Université Abbes Laghrour Khenchela
Faculté des Sciences et de la Technologie
Département Mathématiques et Informatique

Mémoire

Pour obtenir le diplôme de

Magistère en informatique

Option **: Système d'Information et de Connaissances (SIC)**

Présenté par : **Abderaouf BAKHOUCHE**

Novembre 2013

Vérification des critères de validité fonctionnelle des assemblages dans les applications à base de composants (Approche basée Agents mobiles)

JURY

Mohamed Ridda LAOUAR	Directeur de mémoire	MCA	Université de Tébessa
Farid MOKHATI	Président	MCA	Université d'Oum El Bouaghi
Hakim BENDJENNA	Examinateur	MCA	Université de Tébessa
Abdelkarim AMIRAT	Examinateur	MCA	Université de Souk Ahras

Résumé

L'informatique d'aujourd'hui est de plus en plus distribuée, connectée et hétérogène, , les applications sont de ce fait plus compliquées à développer par leurs nombre et leur taille et surtout leur dynamicité, la réutilisation est devenue une nécessité pour palier à l'explosion du nombre et de la complexité des applications.

Le paradigme composant représente une approche de conception et de développement de logiciel ayant un grand impact. En proposant des abstractions pour organiser le logiciel comme une combinaison d'éléments logiciels, avec pour objectif de faciliter son évolution (en premier lieu, remplacement et ajout d'éléments).

La validité d'une application à base de composant consiste en trois point de vue, un point de vue fonctionnel dans lequel on cherche à s'assurer que le comportement global issus des comportements individuels des composants se rencontre bien avec les besoins prévues de l'application; le deuxième point de vue concerne l'aspect structurel de l'application c à d lié à la dynamique de l'environnement d'exécution de l'application, et finalement un point de vue de consistance qui consiste à vérifier la compatibilité entre les signatures des composants.

Le travail demandé cible à traiter le point de vue fonctionnel c.à.d. proposer un mécanisme pour une vérification automatique des critères de validité fonctionnelle d'un assemblage de composants en faisant intervenir les agents mobiles.

Mots-clés : Vérification, Composants, SMA, Agents Mobiles, validité fonctionnelle.

Summary

Today IT is increasingly distributed and heterogeneous connected, applications are therefore more complicated to develop their number and their size and especially their dynamicity, reuse has become a necessity for bearing the explosion of the number and complexity of applications.

The component paradigm is an approach for design and development of software that is a big impact. By proposing abstractions to organize the software as a combination of software components with the aim to facilitate the development (first for all, replacement and addition of elements).

The validity of a component-based application consists of three terms, a functional point of view in which one seeks to ensure that the overall behavior from individual behaviors of components meeting the requirements laid down well with the implementation, the second point of view concerns the structural aspect of the application i.e. linked to the dynamics of the application execution environment, and finally a perspective of consistency that is to check the compatibility between the signatures of components.

The work required target treat functional point of view i.e. provides a mechanism for automatic verification of functional criteria of validity of an assembly of components involving mobile agents.

Keywords: Verification, Components, MAS, Mobiles Agents, functional validity

ملخص

الإعلام الألي تطور بشكل كبير و أصبحت تطبيقاته و وسائله موزعة و مختلفة المصادر و كذلك مترابطة فيما بينها، وهذا ما نجم عنه تعقيدات كبيرة في تطوير و إنجاز هذه التطبيقات. كل هذا بسبب عدد و حركية التطبيقات، و مما جعل إعادة الإستعمال ضرورة لمواجهت كل هذه التعقيدات

نموذج البرمجة بالمكونات هو نهج لتصميم وتطوير البرمجيات وله تأثير كبير .من خلال اقتراح تجريدات لتنظيم البرنامج ومزيج من مكونات البرامج يهدف الى تسهيل التطوير (أساسا استبدال وإضافة عناصر).

