptation évolutive et l'adaptation perfective.

- *Adaptation corrective :* le but de l'adaptation est de corriger les erreurs de fonctionnement d'une entité au sein de l'application.
- *Adaptation adaptative :* le but de l'adaptation est de faire évoluer l'application en fonction d'un changement de contexte (nouveau système d'exploitation par exemple).
- *Adaptation évolutive :* ajouter des fonctionnalités à cause de l'évolution des besoins du client.
- *Adaptation perfective :* L'objectif est d'optimiser les performances de l'application. Pour ce faire, on peut améliorer le comportement de certaines parties de l'application pour résoudre une tâche plus efficacement.

1.4.2 Deuxième dimension : Sur quoi porte une adaptation ?

On peut apercevoir un système logiciel comme une architecture composée d'un ensemble d'éléments reliés entre eux. Une adaptation peut donc porter sur un élément, sur une liaison entre deux éléments ou bien sur l'architecture entière. L'adaptation d'un élément peut aller d'un simple paramétrage à une transformation complète. Par un élément on désigne une partie d'un système qui peut s'agir d'une fonction, d'une classe, d'un composant, ou toutes autres entités logicielles. Pour une liaison, l'adaptation consiste à modifier la nature de la liaison ou de l'appliquer sur d'autres couples d'éléments, par exemple, remplacer un lien d'héritage par un lien de délégation. Au niveau de l'architecture, l'adaptation consiste à ajouter, supprimer ou remplacer un élément ou une liaison ; comme il peut s'agir d'un redéploiement qui consiste à modifier la répartition d'un système logiciel sur un environnement d'exécution matériel composé d'un ou de plusieurs sites de déploiement.

1.4.3 Troisième dimension : À quels moments peut-on adapter une application ?

L'adaptation d'une application peut être effectuée à des moments différents et qui recouvrent toutes les étapes de son cycle de vie. L'adaptation peut être une *adaptation statique* et qui concerne dans ce cas que les étapes d'analyse, de conception et

d'implantation. Par exemple, Les modifications des spécifications d'un élément qui est en cours de développement engendrent une adaptation statique de son implantation. Nous parlons d'une *Adaptation semi-dynamique pour* désigner l'adaptation qui peut apparaître à l'étape de déploiement de l'application. Dans ce cas, on s'intéresse au processus d'installation de l'application. Des opérations d'adaptation peuvent avoir lieu pendant que le processus d'installation de l'application est en cours. En effet, lorsque le contexte de déploiement d'une application n'est pas totalement connu à l'avance, il doit être découvert dynamiquement au moment du déploiement [152]. En dernier, l'adaptation est dite *Adaptation dynamique :* puisque elle intervient lors de la phase d'exécution de l'application. Les modifications dynamiques permettent de satisfaire les besoins les plus tardifs. Par exemple, lors d'un changement de contexte d'exécution comme dans le cas d'une faible quantité de batterie sur un dispositif mobile, il est souhaitable d'économiser de l'énergie par arrêter certains services facultatifs, l'adaptation consiste alors à réorganiser l'architecture logique de l'application pour minimiser les aspects gourmands en ressources. Les opérations d'adaptation dynamique doivent être optimisées. Ainsi, l'effort à faire pour optimiser les opérations d'adaptation peut être de plus en plus grand lorsqu'on se place dans des systèmes à fortes contraintes comme les applications temps-réel.

Il est important de noter qu'il existe des applications critiques qui doivent s'exécuter de manière continue car elles doivent être disponibles à tout moment. Dans ces cas, il faut envisager d'adapter dynamiquement ces applications. Il existe encore d'autres cas où les adaptations doivent être faites sans arrêter l'application concernée par les adaptations. Par exemple, dans les cas d'applications dont les environnements d'exécution changent constamment, il est inadapté d'arrêter les applications à chaque modification ; de même pour les applications dont l'arrêt est coûteux pour les entreprises.

1.4.4 Quatrième dimension : Qui décide l'adaptation d'une application ?

L'adaptation d'applications est un processus qui se déroule en plusieurs étapes. Chacune de ses étapes fait intervenir des acteurs particuliers. Plusieurs acteurs humains ou logiciels peuvent intervenir pour déclencher le processus de l'adaptation. Si le concepteur de l'application a anticipé l'adaptation, l'application peut initier sa propre adaptation automatiquement sans intervention humaine. Dans ce cas des entités logicielles par exemple des composants ou des capteurs physiques peuvent jouer ce rôle d'initiateur d'adaptation, ceci devient possible si les critères de déclenchement de l'adaptation sont clairement identifiés et donc facilement automatisables. Ce genre d'adaptation déclenchée automatiquement figure bien dans applications dites *autonomiques* ou auto-adaptatives [120] et les applications sensibles au contexte où par exemple en fonction d'une position géographique un capteur physique peut déclencher le processus d'adaptation. Dans l'autre côté, le déclenchement d'une adaptation est initié par un acteur humain qui peut être l'un des principaux acteurs du cycle de vie d'un logiciel, c'est-à-dire, le concepteur, le développeur et l'administrateur.

1.4.5 Cinquième dimension : comment mettre en œuvre l'adaptation d'une application ?

Il existe plusieurs approches pour réaliser l'adaptation d'une application, les comparaisons entre ces approches peuvent s'effectuées selon plusieurs critères : selon les étapes du processus d'adaptation ; par rapport aux techniques utilisées ; et selon la qualité du processus d'adaptation en mettant l'accent sur les garanties à obtenir en contrôlant le processus d'adaptation.

Les techniques d'adaptation peuvent être basiques comme un simple *re-paramètrage* ou une reconfiguration à condition que l'adaptation soit anticipée. Dans ce cas, le système à adapter est construit selon un ensemble de paramètres prédéfinis dont les valeurs peuvent être spécifiées après la construction du système. Toujours dans le cas d'adaptation anticipée, celle-ci peut être réalisée par la *transformation de code* portant sur le code source ou sur du « *bytecode* » comme il est fait dans les travaux de Hnetynka [77]. Dans l'autre direction où les adaptations sont non anticipées, il existe plusieurs approches qui permettent l'adaptation à un haut niveau. Parmi ces approches on note l'existence d'approche qui se base sur La capacité d'un système à s'observer et donc à répondre à des questions sur son état. Cette propriété est considérée comme essentielle pour l'adaptation de systèmes complexes. La *réflexion* [143] peut servir d'exemple pour ce genre d'adaptation, il s'agit d'une technique permettant à un logiciel de s'auto-représenter et de s'auto-manipuler. Ce mécanisme est utilisé pour programmer de manière générique en manipulant le code de base d'une application. La *programmation par aspects (AOP)* [54] présente une autre alternative pour faire des adaptations de haut niveau, c'est un mécanisme permettant de réaliser des opérations transversales sur le code métier d'une application ou sur son code technique. Cette technique a pour objectif la réalisation d'adaptation évolutive de logiciels [102], elle également utilisée pour l'adaptation d'applications à base de composants.

Les techniques utilisées pour mettre en œuvre l'adaptation aux niveaux d'applications informatiques peuvent aussi être catégorisées par d'autres critères autres que le bas et le haut niveau. Une classification peut se faire selon l'élément sur lequel agit le processus d'adaptation. Selon cette vision, on a des techniques qui agissent sur les liaisons reliant les différent éléments d'une application, et d'autres techniques qui se contentent d'agir sur les éléments ou les composantes de l'application ainsi que d'autres techniques qui agissent sur l'architecture logicielle d'une application.

Les techniques de mise en œuvre de l'adaptation qui situent les modifications au niveau des liaisons sont répertoriées en deux classes, celles où l'adaptation consiste à modifier la nature de la liaison et les techniques où l'adaptation consiste à appliquer la liaison sur d'autres couples d'éléments. Parmi les techniques de la première classe on peut citer *l'interposition*. Il s'agit d'une technique utilisée dans le cas des liaisons dites *Blanches* signifiant des liaisons sur lesquelles on peut effectuer des modifications. Par exemple le

concept de *fabrique de liaisons* proposé dans [49] permet d'adapter les liaisons. Une autre bonne raison pour s'intéresser aux liaisons blanches c'est le pouvoir de contourner le problème d'adaptation des boîtes noires, c'est-à-dire, les éléments intégrés dans une application dont on n'a pas les détails d'implémentation. Pour la deuxième classe on peut citer comme exemple des travaux basés sur la technique de délégation [2] où le travail d'une entité est délégué vers une autre. La mise en place d'une délégation revient à générer une autre liaison, mais celle-ci peut être perçue comme une extension et donc une adaptation de la liaison initiale.

Pour les techniques d'adaptation qui agissent sur les éléments ou les entités du système sans toucher aux liaisons, l'adaptation consiste à modifier l'état d'une entité ou son code. Le changement d'interface sans changement du code est considéré comme du changement de liaisons. Les techniques de *transformation de code* déjà citées appartiennent à cette catégorie de travaux. Ces techniques consistent à modifier directement le code d'une entité. La modification de cette dernière est seulement possible quand celle-ci est blanche. La transformation peut être faite manuellement ou réalisée par des technologies comme le tissage d'aspects [92] dans la programmation dite par aspects et qui vise à permettre une séparation des différentes préoccupations des programmeurs face à la réalisation d'un programme. Dans ce paradigme de programmation, il est possible d'implémenter des comportements s'entrelaçant avec le reste du système d'une manière modulaire. La programmation par aspects s'attache tout particulièrement à permettre d'un côté l'expression séparée des parties techniques et des parties fonctionnelles des applications. D'un autre côté, plusieurs récents travaux se sont intéressés à décomposer également les parties fonctionnelles. L'entrelacement entre les aspects, aussi nommé tissage (weaving), qui s'appuis sur la manipulation des points de jonctions est aux cœurs de plusieurs approches qui présentent la particularité de permettre la manipulation (ajout, retrait ou re-ordonnancement) des aspects de façon dynamique, autorisant ainsi une adaptation en fonction de l'environnement.

Les travaux qui s'intéressent à l'adaptation de l'architecture complète, sont basés sur des procédures de configuration et de reconfiguration. Cette activité consiste en une composition et recomposition d'éléments ou composantes à l'intérieur de l'architecture. Cela peut donc être réalisé soit par retrait, remplacement, paramétrage de certaines entités et/ou liaisons, soit par ajout de nouvelles entités.

1.5 Synthèse sur le contexte du travail

Noun nous somme intéressés dans nos travaux au développement d'applications distribuées, ouvertes et capables de s'adapter en réponse à des besoins applicatifs ou pour être plus conformes avec les environnements d'exécution qui ne cessent de changer constamment. Le développement de telles applications nécessite des moyens qui peuvent fournir de réelles solutions en vue de répondre aux caractères spéciaux imposés sur les applications par les

propriétés de distribution, d'ouverture et d'adaptation. Par le terme moyens nous désignons techniques, approches, méthodes, langages de programmation et tout autres supports qui peuvent servir pour produire des éléments de solutions pour le développement d'applications distribuées, ouvertes et adaptables.