صحة التطبيقات القائم على البرمجة بالمكونات يتكون من ثلاثة شروط: وهي من وجهة نظر وظيفية اين نسعى لضمان أن السلوك العام المنبثق من السلوكيات الفردية للمكونات للوفاء بالشروط المحددة، ومن جهة نظر ثانية يتعلق بالجانب الهيكلي للتطبيقات أي التطابق مع ديناميكية البيئة في تنفيذ التطبيق، وأخيرا منظور التناسق الذي هو التحقق من التوافق بين توقيعات المكونات.

العمل المطلوب يستهدف العلاج من وجهة النظر الوظيفية أي توفير آلية للتحقق الآلي من معايير وظيفية من صحة تجميع المكونات التي تدخل فيها نظم العملاء المتنقلة.

كلمات البحث: التأكيد، المكونات، نظام متعدد العملاء عملاء متنقلة، صحة وظيفية

Dédicace

A ma mère et mon père

A mon épouse

A Houda, Youga et Hennou sans oublier Hlima

A ma chère grande mère

A mes oncles et mes tantes

A toutes les personnes que j’aime

Remerciements

Je remercie respectueusement mon encadreur le docteur Mohamed Ridda LAOUAR qui m'a accepté dans son équipe de m'avoir soutenu et dirigé dans ce travail.

Je tiens également à exprimer mes remercîments à mon ami Abderahim SIAM pour son aide précieuse

Sans omettre d'exprimer toute ma gratitude aux membres de jury

Je rends un grand hommage à ma mère et mon père qui m'ont porté et supporté toute ma vie

Et je remercie mon épouse qui m'a soutenu tout au long de ce travail

Je n'aurais garde d'oublier de remercier tous ceux qui m'ont aidé de prés ou de loin.

Table des matières

Table des figures

Introduction générale

Le monde dans lequel nous vivons aujourd'hui est complètement différent de celui qu'il y à trente ou quarante ans, jadis l'innovation scientifique était surtout dans l'aéronautique et la conquête de l'espace, le nouveau millénaire a apporté avec lui une nouvelle -tendance- et de nouveaux défis après ceux de l'informatique personnelle et personnalisée c'est au tour de l'inter connectivité sans frontières et sans obstacles, tout est connecté (Smartphones, agences, banques, véhicules, entreprises…), tout ce monde utilise des systèmes informatiques de plus en plus complexes et compliqués, ce qui a engendré des besoins nouveaux (sécurité, rapidité, interopérabilité, surcharge des réseaux…) en plus des anciens (optimisation des systèmes, conformité aux exigences, gestion du développement de logiciels en coût et en temps…), tout ces défis concernent aussi le génie logiciel qui a de tout temps suscité un besoin accru en réutilisation d'éléments déjà existants, l'assemblage de ces éléments permet de réduire la création de nouvelles entités donc gagner en souplesse, en coût et en temps.

Les éléments fabriqués sont toujours construits au sein d'un contexte. Il y a donc présence d'effets de bords, de pré-requis et de contraintes d'utilisation, ces contraintes contextuelles peuvent influer sur la qualité du produit final, par exemple dans les systèmes critiques peuvent devenir fatales, l'exemple de la fusée Ariane qui a explosé à la suite d'un dépassement de capacité d'un de ses calculateurs de vitesse est révélateur.

Une des solutions les plus prometteuses est l'approche de développement de logiciel basé composant (CBD[1]). Cette approche est basée sur l'idée que les systèmes logiciels sont construits par l'assemblage des composants déjà développés et préparés pour l'intégration avec une architecture logicielle bien définie. Cette approche de développement de logiciel est différente des approches traditionnelles dans lesquelles les systèmes logiciels sont construits à partir de zéro.