Les propriétés de distribution, d'ouverture et d'adaptation chacune de son côté, imposent des caractères et des contraintes spécifiques sur les applications sujets de développement. La distribution impose que les applications soient implémentées sous formes d'entités ou de composantes qui s'exécutent sur des machines distantes les unes des autres avec une distribution de computation et une décentralisation des ressources et des connaissances. Le développement d'applications distribuées et plus particulièrement leurs conceptions s'articulent sur: la transparence, l'extensibilité, la sécurité, les qualités de service et la gestion des pannes pour maitriser leurs complexité ainsi que les phénomènes émergeants du à la distribution. L'ouverture de son côté impose sur une application d'être situées dans un environnement où elle interagit, communique et faisant des échanges aves d'autres systèmes de son environnement ; d'obéir à des règles d'interopérabilité et d'avoir l'aptitude de changer les frontières par l'intégration de nouvelles composantes ou par retirer des composantes parmi celles qui la constituent. La prise en compte de l'adaptabilité des applications comme une préoccupation essentielle influence fortement leurs développements, particulièrement, si la problématique de l'adaptation est intégrée dans le cycle de vie du logiciel. Cela permet de diminuer les coûts de modifications à porter sur le logiciel pour l'adapter. L'adaptation correspond au processus de modification nécessaire pour permettre un fonctionnement adéquat d'une application dans un contexte donné. Les modifications à porter sur une application en vue de son adaptation peuvent être situées dans un espace définit par les cinq repères suivants : Dans quels buts adapte-t-on une application ? Sur quels éléments de l'application peut porter une adaptation ? À quels moments peut-on adapter une application ? Qui décide d'adapter une application ? Comment réaliser l'adaptation d'une application ? Devant le changement de contexte on distingue deux situations différentes, la première situation présente le cas où les changements sont déjà prévus alors une étape de reconfiguration du logiciel peut répondre aux besoins d'adaptation. La seconde situation présente le cas où le changement des conditions n'a pas été prévu, alors il faut adapter l'application en modifiant une partie ou même sa totalité.

Les propriétés de distribution, d'ouverture et d'adaptation ont été abordées soit séparément les unes des autres soit abordées toutes ensembles par les chercheurs en génie logiciel, en systèmes distribués et même en intelligence artificielle selon plusieurs différentes manières et sous différents angles de visions où à chaque fois l'accent est mis sur un aspect particulier de la distribution, de l'ouverture ou de l'adaptabilité. Le développement d'applications distribuées, ouvertes et adaptables est abordé dans plusieurs approches appartenant à des paradigmes de développement différents tel que les architectures logicielles [103], les services [112], les composants [164], les agents [52], les agents mobiles [18], les systèmes multi-agents [52], les approches connexionnistes basées sur l'émergence [90] ou

plusieurs autres paradigmes, ainsi que dans les approches dans lesquelles plusieurs de ces paradigmes sont combinés. Chacun des paradigmes de développement propose des supports qui peuvent être exploités pour développer des éléments de solutions pour un ou plusieurs problèmes relatifs à la distribution, l'ouverture ou l'adaptation. Comme exemples, les composants logiciels [164] et les systèmes multi-agents [52] proposent la structuration du logiciel sous forme de combinaisons de plusieurs entités logicielles ; cette structuration présente un support naturel pour la distribution des applications. Les mécanismes de reconfiguration dynamique d'architectures logicielles qu'on trouve dans plusieurs approches centrées sur l'architecture logicielles peuvent servir de support pour mettre en œuvre des applications adaptables. L'interopérabilité qui présente l'un des volets de l'ouverture trouve des supports dans les interactions agents – agents qui sont abordées avec le développement de langages de communication comme KQML [55] et ACL [56] et qui permettent aux agents de pouvoir échanger des messages tout en conservant leurs propres architectures. Les exemples de supports offerts par les différentes approches et paradigmes sont encore nombreux.

Nous nous sommes intéressé dans nos travaux aux supports offerts par les approches de développement à base de composants logiciels [164] et ceux offerts par les approches à base d'agents, particulièrement mobiles. Nous envisageons trouver une bonne combinaison entre ces deux paradigmes pour qu'on puisse proposer de bonnes solutions pour la majorité des points sur lesquels s'articule le développement d'applications distribuées, ouvertes et adaptables.

1.6 Conclusion

Dans ce chapitre, nous avons tenté de faire un aperçu sur les applications distribuées, ouvertes et adaptables ainsi que les difficultés à rencontrer lors du développement de telles applications. Dans cet aperçu ont été identifiés tous les éléments autour desquels s'articule le développement d'applications distribuées, ouvertes et adaptables ainsi que toutes les particularités imposées par les propriétés de distribution, d'ouverture et d'adaptation. Les composants logiciels, les agents, les agents mobiles et les systèmes multi agents se présentent comme solutions et contribuent chacun de son côté dans le développement d'applications distribuées, ouvertes et adaptables. Dans les deux chapitres qui suivent nous présenterons l'approche de développement à base de composants et le développement basé agents en étudiant les possibilités offertes par chacune de ces approches de développement et faisant une analyse comparative entre ces deux paradigmes importants pour arriver en fin à chercher les fertilités croisées et les possibilités de combinaisons des deux approches pour proposer des solutions pour le développement d'applications distribuées, ouvertes et adaptables dans les quelles nous pouvons tirer profit des importants points forts de chacune des deux approches.

Chapitre II

Composants logiciels et applications à base de composants

Chapitre II

Composants logiciels et applications à base de composants

2.1 Introduction

Depuis des années le monde à connu une énorme utilisation des logiciels dans tous les domaines tel que l'industrie, l'enseignement, l'administration et même dans plusieurs aspects de la vie quotidienne. Cela se traduit en une augmentation des exigences sur la production du logiciel, on demande de plus en plus des logiciels fiables, flexibles, robustes, adaptables, mieux exploitables et que l'on peut facilement installer et déployer. Cette demande grandissante du logiciel ainsi que l'augmentation sur le plan des exigences de qualités ont provoqué une grande complexité à la fois au niveau des logiciels mêmes qu'aux niveaux des processus de développement de ces derniers. Comment vaincre cette complexité ou au minimum la réduire ? Comment adapter rapidement des logiciels face aux changements ? Et finalement, comment prendre en compte l'évolution du logiciel dés son développement ? Ces points présentent des défis très importants pour les développeurs de logiciels et de systèmes informatiques de manière générale.

La réutilisation présente une clé de solutions pour ces problèmes. L'idée de réutiliser des logiciels en tous ou en partie est très ancienne. Malgré quelques succès, la réutilisation n'est pas devenue une force motrice pour le développement du logiciel. Dans plusieurs approches la réutilisation n'est pas unie avec le processus de développement tout en constatant l'absence de définitions exactes de ce qui est réutilisable d'un côté et l'absence de formalisations du comment introduire des changements aux éléments réutilisables d'un autre. L'approche de développement par composants (*component-based development (CBD))* rétablie l'idée de réutilisation en introduisant de nouveaux éléments. Dans cette approche le logiciel est construit par assemblage de composants développés auparavant et préparés pour être intégrés. Cela influence très considérablement sur la gestion de la complexité, l'augmentation de la productivité et l'amélioration des qualités des logiciels.

Les développeurs du logiciel ainsi que leurs clients ont attendu beaucoup du CBD, mais les expériences ont montré que le développement à base de composant exige une approche systématique pour se concentrer sur les aspects du composant pendant le développement de

logiciels [44]. Les techniques du génie logiciel traditionnel doivent être ajustées à la nouvelle approche et de nouvelles procédures doivent être développées. Le génie logiciel à base de composants (*Component-based software engineering* (CBSE)) est reconnu comme nouvelle sous-discipline du génie logiciel où principalement on s'intéresse à fournir des supports pour le développement de composants autant qu'entités réutilisables, des supports pour l'assemblage de composants ainsi que des supports pour la maintenance et la mise à niveau des applications à base de composants par remplacements où personnalisations de composants [44].

Le présent chapitre fait l'objet d'étude et de présentation de l'approche de développement à base de composants logiciels ainsi que la majorité des notions et des concepts en relation avec le concept "composant".

2.2 Composants logiciels & génie logiciel basé composant

Il est très importants de clarifier les concepts liés au CBSE afin d'enlever les ambigüités causées par les différentes définitions des concepts proposées par les auteurs et qui représentent leurs différentes compréhensions dans différentes situations. À la première vue, le développement de logiciel à base de composants paru comme une extension du développement orienté objet, alors que plusieurs arguments comme la granularité, la composition et le déploiement ainsi que le processus de développement permettent de distinguer clairement les composants des objets. Dans cette section nous présentons les concepts de composant, interface, contrat, patterns, et les Framework qui présentent des concepts très importants liés au CBSE.

2.2.1 Différentes définitions du composant

D'une manière simple nous pouvons considérer un composant schématisé dans la figure 2.1 comme une unité réutilisable pour la composition et le déploiement. Dans la littérature plusieurs définitions ont été attribuées à ce concept qui est au cœur du CBSE. Nous présentons quelques définitions qui nous semblent pouvoir couvrir toutes les manières selon les quelles ce concept peut être aperçu.

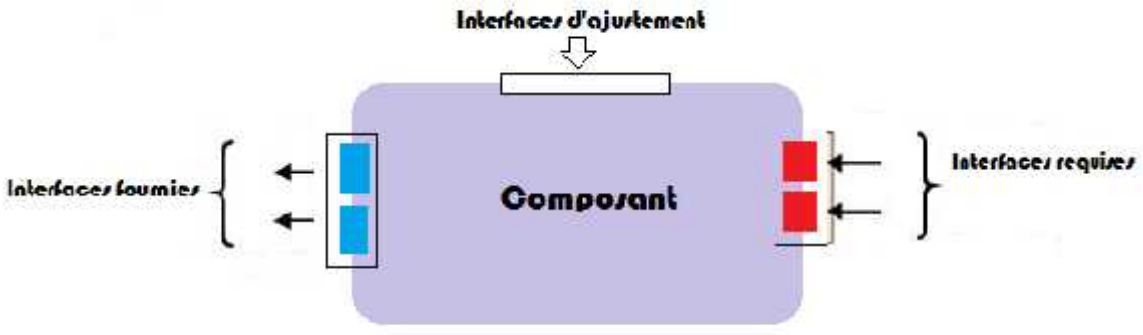

Figure 2.1 : *Composant logiciel*

Szyperski dans [164] définit un composant comme *« Une unité de composition avec des interfaces spécifiées contractuellement et seulement des dépendances explicites vis à vis de son contexte. Un composant logiciel doit pouvoir être déployé indépendamment et fait l'objet de composition par des tiers. »*

En analysant cette définition, on peut constater les points suivants: le composant communique avec son environnement à travers des interfaces, par conséquent celles-ci doivent être spécifiées clairement tout en encapsulant l'implémentation à l'intérieur du composant. Cette séparation entre spécification et implémentation n'est pas similaire à celle que nous pouvons voir dans certains langages de programmation tel que ADA où les déclarations sont séparées des implémentations, ou dans des langages de programmation orientés objet où les définitions des classes sont séparées de leurs implémentations, la différence réside dans les besoins d'intégration du composant dans une application, l'intégration du composant doit être indépendante de son cycle vie. En d'autres termes, il n'est pas nécessaire de recompiler une application quand on ajoute un nouveau composant. Pour que le composant peut être *déployé indépendamment*, il est nécessaire de faire une distinction claire entre le composant et son environnement ainsi que entre le composant et les autres composants.