L'un des points importants dans l'approche basée composants et qui présente un problème est de trouver et savoir assembler ces composants et vérifier leur assemblage pour répondre aux besoins des développeurs des Systèmes Informatiques, c'est-à-dire adaptés au contexte et aux spécificités du système en cours de développement (fonctionnelles). Les difficultés sont liées le plus souvent à l'absence d'une description « de haut niveau » des

[1] Component Based Development

composants réutilisables et au faible couplage entre la base de composants réutilisables et le processus qui en assure la réutilisation pour assurer le bon système.

Proposer un mécanisme pour une vérification automatique des critères de validité fonctionnelle d'un assemblage de composants en faisant intervenir les agents mobiles. Par l'intégration des contrats qui assurent la définition explicite des fonctionnalités fournies par les composants, de cette manière on répond au problème des contraintes mais sachant les contrats n'assurent pas la manière dont ces mêmes fonctionnalités se déroulent (Huot, 2004) et qui présentent une certaine défaillance sémantique on a proposé d'utiliser un mécanisme connu en informatique en général et adapté au développement logiciel qui est la session , les sessions sont des canaux de communications qui offrent des interactions sûres et qui représentent pour nous une offre de solution au problème cité plus haut, plusieurs auteurs les ont adapté en ajoutant un autre principe de résolution de la compatibilité entre systèmes de type qui est le sous-typage donc on les appelle types de sessions.

De cette manière et avec un formalisme explicite et unifié nous espérons garantir jusqu'à certain degré la vérification sémantique donc fonctionnelle de l'assemblage concluant un système conforme et cohérant. En plus, un système de vérification et de validation d'assemblage automatisé en partie grâce aux agents qui diminue la probabilité d'erreurs statiques (avant exécution) et respectant la sémantique des spécifications lors du développement par le biais des contrats et les types de sessions.

Organisation du mémoire :

Dans un premier chapitre nous introduisons le contexte de la problématique qui est les applications distribuées et reparties. La vérification des assemblages de composants systèmes pour ce type d'applications et leurs caractéristiques.

Puis dans un deuxième chapitre le paradigme composant, avec des définitions et des propriétés des composants et quelques modèles qui existent, leurs avantages et limites, ainsi que les catégories et les représentations des services rendus par les composants, à travers les interfaces des composants qui exigent une bonne description, pouvoir décrire le comportement des composants et donc les évaluer individuellement pour mieux savoir les assembler, moyennant des formalismes de description qui seront détaillés dans le chapitre suivant.

Le troisième chapitre traite donc les différents formalismes de description de manière générale et la description des composants en particulier, ces formalismes qu'on va utiliser pour donner un aspect formel à la vérification sémantique des composants et de leur assemblage, avec les contrats et les types de session la vérification peut être plus rigoureuse.

Le chapitre quatre (04) quant à lui introduit les Systèmes Multi Agent (SMA) et spécialement les agents mobiles utilisés dans le cadre de ce mémoire, et présente différents aspects des SMA, en plus des apports mutuels entre SMA et composants systèmes, pour mieux éclairer leurs complémentarités l'un vis avis de l'autre.

Le chapitre cinq (05) détaille notre proposition pour vérification des systèmes d'assemblages par les deux mécanismes cités plus haut, les contrats et les types de session, en décrivant des définitions et des propositions des deux mécanismes ainsi que le système hybride appelé contrats de session (CS).

Et enfin dans le chapitre six (06) une étude de cas applicatif sur un serveur de vote du système de vérification proposé et les assemblages avec un classement. Cette vérification est réalisée en partie par les agents qui contribuent à l'efficacité du système de vérification.

Chapitre I : Contexte et problématique de vérification des assemblages des composants logiciels

1.1 Introduction

Lorsque l'on utilise le terme « informatique », on peut entendre plusieurs sens allant du logiciel de bureautique (traitement de texte, tableur ...) jusqu'au composant d'un ordinateur (processeur, mémoire etc..) ou encore un gigantesque système de traitement de données.

D'un point de vue matériel, jusqu'à présent les technologies des réseaux de communication sont construites, sur des architectures de communication quasiment fixes, ayant éventuellement une topologie particulière (anneau, bus, arbre par exemple) et dans lesquelles l'évolution des connexions (apparition/rupture) était un phénomène rare. Cependant, avec l'utilisation de plus en plus courante des technologies reparties et connectées, les réseaux évoluent de plus en plus vers de nouvelles architectures dynamiques à large échelle. Les sites deviennent dynamiques, apparaissent et disparaissent fréquemment, les connexions évoluent de façon continue.

Les interconnexions de machines physiques, appelées réseaux d'ordinateurs, ont été introduites au départ pour faciliter les échanges de données entre les ordinateurs sans utiliser de fastidieux supports de stockage (bandes, disquettes, cartes perforées etc.) et pour permettre la mise en relation des usagers séparés par de longues distances (Tanenbaum, 1990). À l'origine, ces réseaux formaient des structures de communication indépendantes, mais pour permettre aux utilisateurs d'avoir un environnement homogène, les techniques d'interconnexion de réseaux se sont largement répandues afin d'offrir un système affranchi des problèmes d'hétérogénéité matérielle (Comer, 1996).

Ces systèmes, se sont développés de manière continue jusqu'à former le fameux Internet constituant le World Wide Web. Pour tirer partie au mieux de ces environnements connectés, le développement d'applications pouvant prendre en compte un ensemble de machines a été rapidement étudié. C'est ce type d'applications qu'on désigne par réparties ou distribuées. Celles-ci peuvent s'exécuter sur plusieurs machines différentes, utiliser des ressources distribuées, mettre en place des mécanismes de tolérance aux fautes, etc. Ce qui permet d'aller bien au-delà du simple échange de données.

1.2 Les Applications Réparties

Les applications Réparties utilisent, au cours de leurs exécutions, un ensemble d'éléments, encore appelés ressources, qui peuvent être réparties sur différents sites du réseau sous-jacent. Pour accéder et utiliser les ressources, plusieurs schémas d'organisation sont envisageables.

Quatre grands types de schémas organisationnels selon (Cubat dit Cros, Agents Mobiles Cooperants pour les Environnements Dynamiques, 2005). La conception d'une application répartie pouvant mélanger plusieurs de ces types, il s'agit ici d'une classification générale non stricte.

- Mémoire distribuée partagée : Ce schéma d'organisation est basé sur la notion d'un espace d'échange partagé par un ensemble de processus distribués, Cette mémoire, généralement de grande taille, permet de faciliter les échanges de données pour les applications dont le partage est un facteur d'importance.
- Abonnement/publication : Ce schéma d'organisation permet de faire communiquer indirectement différents processus grâce à la collaboration d'un tiers. Le principe est de faire savoir, aux processus qui le souhaitent, qu'un évènement précis est arrivé dans l'environnement, la création d'un fichier par exemple.
- Pair à Pair : Dans ce schéma d'organisation, tous les éléments possèdent le même rôle au sein de l'application. Ces éléments sont appelés des servants qui possèdent un ensemble de voisins avec lesquels ils peuvent dialoguer en leur nom ou pour celui d'un autre. Ainsi, si deux servants non-voisins veulent communiquer, ils peuvent le faire grâce à un ensemble d'intermédiaires relayant la communication.
- Client/serveur : Dans ce dernier schéma, on sépare une fonctionnalité précise de celui qui l'utilise grâce à une répartition des éléments et à une communication à distance. En d'autres termes, une application rendant une certaine fonctionnalité (un service) est placée sur un nœud du réseau (le serveur) en attente de requêtes émisses par d'autres applications (les clients) réparties sur le réseau.