Une autre définition selon laquelle il est possible d'utiliser le concept de composant dans des niveaux d'abstractions autres que le niveau d'implantation a été proposée dans [82] où le composant est défini comme *« Une unité de conception (de n'importe quel niveau d'abstraction) identifiée par un nom, avec une structure définie et des directives de conception sous la forme de documentation pour supporter sa réutilisation ».* Cette définition est proche de celle proposée dans [44] *«Un composant logiciel fusionne deux perspectives distinctes : entant qu'implémentation un composant peut être déployé et assemblé dans un système ou un sous système ; d'un point de vue architectural le composant exprime les règles de conception imposées par un modèle standard de coordination entre les composants… »* Dans cette définition on a deux aspects physiques différents à apparaître : le composant d'implémentation qui est un programme ou une partie de programme, et le composant de conception qui est un modèle.

Selon D'Souza et Wills [47] «un composant est une partie réutilisable d'une application, il est développé indépendamment et peut être combiné avec d'autres composants pour construire des unités plus grandes. Un composant peut être adapté mais il ne peut pas subir à des modifications ». La spécificité de cette définition réside dans le fait qu'il est noté en clair qu'un composant ne peut pas être modifié.

Brown [29] propose la définition suivante : *« Un composant logiciel est un élément logiciel conforme à un modèle de composants et peut être indépendamment déployé et composé sans modification selon un standard de composition.»* en plus des aspects qui apparaissent des les autres définitions, dans cette définition il mentionné qu'un composant

doit être conforme à un modèle de composant (voir les modèles de composant dans la section 2.4). Cette définition est très proche de la vision de l'industrie qui est différente de la manière avec laquelle les composants sont vus dans les milieux académiques.

Pour conclure, nous revenons au point de départ. Il existe plusieurs définitions pour le concept composant, dans chacune l'accent est mis sur un ou plusieurs aspects. Tous le monde est d'accord sur quelques points, le composant est une unité de composition, il doit être spécifié pour être composé avec d'autres composants ou intégré dans des applications ; les composants sont réutilisables, la réutilisation dans le génie logiciel à base de composant est différente de celle que nous pouvons trouver dans la technologie orienté objet ou d'autre technologies liées au génie logiciel traditionnel. Cette différence réside dans le fait qu'un composant peut être utilisé au moment de l'exécution « *run time* » sans le besoin de recompilation, le deuxième argument de cette différence apparait parce que le composant détache ses interfaces de son implémentation ce qui permet la composition sans être obligé de savoir aucun détail sur l'implémentation du composant.

2.2.2 Interfaces de composant

Une interface d'un composant est définie comme la spécification de ses points d'accès [164]. Ces points sont utilisés par les clients du composant pour accéder aux services fournis par ce dernier. Une interface ne fourni aucun détail d'implémentation de ses opérations. Au lieu de cela, dans une interface on se contente de nommer une collection d'opérations et fournir des descriptions et des protocoles pour ces opérations. Ce mécanisme augmente l'adaptabilité du composant ainsi que ses performances dans le sens où nous pouvons ajouter de nouvelles interfaces et implémentations sans modifier les implémentations existantes ou même remplacer la partie implémentation sans reconstruire le système.

Les interfaces servent aussi pour permettre la personnalisation d'un composant. Les interfaces définies dans des technologies standardisées peuvent exprimer des propriétés fonctionnelles dans lesquelles sont inclues : les signatures des opérations ainsi que les comportements des composants. La majorité des techniques pour la description des interfaces comme les langages de définition d'interfaces (IDL) [68] ne mettent l'accent que sur la partie signature (que les aspects syntaxiques). Ces techniques soufrent aussi de certaines incapacités pour décrire les propriétés non fonctionnelles tel que l'exactitude, la précision, la disponibilité, la sécurité, et la latence. Les interfaces d'un composant sont partitionnés en : les interfaces fournies qui décrivent les services fournis par le composant et les interfaces requises qui spécifient les services requis par le composant.

2.2.3 Contrats

Les contrats [113] trouvent leurs raisons d'exister par le besoin de décrire le comportement global d'un composant et par la limitation de la majorité des techniques de descriptions d'interfaces comme les IDL [68] qui ne traitent que les signatures selon une

vision purement syntaxique. Les contrats [113] permettent d'avoir des spécifications plus précises pour les comportements des composants.

Bertrand Meyer est parmi les premiers qui ont introduit les contrats dans le développement des systèmes, il mentionne dans [113] qu'un contrat précise les contraintes globales (invariantes) qu'un composant doit maintenir.

Pour chaque opération le contrat précise les pré-conditions que les clients doivent satisfaire ainsi que les post-conditions promises en retour par le composant. L'ensemble des pré-conditions, post-conditions et les invariants à maintenir constituent la spécification du comportement d'un composant.

En plus de leurs utilisations pour spécifier les comportements des composants individuellement, les contrats sont utilisés aussi pour spécifier les interactions entre un ensemble de composants. Cette spécification concerne : l'ensemble de composants ; les rôles de chaque composant à travers un ensemble d'obligations ainsi que leurs types ; et un ensemble d'invariants à maintenir par les composants. L'utilisation des contrats dans ce cas favorise la réalisation ainsi que le raffinement d'unités logicielles à base de composant de grande granularité, cela est du aux facteurs suivants :

- Les contrats permettent aux développeurs du logiciel d'isoler et de spécifier explicitement avec un niveau élevé d'abstractions les rôles de tous les composants dans des contextes particuliers.
- La présence de plusieurs contrats permet de modifier indépendamment les rôles de chaque composant, ou de faire des extensions des rôles.
- De nouveaux contrats peuvent être obtenus par l'association de différents participants avec différents rôles.

Puisque un composant peut participer dans plusieurs contrats son comportement global peut être assez complexe. Plus encore, les contrats spécifient les conditions selon lesquelles les composants interagissent avec d'autres composants en termes de pré et post conditions.

2.2.4 Patterns

Le concept de patterns [4] a été introduit à la fin des années 70. Un pattern défini une solution récurrente pour un problème récurrent. Dans un pattern en capture les solutions d'un problème, les solutions fournies ne sont pas des théories ou des abstractions de principes et de stratégies. Les patterns décrivent en détail les relations entre les structures du système et les mécanismes.

On peut se poser la question sur la relation entre les deux concepts, patterns et composant. On peut dire d'un côté que les design patterns peuvent être utilisés pour la conception et la documentation d'un composant, spécialement les composants dans lesquels les unités réutilisables doivent être identifiées. D'un autre côté les composants en tant qu'entités réutilisables peuvent être considérés comme des implémentations de quelques

design patterns. Les designs patterns peuvent servir comme moyen pour décrire les détails d'implémentation de bas niveau d'un composant à la fois pour son comportement et sa structure. Les patterns peuvent aussi décrire les relations entre des composants dans le contexte d'un langage particulier de programmation.

Les design patterns peuvent aussi être utilisés pour la description de composition, c à d décrire des assemblages qui associent plusieurs composants.

2.2.5 Framework

Dans cette section, nous essayons de mettre en clair la relation entre les Framework et le génie logiciel à base de composant (CBSE) et répondre à la question comment utiliser les Framework dans le CBSE ?

Les Framework peuvent être définies selon plusieurs points de vue à différents niveaux d'abstractions. Les Framework peuvent être considérés comme de réutilisables conceptions dans lesquelles la conception consiste à représenter le système à concevoir en classes abstraites et en interactions entre leurs instances. Les Framework peuvent aussi être définis comme des squelettes d'applications qui peuvent être personnalisés par les développeurs [87]. D'un point de vue architectural, un Framework représente une microarchitecture qui fournit un modèle (Template) incomplet pour un système dans un domaine donné [87].

L'idée principale autour de laquelle le concept de Framework tourne c'est que l'on peut extraire des résultats abstraits de l'effort de conception d'un système. Ces résultats peuvent être utilisés dans d'autres situations. Les Framework font la description de ce que peut être utilisé avec quelques modifications dans des situations similaires dans le même domaine ou dans le même espace de problème.

Le CBSE propose de développer des logiciels par « assembler des pièces ». Dans de telles situations il est essentiel d'avoir un contexte dans lequel les pièces peuvent être utilisées. Les Framework peuvent jouer le rôle de ce contexte. Les Framework peuvent jouer le rôle d'une carte mère avec des slots vides auxquels des composants peuvent être insérés dynamiquement pour créer une instance fonctionnelle. La contribution principale des Framework réside dans le fait que ces derniers forcent les composants à achever leurs tâches à travers des mécanismes contrôlés par les Framework. Le visuel studio de Microsoft [117] est un bon exemple où les formes sont construites par ajouter des composants (contrôles) sur une forme initialement vide ; le développeur ajoute des comportements aux contrôles à travers une interface de messages et d'évènements échangés entre les composants.

2.2.6 Récapitulation

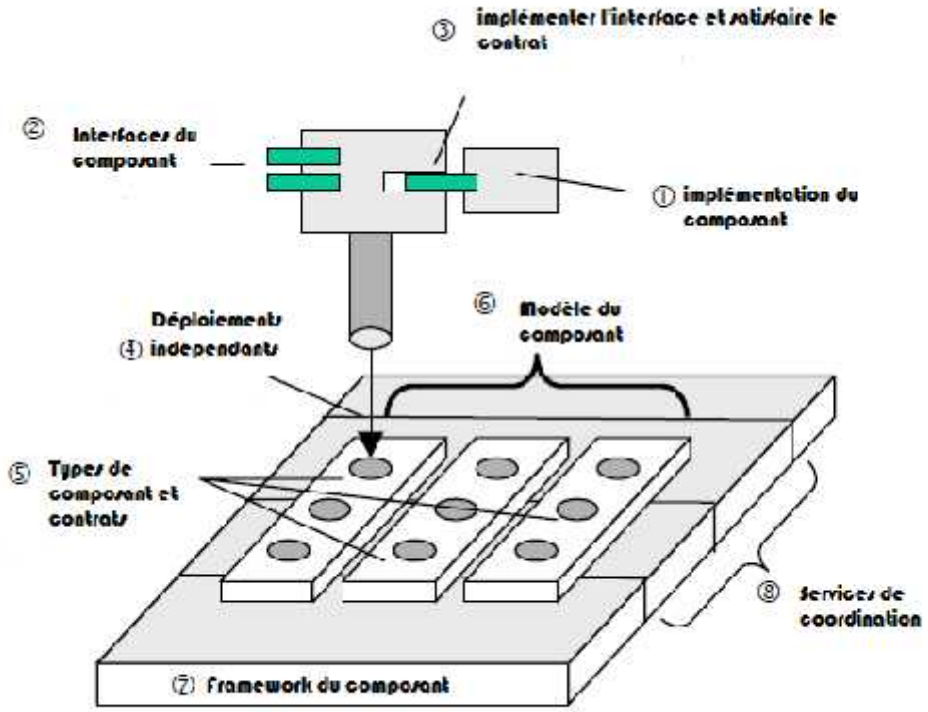

Figure 2.2 : *Relations entre composant, interface, contrat, patterns et Framework* [2.2]

Les relations entre les concepts de composant, interface, contrat, patterns, et les Framework sont schématisées dans la figure 2.2 tirée de [44].Un composant (1) est une implémentation logicielle réalisant un ensemble de services décrits par une ou plusieurs interfaces (2). Des contrats (3) décrivent des conditions et des obligations que les composants doivent respecter. Les contrats assurent que des composants développés indépendamment obéissent à des règles d'interaction, et peuvent être déployés dans un environnement de compilation et d'exécution standard (4). Les composants ont des types(5). Un ensemble de types de composants et leurs interfaces ainsi que des patterns de conception décrivant les interactions entre les différents types de composants forment un modèle de composant (6). Un Framework d'applications (7) fournit un ensemble de critères de déploiement et d'exécution (8) comme support pour les modèles de composants.

2.3 Spécifications des composants logiciels

Les composants logiciels sont accessibles qu'à travers leurs interfaces. Celles-ci doivent fournir toutes les informations nécessaires pour leurs utilisations et leurs déploiements dans différents contextes tout en cachant tous les détails des opérations que les composants implémentent. La spécification d'un composant revient en la spécification de ces interfaces, cette spécification regroupe la définition des opérations du composant ainsi que les dépendances aux contextes, ce que veut dire les aspects fonctionnels du composant ainsi que ses propriétés non fonctionnelles. Toutes ces informations sont utiles aussi pour les développeurs des composants dans le sens où la spécification du composant peut servir comme cadre pour la définition abstraite de sa structure interne.

2.3.1 Spécifications fonctionnelles des composants

Les spécifications des composants concernent les aspects fonctionnels ainsi que les aspects extra fonctionnels du composant. Puisque un composant est visible qu'à travers ses interfaces la spécification de ce dernier revient en une spécification de ses interfaces, cette spécification doit définir d'une manière précise et complète toutes les opérations ainsi que toutes dépendances aux contextes. Les techniques de spécifications utilisées pour décrire les aspects fonctionnels se devisent en deux classes. Des techniques limitées en ce que est appelées spécifications syntaxiques et des techniques qui permettent de décrire des aspects concernant la sémantique de ce que fait le composant.

2.3.1.1 Spécifications syntaxiques

Les composants logiciels implémentent et fournissent un ensemble d'interfaces ou de types. Chaque interface regroupe un ensemble d'opérations où chaque opération porte un nom et a un ensemble de paramètres en entrée et \ou en sortie. Les techniques de spécification syntaxiques associent des types pour tous ces éléments. Certaines techniques permettent de spécifier si le composant a besoin d'interfaces qui sont implémentées par d'autres composants. Les technologies tel que COM [114], ou CORBA (Object Management Group's Common Object Request Broker Architecture) [125] et les JAVABeans de Sun [161] permettent de développer et de déployer des composants réutilisables où les spécifications sont principalement syntaxiques. Pour les deux premières technologies des langages de définition d'interfaces IDL [68] sont utilisés alors pour les JAVABeans les interfaces sont décrites en JAVA.

La figure 2.3 illustre un exemple [44] de contenu d'un fichier de spécification d'un composant COM [114]. Deux interfaces sont spécifiées contenant trois opérations qui

constituent les fonctionnalités d'un composant vérificateur orthographique simple. Les deux interfaces héritent de l'interface COM standard IUNKOWN. Toutes les opérations retournent une valeur de type HRESULT pour indiquer un échec ou un succès de l'exécution de l'opération. Dans cette spécification on a une seule instance du composant associée à deux instances d'interfaces. La première interface est associée à une seule opération, cette dernière à son tour est associée à une seule instance de paramètre d'entrée et deux instances de paramètres de sortie.

Nous pouvons constater que cette spécification peut fournir l'information sur les points suivants : quelles sont les opérations fournies, quel est le nombre de paramètres et quels sont leurs types ; mais aucune information sur l'effet de l'invocation d'une opération n'est fournie. Les spécifications syntaxiques présentent des insuffisances pour traiter la substitution et l'évolution de composants. La substitution consiste en la possibilité de remplacer un composant par un autre, d'un point de vue syntaxique cette substitution n'est possible que si le nouveau composant implémente les mêmes interfaces que le composant à remplacer ou les interfaces du nouveau composant sont des sous-types des interfaces du composant à remplacer. L'évolution d'un composant est vue comme un cas spécial de la substitution.

```
interface ISpellCheck : IUnknown
{
  HRESULT check([in] BSTR *word, [out] bool *correct);
};

interface ICustomSpellCheck : IUnknown
{
  HRESULT add([in] BSTR *word);
  HRESULT remove([in] BSTR *word);
};

library SpellCheckerLib
{

 coclass SpellChecker
 {
    [default] interface ISpellCheck;
    interface ICustomSpellCheck;
 };
};
```

Figure 2.3 : *Exemple de spécification d'un composant COM [2.2]*

2.3.1.2 Spécification des aspects sémantiques

Pour utiliser effectivement les composants, les spécifications syntaxiques ne suffisent pas seules. Il est clair que l'on a besoin d'informations sémantiques sur les composants tel que les codes d'erreurs possibles pour chaque opération, les contraintes sur l'ordre d'invocations des opérations et les combinaisons de valeurs et paramètres que les opérations acceptent.

Depuis que l'idée de développer des logiciels à partir de bibliothèques de composants logiciels produits en masse est annoncée, plusieurs méthodes pour concevoir des composants ont été proposées, la majorité des méthodes ont la tendance d'inclure des informations sémantiques dans la spécification des composants. Par exemple la méthode Design By Contrat [113] ainsi que plusieurs autres méthodes proches associent pour chaque opération des pré-conditions et des post-conditions qui représentent des assertions qui doivent être satisfaites avant l'invocation de l'opération et les assertions que le composant garantie après son invocation respectivement. En plus des pré-conditions et des post-conditions, le contrat liste les contraintes globales que le composant doit maintenir (invariants). Un contrat peut se décomposer en quatre niveaux différents [20] : un niveau syntaxique où le contrat n'est assuré que par les signatures des opérations ; le deuxième niveau : contrat par contraintes, où il est décrit pour chaque service offert les conditions d'utilisation et le résultat attendu ; le troisième niveau : contrat par synchronisation, garantit la bonne marche du système en cas d'utilisation concurrente des interfaces ; et finalement le quatrième niveau qui représente le contrat de qualité de service. Ce type de contrat décrit les conditions d'utilisation de l'interface permettant de garantir la QoS de l'interface. L'utilisation de telles assertions dépond d'un état du composant que ce dernier doit maintenir. Pour cela, pour chaque interface du composant sont ajoutées des informations qui présentent une partie de l'état du composant qui affecte et qui est affectée par l'invocation des opérations de l'interface.

Sous le titre *« UML components »* J.Cheesman et J.Daniels ont publié un ouvrage pour proposer la méthode ***UML component*** [36] dans laquelle ils utilisent UML et OCL [174] (*Object Constraint Language*) pour écrire des spécifications de composants. Les auteurs utilisent la notation UML pour définir le concept du contrat d'assemblage que chaque composant doit remplir pour être intégré facilement dans un système à base des composants existant. La méthode a enrichi la notation d'UML avec des stéréotypes pour désigner la spécification des composants. Par exemples, les stéréotypes suivants ont été proposés: Component specification, Subcomponent, Date Type, Interface Type et Information Type. D'un autre côté, OCL est utilisé pour vérifier et contrôler les pré-conditions et les post-conditions des opérations qui caractérisent le comportement de chaque composant. La méthode propose également six workflows (Capture de besoin, spécification, approvisionnement, assemblage, test et le déploiement). Chacun de ces éléments contient d'autres sous workflows, par exemple le workflow '*spécification*' contient trois activités : identification des composants, interaction entre les composants et la spécification des

composants. Cette méthode est limitée au niveau de conception, aucune explication n'est fournie à-propos de la manière comment interpréter la spécification du composant dans l'implémentation, ou comment vérifier les spécifications du contact au niveau de la compilation.

Une autre méthode dont ***UML component*** est inspirée a été proposée par D'Souza et A. Wills. La méthode est appelée ***Catalysis*** [47]. Dans *Catalysis* un composant est défini comme un package logiciel cohérant, qu'on peut développer pour la construction ou l'extension d'un système plus large. Les composants de la méthode sont abstraits, et définis comme un 'Type'. chaque 'Type' est une classe stéréotypée. Un type présente un comportement dans un domaine. Le comportement externe de chaque 'Type' est défini par ses interfaces.

La méthode ***Component Unified Process*** [144] et qui présente une adaptation du *Processus Unifié* [81] orienté vers les systèmes à base de composants a été aussi une source d'inspiration pour les auteurs de ***UML component.*** La méthode utilise la notation UML pour exprimer le concept de composant durant le cycle de vie de développement des logiciels. L'accent est mis sur l'obligation de l'omniprésence du composant dans tout le cycle de conception du système, car dans la notation classique d'UML on trouve les composants que dans la phase de déploiement. Cette méthode opte pour le développement dans le contexte des architectures dirigées par les modèles [22] en proposant un méta-modèle indépendant d'une plate-forme technologique (PIM). Par la suite, une projection est effectuée vers des plates-formes spécifiques (PSM).

Plusieurs autres méthodes peuvent également servir d'exemples où la spécification des composants et leurs interfaces incluent des informations sémantiques. Nous pouvons citer quelques méthodes comme la méthode ***Business Component Factory*** [75] proposée par Herzum et Sims, la méthode ***KobrA*** [12][13],la méthode ***Rational's Unified Process*** [94]proposée par Jacobson, Booch, et Rumbaug, et finalement la méthode de sélection de perspectives[5] (Select Perspective method).

La notion de substitution de composants dans le cas des spécifications prenant en compte la sémantique n'est pas similaire à la substitution dans les approches qui se contentent des spécifications syntaxiques des composants [44]. La différence réside dans le fait que la condition disant que le nouveau composant doit implémenter les mêmes interfaces que le composant à remplacer n'est plus nécessaire. Pour que la substitution soit possible et d'une manière sûre le nouveau composant doit implémenter les mêmes opérations avec les mêmes signatures que le composant à remplacer implémente tout en assurant les pré-conditions et les post-conditions précitées. En d'autres termes les interfaces du nouveau composant doivent avoir des pré-conditions plus légères et des post-conditions plus fortes que le composant à remplacer. Il faut bien noter que cette vision n'est valide que pour les systèmes séquentiels

contrairement aux systèmes qualifiés de concurrent où la notion de substitution sûre est discutée autrement.

Il est important de noter que l'on constate un manque de représentations formelles pour les différents éléments formants le composant, ce manque empêche l'évaluation de certaines propriétés du composant comme la substitution. Ce manque cause aussi plusieurs autres conséquences comme la difficulté pour sélectionner et choisir des composants sans avoir de représentations riches de ces derniers.

2.3.2 Spécifications des propriétés extra-fonctionnelles des composants

Pour réussir une bonne utilisation des composants on a besoin d'informations supplémentaires autres que les spécifications fonctionnelles. Ces informations peuvent être qualifiées de propriétés non fonctionnelles ou extra fonctionnelles.

Par propriétés extra fonctionnelles nous désignons les propriétés des composants qui déterminent leurs comportements, mais ces propriétés ne sont pas décrites sous forme de fonctions et par la suite elles ne sont pas invocables. Ces propriétés regroupent des contraintes temporelles comme le temps d'exécution et la latence ainsi que des contraintes liées à la fiabilité, la performance, l'efficacité, la synchronisation et la sécurité [44]. Au niveau système, d'autres propriétés peuvent être considérées, ces propriétés sont en relation avec la maintenance, l'adaptabilité et l'utilisation du système. Quand on traite les propriétés extra fonctionnelles, l'un des problèmes les plus importants qui se pose est la détermination des relations entres les propriétés des composants et celles du système. Ainsi que la détermination des propriétés à considérer lors de l'évaluation des composants et lors de leurs compositions.