1.3 Les systèmes distribués

Les systèmes distribués (ou encore systèmes répartis), en opposition aux systèmes centralisés, sont composés de plusieurs machines indépendantes. Du point de vue de l'utilisateur, aucune différence n'est perceptible entre un système distribué et un système

centralisé. Cela dit, ces systèmes permettent de garantir des propriétés qui ne sont pas disponibles dans un système centralisé, comme par exemple la redondance, qui permet de pallier les fautes matérielles ou de rendre un même service disponible à plusieurs acteurs sans perte de temps ; la performance, garantie par la mise en commun de plusieurs unités de calcul permettant de réaliser des traitements parallélisables en un temps plus court ; et la protection des données, qui ne sont pas disponibles partout au même moment, mais dont seules certaines vues sont exportées.

Un système distribué est généralement séparable en plusieurs modules entièrement autonomes, chacun responsable de son propre fonctionnement. Cette autonomie permet d'une part d'utiliser des technologies, plateformes ou langages hétérogènes dans chacun de ces modules, et d'autre part de les exécuter simultanément et garantir ainsi une programmation concurrente.

Les applications apparues récemment sont de plus en plus distribuées, ouvertes et à architecture complexe. De telles applications sont composées d'un nombre, généralement important, d'entités logicielles dispersées sur le réseau coopérant pour fournir à leurs utilisateurs les services qu'ils requièrent. Cette complexité est liée, par exemple, à l'extension des réseaux de communication, à l'hétérogénéité matérielle et logicielle, au fort degré d'interaction et aux exigences d'évolutions nécessaires et permanentes. Le but étant d'offrir des services de plus en plus performants, robustes et sûrs, et qui s'adaptent aux différents besoins des clients. L'adaptation ou la reconfiguration logicielle constitue l'un des enjeux et défis techniques les plus importants. En effet, l'adaptabilité d'une application est devenue impérative dans un contexte de systèmes fortement distribues et contenant un grand nombre de composants. Dans ce cas, une reconfiguration ad-hoc n'est pas envisageable.

1.4 Adaptation dynamique

Pour qu'un système fournisse un service adapté, il est nécessaire de le configurer correctement. Dans les cas simples, la configuration peut être réalisée manuellement. Dans le cas d'applications complexes, l'utilisateur n'a pas forcément les connaissances nécessaires pour choisir les meilleurs paramètres ou la meilleure configuration. De plus, si l'environnement d'exécution de l'application a des caractéristiques variables, il peut être nécessaire de réajuster ces paramètres régulièrement. Le but de l'adaptation dynamique est de

pallier ce problème en proposant un mécanisme pour définir (et redéfinir) la configuration de l'application.

1.4.1 Reconfiguration dynamique des applications distribuées

Trois types de reconfiguration existent selon (Hofmeister, 1993)

- Les changements géométriques
- Les changements structurels
- Les remplacements d'entités

1.4.2 Vérification de l'assemblage de composants et reconfiguration

Concernant la vérification de la compatibilité entre les composants des travaux ont été mené en utilisant le modèle des automates d'interface. Un composant est muni de contrats spécifiques et de dépendances avec son environnement. Une interface décrit les services offerts et les services requis. Introduit par (Alfaro & Henzinger, 2001), le formalisme des automates d'interface permet de décrire l'enchaînement des appels de services.

Ce travail a consisté, dans un premier temps, à intégrer la sémantique des actions dans les automates d'interface, chaque action étant munie de pré et de post conditions. La vérification de la compatibilité entre composants se ramène à la détection d'états illégaux en se basant sur le produit synchronisé de deux automates. L'opération de substitutivité est décrite sous la forme d'une relation d'affinement entre les automates d'interface.

Proposer un mécanisme pour une vérification automatique des critères de validité fonctionnelle d'un assemblage de composants en faisant intervenir les agents mobiles. Par l'intégration des contrats assurant la définition explicite des fonctionnalités fournies par les composants, de cette manière on assure une partie mais sachant les contrats n'assurent pas la manière dont ces mêmes fonctionnalités se déroulent (Huot, 2004) et qui présentent une certaine défaillance sémantique on a proposé d'utiliser un mécanisme connu en informatique en général et adapté au développement logiciel qui est la session , les sessions sont des canaux de communications qui offrent des interactions sures et qui représentent pour nous une offre de solution au problème cité plus haut, plusieurs auteurs les ont adapté en ajoutant un autre principe de résolution de la compatibilité entre systèmes de type qui est le sous-typage donc des types de sessions.