Depuis longtemps le besoin d'enrichir les spécifications des composants par des propriétés extra fonctionnelles a été reconnu. Plusieurs propositions ont vu le jour présentant des résultats de recherches menées par la communauté des chercheurs en génie logiciel. Nous pouvons citer comme exemple le concept de ***FURPS scheme*** incorporé dans ***Rational Unified Process framework*** [94], ***UML QoS*** [103] et ***Fault Tolerence Profile*** [103]. Parmi les concepts proposés pour décrire les propriétés non fonctionnelles figure le concept ***Credential*** [28] qui est un triplet (Attribut, valeur, crédibilité) où l'attribut représente la propriété, la valeur représente une mesure sur la propriété et la crédibilité indique la manière selon laquelle la mesure est obtenue ; ce concept a été incorporé dans l'approche de construction de système à partir de composant préexistants ***ensemble*** [171]. Nous citons encore les travaux sur les formalismes des propriétés extra fonctionnelles comme la formulation systématique des propriétés non fonctionnelles proposée par Xavier Franch [178] et l'approche proposée dans [62] pour l'intégration de Qos dans le développement du logiciel. Les travaux portant sur les besoins non fonctionnel en génie logiciel ainsi que les descripteurs de propriétés non fonctionnelles peuvent également servir d'exemples.

2.4 Catégorisation des composants logiciels

Les composants peuvent être vus de différentes manières, leurs classifications dépendent de plusieurs points de vue comme le niveau de transparence, le niveau d'abstraction, la spécialisation, et tout autres points de vue permettant une catégorisation des composants. L'utilité des classifications des composants réside dans la possibilité de mieux entourer leurs utilisations.

Selon le degré de transparence signifiant un degré de visibilité et de détails apparents les composants sont répartis en composants boites noires, composants boites blanches et composants boites grises.

- *Composants boites noires* : des composants dont le code binaire est vendu avec un mode d'emploi et des spécifications. la réutilisation d'un composant boîte noire est une sorte de réutilisation d'une mise en œuvre sans tenir compte sur autres choses que son interface et ses spécifications.
- *Composants boites blanches* : Totalement opposé aux composants boites noires, la solution dans le cas des composants boîte blanche est de fournir tout le code des composants. Dans ce cas, on constate une perte au niveau de possibilités de réutilisation dans le sens où il devient très peu probable que les composants peuvent être remplacés par de nouvelles versions car ceux-ci peuvent dépendre de certains détails d'implémentation qui peuvent être sujet de changement dans les nouvelles versions.
- *Composants boites grises* : il s'agit d'une solution intermédiaire entre les composants boites noires et les composants boites blanches. Les composants de cette catégorie dévoilent une partie contrôlée de leurs mises en œuvre dans le cadre de la spécification.

Selon la spécialisation des composants, trois types principaux de composants peuvent être distingués [91] : les composants conceptuels, les composants logiciels et les composants métiers.

Les composants conceptuels : Un composant conceptuel est une solution à un problème conceptuel sous la forme d'un modèle (ou une partie d'un modèle) destinée à être réutilisée [91]. Cette solution peut être spécifiée avec un langage de modélisation comme UML. Les composants conceptuels peuvent être des *composants produits* ou des *composants processus*. Un composant produit est une partie cohérente d'un modèle qui peut être réutilisée avec d'autres composants produits pour assembler un modèle complet. Un produit correspond au but à atteindre lorsqu'on utilise la solution offerte par le composant produit. Par exemple, le patron « composite » de Gamma [60] peut être considéré comme un composant conceptuel produit. Un composant processus est une partie cohérente d'un processus qui peut être réutilisé avec d'autres composants processus pour assembler un processus complet. Un processus correspond au chemin à parcourir pour atteindre la solution offerte par le

composant processus. Par exemple, le patron « revue technique » [6] est un composant conceptuel processus.

Les composants logiciels : Nous pouvons prendre la définition de D'Souza [47] où un composant logiciel est défini comme un paquetage cohérent d'implantations logicielles et qui peut être indépendamment développé et délivré. Il possède des interfaces explicites spécifiant les services offerts et les services requis par le composant. Il peut être composé avec d'autres composants et être éventuellement paramétrable sans modifier son implantation. Il existe d'autres définitions qui s'intéressent à des aspects spécifiques des composants logiciels, certaines insistent sur la séparation du processus de production et de la réutilisation du composant. Cette propriété permet l'industrialisation du processus de production des systèmes divisé en deux sous-processus : le processus de développement pour la réutilisation dans lequel les composants sont développés avec une optique de réutilisation et le processus de développement par réutilisation de composants dans lequel les systèmes sont construits en utilisant des assemblages de composants. D'autres définitions comme celle de *Heineman* [74] introduit la standardisation des composants conformément à des modèles de composants. Ainsi, les modèles de composants permettent l'outillage des approches de développement à composants en fournissant des environnements de développement. Les modèles de composant présentés dans la section suivante servent d'exemples pour cette catégorie de composant.

Les composants métiers: Le concept de composant métier résulte de celui d'objet métier défini par l'OMG (Object Management Group) comme suit : *« Business Objects are representations of the nature and behaviour of real world things or concepts in terms that are meaningful to the enterprise. Customers, products, orders, employees, trades, financial instruments, shipping containers and vehicles are all examples of real-world concepts or things that could be represented by Business Objects »*[126].

Ce que peut être traduit en: les objets métiers sont des représentations de la nature et des comportements des objets du monde réel ou des concepts en termes significatifs pour l'entreprise. Les clients, produits, ordres, employés, affaires, instruments financiers, conteneurs de transport et véhicules sont des exemples de concepts ou d'objets du monde réel représentables par les objets métiers.

Un composant métier peut être vu comme un composant logiciel qui fourni des fonctions dans un domaine métier. Certains spécialistes définissent un composant métier comme une unité de réutilisation de connaissances de domaines d'un point de vue uniquement conceptuel. Par exemple, Casanave propose la définition suivante: *« Un composant métier est vu comme une représentation de la nature et du comportement d'entités du monde réel dans des termes issus du vocabulaire d'une entreprise »* [34].

Parmi les modèles de composants métiers existants nous pouvons citer à titre d'exemple le modèle de composant métier *Symphony* [73].Symphony a pour but de spécifier les différentes phases d'un projet, de définir les tâches de chacun des intervenants et de contrôler les coûts, les délais et la qualité de l'application logicielle produite. Symphony se présente

sous forme d'un guide méthodologique offrant une solution basée sur l'utilisation de composants. Cette démarche s'appuie sur le langage unifié UML et elle est construite autour d'un certain nombre de pratiques. Elle est itérative, incrémentale, orientée utilisateur, orientée composants, pilotée par les cas d'utilisation et adopte un modèle de cycle de vie en Y [9].

2 .5 Modèles et technologies de composants logiciels

Un modèle de composants regroupe un ensemble de conventions à respecter lors de la construction et l'utilisation des composants. Ces conventions permettent de définir et de gérer d'une manière uniforme les composants. Elles couvrent toutes les phases du cycle de vie d'un logiciel à base de composants : la conception, l'implantation, l'assemblage, le déploiement et l'exécution. Le concept de modèle de composant présente un appui essentiel pour le développement, la composition, la communication, le déploiement et l'évolution des composants. Le modèle de composant est un facteur important pour traiter les aspects de l'interopérabilité, l'évolutivité, la maintenabilité, ainsi que de nombreux autres attributs de qualité.

Contrairement aux ADLs (langages de description d'architecture) qui traitent principalement les activités de conception [110], les modèles et les technologies de composants académiques ou industriels comme COM [115] et Java Beans[161] mettent l'accent sur les dernières phases de développement à savoir l'implémentation, le déploiement et l'exécution. Il est tout à fait clair que les ADLs et les technologies de composants adressent des problèmes différents à des niveaux d'abstraction différents. Alors que les ADLs s'intéressent aux problèmes liés à la réalisation et le développement proprement dit, les modèles et les technologies de composants traitent les problèmes liés à la capacité d'analyser le fonctionnement des systèmes ainsi qu'aux supports d'exécution en mettant l'accent sur les aspects pragmatiques.

La majorité les modèles de composants existants offrent des possibilités d'ajouter de nouveaux services ou de nouvelles fonctionnalités aux systèmes logiciels d'une manière transparente. Cela est dû aux possibilités offertes par la majorité de modèles de composants entre autres le pouvoir de définir explicitement les composants et les connexions entre ces derniers ; le pouvoir de définir explicitement les implémentations de composants à partir de codes natifs ; et finalement définir explicitement les propriétés extra fonctionnelles des composants. La majorité des modèles de composants industriels visent certains objectifs communs comme l'augmentation de l'indépendance des codes ; augmenter la transparence des services ; la mise en œuvre de la distribution et la généralisation de service.

Il existe plusieurs modèles et de technologies de composants logiciels comme COM [115], CCM [107], JAVA BEANS[161], .NET [116], FRACTAL[30],et OSGI[128]. Ces modèles de composants se diffèrent les uns des autres principalement dans les points suivant : les concepts clés de chaque modèle ; la manière d'implémenter les composants ; la manière

dont sont réalisés les assemblages et finalement le cycle de vie du composant ainsi que celui de l'application à base de composants. Dans ce qui suit nous présentons en se basant sur ces points quelques modèles de composants logiciels ayant connu des succès importants et de larges utilisations, il s'agit de COM, CCM, JAVA BEANS, .NET et FRACTAL.

2.5.1 Le modèle de composant COM

Les technologies COM (Component Object Model) [115], DCOM [50], MTS [139], et COM+ [139], proposées par Microsoft présentent de réelles tentatives pour augmenter l'indépendance autre les programmes d'un côté et pour vaincre l'hétérogénéité entre les langages de programmation d'un autre. La technologie COM se base sur les interfaces et sur des conventions d'interopérabilité, DCOM est une extension de COM qui traite la distribution, la technologie MTS présente une extension de DCOM qui offre des services de transaction ainsi qu'une certaine persistance, COM+ regroupe les trois technologies en un seul modèle. Dans la technologie COM une interface est vue comme une classe virtuelle c++ et prend la forme d'un ensemble de fonctions et de données sans codes associés. Un objet COM est un code binaire dont la source est écrite dans n'importe quel langage de programmation. Ce code binaire peut être sous la forme d'un exécutable ou d'une DLL (dynamic libraries) qui contient un minimum d'information nécessaire pour la liaison dynamique et l'identification de l'objet COM. La composition des COM est supportée par deux techniques, il s'agit de l'agrégation et la contenance qui signifie qu'un objet COM contient d'autres objets COM. Dans le cas de la contenance l'objet contenant déclare certaines interfaces des objets contenus, l'implémentation de ces dernières se fait par délégation aux objets contenus. L'agrégation est un peu plus complexe, l'objet COM composé expose les interfaces des objets qui le composent comme si l'objet composé implémente réellement ces interfaces. Il est à noter que COM et COM+ sont des modèles de composant basés sur le moment d'exécution uniquement. C'est à dire, ces modèles ne proposent pas des supports pour le cycle de vie entier d'un composant ou d'une application à base de composants.

2.5.2 Le modèle CCM

Le modèle de composant industriel CCM [107] (Corba Component Model) développé par OMG est basé sur des expériences d'utilisation des servies CORBA, les JAVA BEANS et les EJB. Dans CCM l'accent n'est pas mis uniquement sur l'assemblage de composants pour construire une application mais l'accent est mis également sur la conception et le déploiement des composants. CCM présente une solution pour le problème de complexité dans les services CORBA, cette complexité vient de la flexibilité offerte par CORBA aux développeurs et qui impose toujours un nombre important de choix. CCM est le résultat des travaux menés pour définir un modèle de composant dans lequel les standards d'EJB sont généralisés et le nombre de détails à spécifier est réduit ainsi que les risques d'inconsistance. La vue externe d'un composant CCM est une extension du langage IDL de CORBA (langage de définition

d'interfaces de CORBA) où chaque interface est constituée de cinq éléments : facettes, points de connexion décrivant l'aptitude d'utiliser des références fournies par des acteurs externes (*receptacles*), points de connexion émetteurs d'événements, points de connexion pour les quels quelques événements seront transférés, et des attributs pour la configuration du composant. Concernant l'assemblage des CCM, la connexion entre les composants est définie comme la référence d'objets, les composants sont connectés par la liaison des facettes aux *receptacles* et les points de connexion émetteurs d'événements aux points de connexion receveurs. Les connexions sont décrites explicitement dans un fichier XML appelé descripteur d'assemblage d'une manière similaire à celle utilisée dans l'ADL Rapide [103]. Les connexions sont établies par le Framework CCM à l'initialisation. Dans CCM l'implémentation d'un composant consiste en un ensemble de segments présentant chacun un code exécutable implémentant au minimum un port. CCM propose un langage de définition d'implémentation de composant (CIDL) pour décrire les segments, les exécuteurs associés, les types de stockage et les classes de conteneurs. CCM donne de l'importance à la possibilité d'obtenir plusieurs services sans être obligé de faire des modifications au niveau du code tout comme MTS et EJB. Cette approche augmente la réutilisabilité et améliore la maintenabilité d'un côté, et affaiblit la complexité d'un autre côté. Tout comme EJB et CORBA, les composants CCM utilisent des conteneurs pour implémenter les accès aux services en utilisant des patrons de conception collectionnés à travers des expériences dans la construction d'applications basées sur la technologie de l'objet.

2.5.3 Le modèle Java Beans

Le modèle Java Beans(1997) [161] présente une première tentative d'intégrer la notion de composant dans le langage Java. Le modèle met l'accent sur la réutilisation et l'augmentation des capacités de composition statique et dynamique. L'objectif principal de Java Beans est de définir un modèle de composant pour java, ce modèle est utilisé pour créer des composants que l'on peut déplacer et composer ensemble dans une application. L'assemblage peut être réalisé visuellement avec un outil d'assemblage appelé *''builder Tools''*. Aucune solution spécifique pour l'assemblage propre à ce modèle n'est proposée, Java beans est conçu pour supporter plusieurs manières d'assemblage de composants. Un java bean est décrit à travers quatre types de port, *méthodes, propriétés, sources d'événements,* et *les ports d'écoute d'événements.* Ces ports présentent l'interface d'un composant java bean qui est implémenté généralement par un simple objet java, cette implémentation peut être vue comme une encapsulation de l'objet dans un composant. Parfois, des implémentations plus sophistiquées (collection d'objets ou envelopper un objet) sont nécessaires. Un autre volet bien couvert par java Beans concerne le paquetage des composants, ces derniers peuvent être paquetés dans des archives que l'on ouvert par un *'builder tool'* pour obtenir tous les composants archivés.

2.5.4 Le modèle .Net

Le modèle .Net de Microsoft [116] ne présente pas une continuité des modèles COM, DCOM et COM+. Ce modèle est caractérisé par la limitation des interopérabilités binaires. .Net est basé sur un langage d'interopérabilité et d'introspection dénommé MSIL pour designer un langage interne de Microsoft. Ce langage est similaire au Byte code de java. MSIL est interprété par CLR (Common Language Runtime) similaire à la machine virtuelle java. A l'opposé de la vision adoptée par OMG où les informations relatives aux relations entre composants sont séparées de ces derniers dans .Net les choses passent comme en langages de programmation. C'est-à-dire, les programmes contiennent les informations relatives aux relations entre composants. Un composant est décrit dans un descripteur (*manifest*) qui regroupe les informations : les méthodes importées et exportées, les événements, le code, les métadonnées, et les ressources. .Net est basé sur une liaison dynamique pour réaliser les connexions entre les composants pendant l'exécution. En ce qui concerne l'implémentation des composants, ces derniers consistent en modules qui sont des codes exécutables ou des DLLs (dynamic libraries). La liste des modules qui composent le composant est donnée au compilateur pendant la compilation du module principale. A la fin de la compilation le *manifest* est généré dans le même fichier avec le module exécutable principal. Les modules ne peuvent pas être assemblés de modules à leurs tours et ils sont chargés qu'aux moments où ces derniers sont requis.

2.5.5 Le modèle Fractal

Fractal [30] est un framework de composition qui définit un modèle à composants modulaire extensible et générique. Ce framework peut être utilisé avec différents langages de programmation pour concevoir, implanter, déployer et reconfigurer les systèmes informatiques de toute sorte. Ce modèle de composants adopte le principe de séparation des préoccupations dans la conception des composants et des systèmes d'information à base de composants. Le principe de séparation des préoccupations préconise la séparation des aspects et des préoccupations d'un système d'information en plusieurs entités distinctes. Par exemple, implanter les services fonctionnels fournis par le système d'information dans une entité différente des entités qui assurent les aspects non fonctionnels comme la configuration, la sécurité, et la disponibilité, etc. En particulier, le modèle de composants Fractal propose trois niveaux de séparation des préoccupations : Séparation des interfaces et des implémentations où le découplage entre les interfaces et leurs implémentations est assuré par le patron de conception *Pont* [60]. Ainsi, les interfaces et les implémentations peuvent évoluer séparément, ce qui augmente l'adaptabilité et l'évolutivité des composants et des systèmes d'information à base de composants ; la programmation orientée composants, ce deuxième niveau correspond à la décomposition des préoccupations d'implémentation en plusieurs sous

préoccupations de plus petites tailles et composables. Ces sous-préoccupations sont implantées dans des entités séparées appelées composants. Et finalement le niveau d'inversion du contrôle où la configuration et le déploiement d'un composant Fractal sont délégués à une entité extérieure au composant de manière que les composants ne se préoccupent plus de trouver et de paramétrer les composants et les ressources dont ils ont besoin.

2.6 Adaptabilité des applications à base de composants

Nous nous somme intéressé dans nos travaux d'une manière particulière à la problématique d'adaptation logicielle, dans le premier chapitre une vue globale de cette problématique est présentée. Nous essayons dans cette section de projeter les différentes dimensions de l'adaptation explicitées dans le premier chapitre sur le cas d'applications à base de composants.

Première dimension : Dans quels buts adapte-t-on une application à base de composants ?

Les principales raisons pour adapter une application à base de composant sont pratiquement les mêmes pour l'adaptation d'applications en général : l'adaptation corrective, l'adaptation adaptative, l'adaptation évolutive et l'adaptation perfective. Néanmoins, il existe d'autres raisons liées beaucoup plus à l'amélioration de la qualité de l'application. Une application à base de composant peut être adaptée pour des besoins *l'interopérabilité* [28] dans le but de rendre un ou plusieurs composants compatibles avec un grand nombre de contextes. Une autre raison pour laquelle on adapte des composants est de les rendre plus *réutilisables* [19] et donc plus découplés de leurs contextes.

Deuxième dimension : Sur quoi porte l'adaptation d'une application à base de composants ?

Une application à base de composants regroupe naturellement un ensemble de composants et se caractérise par une architecture physique ou logique. Il est donc clair que l'adaptation d'une application à base de composants consiste à porter des modifications ou des changements soit au niveau des composants (les entités qui forment l'application) soit au niveau de l'architecture. Le tableau 2.1 résume la situation en mettant en clair pour chaque sujet d'adaptation les actions de modification qui correspondent.

Sujet d'adaptation		Actions de modification
Architecture	*L'adaptation de l'architecture physique*	Migrer des composants vers d'autres sites d'exécution (un redéploiement de l'application).
	L'adaptation de l'architecture logique	ajouter, supprimer ou remplacer des composants. Modifier les connexions entre composants existants (réassemblage de l'application).
Composant	*Reconfiguration du composant*	Reparamétrer les propriétés configurables d'un composant.
	Réimplantation du composant primitif	Réimplantation interne du composant : changer par exemple une structure de données dans le code, ou le changement d'algorithme d'une méthode ; Réimplantation de la structure externe du composant : par exemple fusionner des interfaces ou les éclater en plusieurs interfaces ;
	Réimplantation d'un composant composite	Réimplantation interne du composant ; Réimplantation de la structure externe du composant ; Réorganisation interne du composite.

Tableau 2.1 : *Sujets d'adaptation d'une application à base de composants.*

Troisième dimension : À quels moments peut-on adapter une application à base de composants ?

On peut adapter une application à base de composants à plusieurs moments de son cycle de vie. Cette adaptation peut être statique, comme elle peut être effectuée au moment du déploiement, on parle dans ce cas d'une adaptation semi-dynamique et l'adaptation peut être effectuée en cours d'exécution, il s'agit d'une adaptation dynamique.

Quatrième dimension : Qui décide l'adaptation d'une application à base de composants ?

Dans le cas d'une adaptation manuelle les opérations d'adaptation sont initiées par des acteurs humains. En fonction du moment dans lequel l'adaptation est recommandée ou nécessaire différents acteurs peuvent intervenir. On distingue le concepteur d'application à base de composants, le développeur et l'administrateur.

Pour une adaptation automatique l'initiateur des opérations d'adaptations peut être aussi un acteur humain ou une entité logicielle interne qui peut être un composant particulier chargé de gérer l'adaptation comme il est présenté dans [78], ces composants chargés de l'adaptation peuvent être conçus selon des design patterns spécifiques comme il est fait dans l'approche présentée dans [32]. L'adaptation peut être déclenchée également par une entité logicielle externe.

Cinquième dimension : comment mettre en œuvre l'adaptation d'une application à base de composants ?

La mise en œuvre de l'adaptation d'une application à base de composants dépend étroitement de l'élément sujet de modifications. Dans le cas où la modification concerne le niveau composant primitif l'adaptation est similaire à l'adaptation d'applications d'une manière générale comme il est expliqué dans la section 1.3.5 du premier chapitre. L'adaptation des composants composites est similaire à l'adaptation d'une application à base de composants du fait qu'un composant composite peut être vu d'une certaine manière comme une application à base de composants. Le dernier cas concerne les modifications qui ont un impact sur l'architecture physique ou logique d'une application. L'adaptation de l'architecture physique consiste à effectuer un redéploiement de composants d'une application ce qui pose des problèmes spécifiques liés à la distribution et dépend de l'infrastructure de déploiement utilisée. L'adaptation de l'architecture logique sur laquelle nous mettons l'accent se résume en l'ajout, la suppression et le remplacement d'un composant ainsi que la reconfiguration des liaisons entre composants.

L'adaptation de l'architecture logique est réalisée à travers un enchainement d'exécutions d'opérations d'ajouts de composants, de suppressions de composants, de substitutions de composants et de reconfigurations de liaisons pour modifier la façon dont les composants de l'application sont assemblés en gardant les mêmes composants. Ces opérations reposent sur certains mécanismes que nous pouvons résumer en : i) le mécanisme d'activation et de désactivation de composant qui consiste à autoriser et interdire respectivement les opérations d'un composant ; ii) le chargement d'un composant qui regroupe les actions permettant d'intégrer dans une application un composant extérieur à celle-ci de sorte qu'il devienne un élément de l'application. À l'opposé, le déchargement d'un composant consiste à supprimer toutes ses occurrences ; iii) les mécanismes de connexion et de déconnexion des composants qui dirigent les dépendances entre les composants ; iv) le mécanisme de configuration de composant qui consiste à modifier les valeurs configurables d'un composant à travers des interfaces de contrôle ; v) le mécanisme de sauvegarde et de restauration d'état des composants pour les utiliser lors des substitutions. Dans la figure 2.4 nous explicitons l'utilisation de ces mécanismes pour réaliser la substitution d'un composant par un autre.

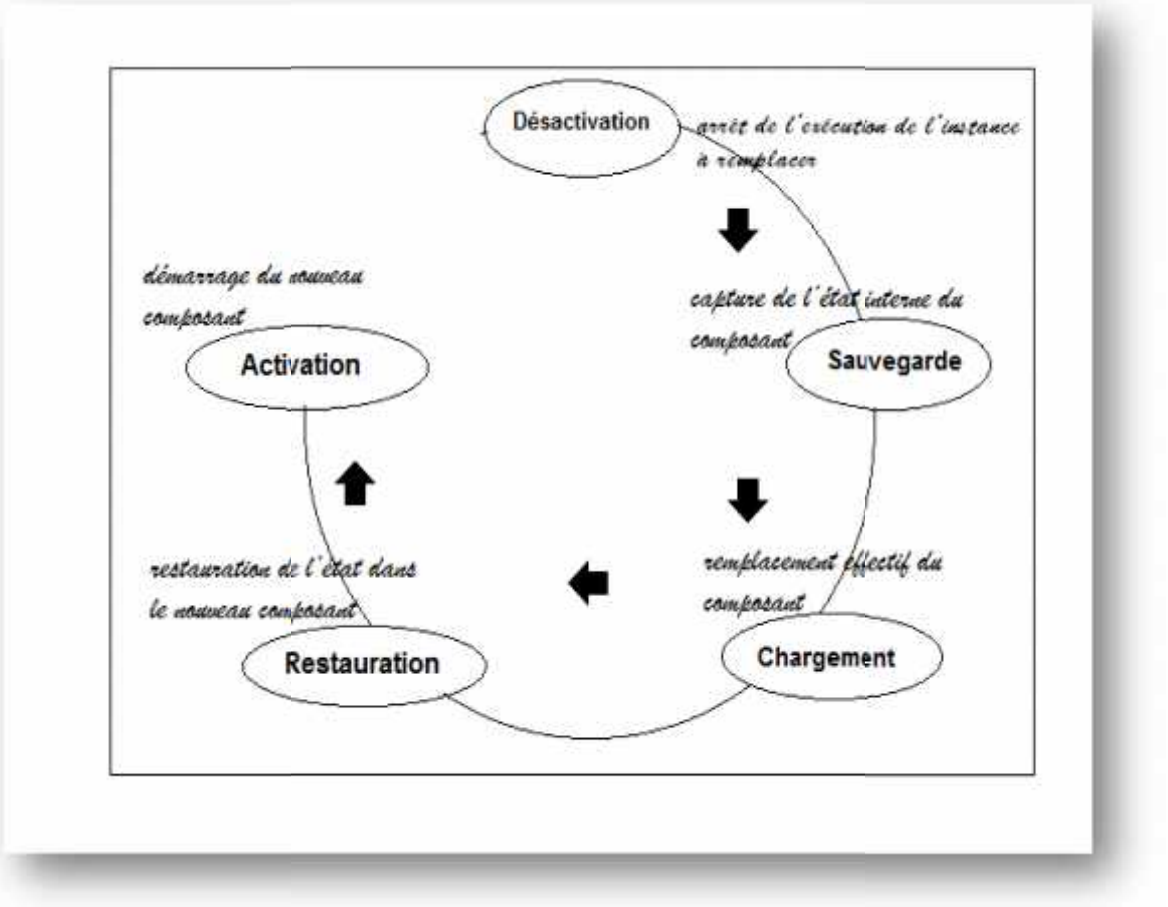

Figure 2.4 : *Mécanismes pour réaliser la substitution d'un composant par un autre.*

Les d'opérations d'ajouts de composants, de suppressions de composants, de substitutions de composants et de reconfigurations de liaisons peuvent être combinées pour donner naissance à d'autres opérations plus complexes [26]. Ces opérations semblent d'une grande utilité dans des cas par exemple où nous voulons remplacer un composant par un assemblage d'autres composants ou l'inverse. La majorité des approches visant le développement d'applications adaptables dynamiquement comme l'approche CommUnity [176] proposent l'utilisation d'opérations d'adaptation complexes et qui sont composées d'opérations simples.

Plusieurs travaux ont été proposés pour traiter la problématique de l'adaptation dynamique au niveau des applications à base de composants. La manière avec laquelle la problématique est traitée diffère d'une approche à une autre. Par exemple dans le domaine des ADLs (Architecture Description Language)[110] l'adaptation apparait sous forme de dynamicité des architectures. Certains ADLs comme Darwin [109] et Rapide [103] permettent d'exprimer des reconfigurations dynamiques des connexions entres les composantes d'une architecture et même d'ajouter, supprimer ou remplacer des composants

et des connexions. Dans les ADLs, l'architecture d'un système est décrite principalement en termes de composants traités en tant que des unités de calcul ou de stockage qui sont dotées d'interfaces fournies et requises, de connecteurs représentants les interconnexions ainsi que les règles d'interaction entre composants et des règles de configurations qui peuvent être décrites sous fourme de graphe de composants interconnectés à travers des connecteurs. En plus des ADLs, il existe des langages pour modifier des architectures abrégés généralement AMLs (Architecture Modification Language)[131] qui représentent des environnements de configuration dynamique des architectures comme [17]. Dans une autre catégorie de travaux la problématique d'adaptations d'applications à base de composant est traitée en tant qu'une problématique de spécifications d'adaptations. Dans cette tendance s'inscrit le modèle proposé dans [111] où les auteurs utilisent des règles exprimées en XML sous la forme d'un contrat qui associe une certaine configuration du contexte d'exécution aux comportements que doit adopter l'application. Dans ce genre de travaux, il s'agit d'exprimer les cas d'adaptation où un comportement est appliqué dès que certaines contraintes de déclenchement de l'adaptation sont satisfaites. Cependant, les travaux basés sur cette idée de règles indépendantes ne peuvent pas apporter quelque chose d'intéressant dans les cas complexes, ces travaux restent applicables que pour des cas simples.

2.6.1 L'adaptation dans quelques modèles de composants

Il existe de nombreux travaux qui traitent l'adaptation d'applications à base de composants en utilisent des modèles de composants particuliers qui peuvent êtres industriels comme EJB ou non industriels où les auteurs proposent leurs propres modèles de composant. L'adaptation d'applications est prise en charge en fournissant une infrastructure d'adaptation particulière. Les adaptations basées sur les modèles de composant SAFRAN [137], Sofa [32], DUCS [23], K-Component [46],OpenRec [175] et CASA [118] peuvent servir de bons exemples .

Le modèle de composant SAFRAN [137] présente une extension du modèle de composant Fractal [30] visant à supporter le développement des composants auto adaptables. Ce modèle de composant est basé sur l'introduction d'une extension réflexive permettant de modifier de manière transparente le comportement d'un composant en fonction de son contexte d'exécution. L'adaptation est traitée en tant qu'aspect dynamiquement dans les applications en utilisant la programmation par aspects sous forme de politique réactive réalisée grâce à un langage dédié (FScript). Dans SAFRAN, d'adaptation consiste à exécuter des actions permettent de réassembler une application, dans le sens qu'un composant composite basé sur le modèle ECA (évènement-Condition-Action). Le déclenchement automatique de ces actions consiste à tester les évènements issus du contexte. En faisant une évaluation de SAFRAN on conclue que l'infrastructure d'adaptation utilisée dans SAFRAN est assez sophistiquée car elle inclut non seulement un modèle de politiques d'adaptation mais aussi un gestionnaire générique de contexte et un langage de reconfiguration. Ainsi, un

concepteur d'applications dispose de tous les outils nécessaires pour définir facilement une application auto-adaptable [67].

DUCS [23] est un modèle à composant supportant la mise à jour des composants de manière dynamique. Le but de DUCS est de fournir une infrastructure pour créer des applications réparties où les composants peuvent évoluer sans arrêter l'application.

L'infrastructure de DUCS est composée de quatre entités. Le plus bas niveau représente les nœuds. Les nœuds sont apparentés à une machine virtuelle (Java). Au dessus de ces nœuds se trouve la base de l'infrastructure fournissant l'environnement d'exécution des composants de l'application. La communication entre les composants utilise cette infrastructure. Au sein de cette infrastructure se trouve le gestionnaire de configuration gérant l'évolution sur les nœuds. Ce gestionnaire de configuration fournit une interface afin de faire évoluer les composants du noeud. Cette évolution est commandée par une « requête de mise à jour ». La dernière entité formant le canevas de DUCS est le gestionnaire d'architecture. Ce gestionnaire gère l'ajout, la suppression et la mise à jour de composants sur tous les nœuds formant l'application. Lors d'une mise à jour DUC propose un support de transfert d'état. À l'aide de fonctions de transfert, l'état d'un composant est réinjecté dans le composant le remplaçant.

K-Component [46] est un modèle à composant au dessus de CORBA ayant pour but de créer des applications auto-adaptables. Ce modèle tend à séparer l'implémentation des composants (en CORBA) de la gestion du dynamisme. Une autre particularité de ce modèle à composant est que les adaptations sont directement effectuées sur l'architecture et reproduites sur l'application. Plus particulièrement, l'application est composée d'un ensemble d'instances de composant et de connecteurs. Elle est directement supervisée par un gestionnaire de configuration. Ce gestionnaire maintient en permanence le graphe des instances de composants et des connecteurs qui composent l'application. La communication entre le gestionnaire de configuration et l'application se fait via des événements d'adaptations émis par l'application ou émis par le gestionnaire de configuration afin de « commander » les reconfigurations. Les reconfigurations sont décrites dans des contrats d'adaptation. Ces contrats sont exprimés avec un ensemble de règles conditionnelles permettant à la fois de reconfigurer l'architecture de l'application en fonction des événements émis par l'application et de définir des contraintes architecturales. La reconfiguration de l'architecture se fait en utilisant des opérations de reconfiguration fournies par le gestionnaire de configuration. K-Component propose donc un modèle à composant intéressant pour créer des applications dynamiques. En effet, il est capable de gérer toutes les sources de dynamisme. De plus, le fait de reconfigurer directement l'architecture de l'application permet d'exprimer les reconfigurations à l'aide de primitives de plus haut niveau.

OpenRec [175] est un modèle à composant où l'adaptation est réalisée par des supports pour la reconfiguration dynamique. OpenRec permet la création d'applications dynamiques ouvertes sans définir un algorithme de reconfiguration au départ mais permet de définir des algorithmes et de les substituer à l'exécution. L'infrastructure d'OpenRec est divisée en trois

couches qui sont elles-mêmes implémentées en utilisant OpenRec. La première couche représente le pilote de reconfiguration qui reçoit les politiques d'adaptation et détermine quand et comment doit être adaptée l'application. Lorsqu'une adaptation est nécessaire, il notifie le gestionnaire de reconfiguration qui en fonction de l'algorithme de reconfiguration utilisé restructure l'application. L'application est donc supervisée par le pilote et le gestionnaire de reconfiguration. La couche application contient des composants et des connecteurs OpenRec, mais également des plug-ins de supervision.

Selon qu'il existe plusieurs modèles et approches à base de composant où chacune propose des mécanismes et des concepts pour supporter l'adaptation des applications à base de composants, il faut mettre en place une certaine mesure de la qualité du processus d'adaptation. Des critères d'évaluation du processus d'adaptation sont proposés dans [67], que nous pouvons résumer en :

- *la cohérence* qui fait référence à la capacité d'empêcher l'introduction d'erreurs de fonctionnement à cause du processus d'adaptation et des problèmes qu'il peut engendrer ;
- *La performance* qui fait référence à la quantité de ressources matérielles utilisées par le processus d'adaptation ;
- *la disponibilité* : qui fait référence à la capacité de garantir que certaines fonctionnalités prioritaires pourront être utilisées durant le processus d'adaptation *;*
- *La simultanéité* qui fait référence à la capacité de coordonner des adaptations qui sont déclenchées ou réalisées simultanément ;
- et finalement *l'ouverture* qui fait référence à la possibilité de configurer/adapter le processus d'adaptation avec des stratégies plus ou moins élaborées.

2.7 Synthèse

A travers l'étude de l'approche de développement à base de composants et l'étude de quelques modèles de composant nous constatons qu'il est d'intérêt de conserver la notion de composant logiciel et d'architecture logicielle dans le développement des applications. Les composants logiciels apportent de la structuration du code de l'application, de la réutilisation et du support pour le déploiement. Le concept de connecteurs utilisés pour inter relier les composants apportent de leurs côté un support permettant de séparer les aspects liés à la mise en œuvre de la communication des aspects métiers de l'application. Les capacités qu'offrent certains modèles de composant en termes de pouvoir de reconfigurer dynamiquement une application présentent de réelles solutions pour la construction d'applications plus adaptables et plus flexibles. La reconfiguration d'une application recouvre les modifications portant sur les éléments constitutifs de l'application ainsi que celles qui portent sur les interactions entre ceux-ci.

Un autre constat que nous avons fait en étudiant quelques approches de développement à base de composants consiste en la redéfinition de langages de description d'architecture pour chaque approche bien qu'il est facilement observable qu'il existe de nombreuses similitudes entre ces langages. Afin de mettre en avant l'interopérabilité entre les approches de construction d'applications à base de composants nous pensons qu'il est intéressant de disposer d'un modèle suffisamment abstrait pour la description d'architecture et commun entre la majorité des modèles de composants et plus particulièrement les modèles académiques.

Il semble aujourd'hui que la majorité des approches de développement à base de composant ainsi que la majorité des modèles de composants se rencontrent bien sur les grandes lignes de la structure du composant avec principalement l'utilisation des notions de ports et d'interfaces. Cet accord sur le plan structure est escorté par une grande variation dans la sémantique des composants ainsi qu'une grande diversité dans les caractéristiques et les objectifs de chaque approche. Devant le nombre important d'approches et de modèles de composants révélant une grande diversité de points de vue, d'objectifs et de caractéristiques, il est important, voire obligatoire, de fixer les axes qui définissent un repère dans lequel nous pouvons situer les approches et les modèles de composants les uns par apport aux autres. Les axes de ce repère représentent les critères selon les quels les approches et les modèles de composants sont évalués. De nombreux critères d'évaluation des approches et des modèles de composants peuvent être définis où un critère peut à son tour définir d'autres sous critères. Les critères qui nous semblent importants et qui se coïncident avec nos objectifs sont :

- *La prise en compte de la dynamique et de l'adaptation de l'application ;*
- *La distance importante reflétant l'abstraction par apport aux plateformes d'exécution ;*
- *La précision dans la description d'assemblages de composants et de leurs interfaces.*

Il est d'une importance extrême de prendre en compte l'évolution de l'architecture logicielle en réponse à la dynamique de l'application. Cette prise en compte ne se résume pas en quelques mécanismes de reconfiguration dynamique même avancés que nous trouvons dans quelques modèles de composants comme FRACTAL [30] ou SAFRAN [137], mais la prise en compte de la dynamique doit aller jusqu'à la spécification de l'évolution elle-même loin de s'appuyer sur des descriptions qui présentent le système sous forme d'images instantanées. Cela ne veut pas dire que la description doit contenir l'ensemble de toutes les configurations possibles, il est clair que ce mécanisme est inefficace si le système doit s'adapter fréquemment. Darwin [109] fait partie des approches qui proposent la description de la dynamique du système. Cela apparait sous forme de mécanismes explicites d'instanciation des composants du système. UML 2.0 [127] à son tour propose par l'intermédiaire des diagrammes de séquences la description explicitement de la création

d'instances de composant ainsi qu'un ensemble d'invariants du système au niveau de la description de l'assemblage. Globalement, la dynamique de l'architecture logicielle est prise en compte que par peu d'approches, même dans celle-ci, l'évolution n'est pas spécifiée et encadrée totalement.

Sur le plan abstraction vis-à-vis de la plate-forme d'exécution, les techniques de description et notamment les ADLs doivent réagir positivement à cet égard. Les ADLs en particulier doivent proposer des mécanismes permettant de profiter de la description d'architecture du système dans la production de l'application et de la génération de code vers différentes plates-formes composants. Cependant, dans certains ADLs l'objectif des spécifications se limite à l'analyse du système à réaliser. De ce fait, ils ne proposent pas de mécanismes de projection de code vers des plates-formes d'exécution. D'autres approches comme celles suivies dans Darwin [109] et SOFA [32] se limitent à proposer une unique plate-forme d'exécution pour leur modèle de composant malgré que les modèles de composant qu'elles proposent et plus particulièrement celui de SOFA sont suffisamment abstraits pour qu'on puisse faire des projections vers d'autres plates-formes à composants. Les techniques qui ne séparent pas la description de l'architecture de l'implantation des services fournis et requis par ses composants mènent à des situations compliquées et très difficiles à gérer si les développeurs veulent par besoins générer du code vers une autre plate-forme à composant. L'approche suivi dans Fractal [30] définit un modèle de composant abstrait avec l'existence de plusieurs implantations permettant de choisir une plate-forme cible en fonction du contexte du système à réaliser. Par exemple, une implantation en C appelée *think* [51] permet de construire des applications embarquées à base de composants. Alors que pour la construction de système d'information d'entreprise plusieurs implantations en Java comme *Julia* [31] ou en c++ comme *PLASMA* [97] sont disponibles, ce que fait de Fractal un bon modèle d'abstraction vis-à-vis de la plate-forme d'exécution.

En ce que concerne le critère de la précision dans la description d'assemblages de composants et de leurs interfaces, ces propriétés permettent d'analyser l'assemblage de composants et de statuer par la suite sur le niveau de qualité de service de l'application. Comme les activités d'analyse d'une description d'architecture s'appuient principalement sur la description des interfaces de ses composants, beaucoup d'approches et d'ADLs ont fourni une définition étendue des interfaces de composant pour disposer d'un maximum d'information sur le composant lors de l'analyse de l'assemblage. Parmi ces définitions étendues des interfaces de composants, des approches comme SOFA proposent au niveau des interfaces des composants et des rôles des connecteurs, une description des messages entrants et sortants de ces composants ou de ces connecteurs. D'autres approches se caractérisent par leurs focalisations sur les contrats [113]. Ces derniers se diffèrent des IDL [68] qui ne traitent que les signatures d'interfaces selon une vision purement syntaxique. Les contrats contiennent des informations réparties sur quatre niveaux différents [20] : un niveau syntaxique dans lequel Le contrat n'est assuré que par les signatures des opérations ; le deuxième niveau

concerne les contraintes, où il est décrit pour chaque service offert les conditions d'utilisation et le résultat attendu ; le troisième niveau traite la synchronisation pour garantir la bonne marche du système en cas d'utilisation concurrente de l'interface ; et finalement le quatrième niveau qui représente le contrat de qualité de service. Ce dernier niveau du contrat décrit les conditions d'utilisation de l'interface permettant de garantir la QoS de l'interface. Les approches qui ne proposent pas d'abstraction du comportement du composant proposent l'utilisation de quelques mécanismes d'extension comme les annexes et les propriétés qui permettent d'ajouter de l'information pour caractériser le composant. Grâce à ces mécanismes les différents niveaux de contrats peuvent être décrits. Cependant, pour le moment ces mécanismes sont principalement utilisés pour des propriétés de qualité de service. Des méthodes comme ***UML component*** [36] dans laquelle les auteurs utilisent UML et OCL [174] proposent de nombreux mécanismes au niveau du langage pour enrichir la définition de l'interface d'un composant. Si OCL est maintenant convenablement adopté et intégré dans le modèle de composant, la définition d'information relative au comportement à tous les niveaux du composant et le manque de cohérence entre ces informations gênent l'analyse d'un assemblage de composants. Pour finir, pour réussir une bonne utilisation des composants on a besoin d'informations supplémentaires autres que les spécifications fonctionnelles. On a besoin de propriétés extra fonctionnelles qui déterminent les comportements des composants en regroupant des contraintes temporelles comme le temps d'exécution et la latence ainsi que des contraintes liées à la fiabilité, la performance, l'efficacité, la synchronisation et la sécurité.

2.8 Conclusion

Dans ce deuxième chapitre, nous avons essayé de donner un aperçu plus ou moins détaillé sur le développement d'applications à base de composant et plus généralement sur la sous discipline du génie logiciel orienté composant. Ce type de développement et qui présente une approche ayant un grand impact sur le développement de logiciel cible comme objectif principal la recherche d'un développement efficace, c'est-à-dire la recherche de méthodes qui permettent un développement qui se fait en peu de temps et produit des applications de qualité. Dans cet aperçu on a présenté les définitions relatives à la majorité des concepts et des notions en relation avec la sous discipline du génie logiciel orienté composant, l'accent est mis sur les définitions faisant le maximum d'accords dans la communauté composant. Egalement on a présenté quelques modèles et approches à composants ayant connu des réussites notamment sur le plan industriel. Nous nous somme intéressé particulièrement aux capacités d'adaptation automatiques des applications à base de composants. Le présent chapitre inclut une étude de quelques mécanismes de reconfiguration automatique d'applications à base de composants qui ont été proposés dans quelques modèles et approches à composants. Nous discutons dans le chapitre suivant la combinaison d'une telle importante approche de développement avec un autre paradigme de développement assez riche et puissant sur plusieurs plans. Il s'agit du paradigme agent et agent mobiles. Nous discutons les apports mutuels et les différentes possibilités de combinaison des deux technologies pour le développement d'applications qualifiées de : ouvertes, distribuées et adaptables.

Chapitre III

Agents & Composants :

Concepts, analyse comparative et apports mutuels