Les agents mobiles ont été introduits en 1994 par la description de l'environnement Téléscript comme une autre alternative de la mobilité. Dans cet environnement, des processus (code et unité d'exécution) peuvent se déplacer d'eux-mêmes d'un site du réseau à un autre pour interagir localement avec des ressources d'autres sites. Cette technologie est alors apparue comme prometteuse pour la conception d'applications distribuées.

1.5 Problématique et Motivations

L'informatique d'aujourd'hui est de plus en plus distribuée, connectée et hétérogène, les applications sont de ce fait plus compliquées à développer par leurs nombre et leur taille et surtout leur dynamicité, la réutilisation qui étais presque un luxe est devenue une nécessité pour pallier à l'explosion du nombre et de la complexité des applications, les composants représentent un bon moyen pour la réutilisation, mais parmi les problèmes posés avec les composants c'est l'assemblage. Deux aspects se font ressentir avec l'assemblage, le premier fonctionnel (exemple processus de calcul, édition des résultats, …) et le second non-fonctionnel (comme par exemple, la sécurité, la gestion de la mémoire, …).

Notre contribution donc se concentre sur la vérification de la validité d'un assemblage de composants systèmes dans sa partie fonctionnelle, qui veut dire services rendus par l'application sans se préoccuper des problèmes du contexte et de l'environnement d'exécution.

À travers la combinaison de deux mécanismes de vérification comportementale, les contrats (Meyer, 1992), les types de session (Vallecillo, Vasconcelos, & Ravara, 2005) qui donnent les contrats de session, nous visons d'augmenter l'efficacité des assemblages et d'avoir un meilleur rendement. Ainsi, à l'aide des agents mobiles nous pourrons apporter plus de souplesse et d'adaptabilité pour la vérification, sans pour autant ignorer l'essentiel qui est la vérification du système produit pour l'utilisateur.

Nous intéressons aussi à proposer un algorithme de classement basé sur la distance entre un assemblage donné et la spécification du client pour la sélection des assemblages valides possibles.

Notre approche porte sur l'assistance aux développeurs des applications dans la construction par assemblages de composants. L'idée est de proposer un mécanisme de vérification de la validité fonctionnelle d'un assemblage de composants à l'aide d'un

formalisme qui résulte de la combinaison de deux formalismes connus, l'un tire son origine d'une pratique assez ancienne et adapté à l'informatique qui est le contrat, l'autre est plus récent et innové pour l'informatique, les types de session, le premier est plus riche en matière de recherche donc plus mure que le second.

À travers ce formalisme utilisé pour la description des composants et qui permet d'autoriser des inférences automatiques des vérifications à différentes granularités d'assemblage et pour un niveau d'abstraction assez élevé, en évitant les adaptations contextuelles aux langages et aux plates-formes d'implémentation.

La collecte d'informations sur les composants (emplacement, versions, descriptions,…) est une tache dynamique qui exige beaucoup d'analyse et de mouvement (changement continuel). Les agents système peuvent être utilisés comme solution pour prendre en charge cette tache.

La mobilité des agents offre plus de pertinence à ce système permettant une meilleure présence sur les différents sites en économisant le temps et la bande passante pour système. Ainsi, l'automatisation permet une évolution dynamique du système anticipant la substitution des composants pour faire face à d'éventuelles pannes, péremption ou encore indisponibilité des composants.

Chapitre II : L'approche par composants

2.1 Introduction

Le développement d'applications est de plus en plus complexe engendré par les besoins croissants des utilisateurs. Le résultat est que les concepts manipulés par la communauté informatique sont de plus en plus abstraits, mais en contrepartie les mécanismes sous-jacents sont de plus en plus complexes.

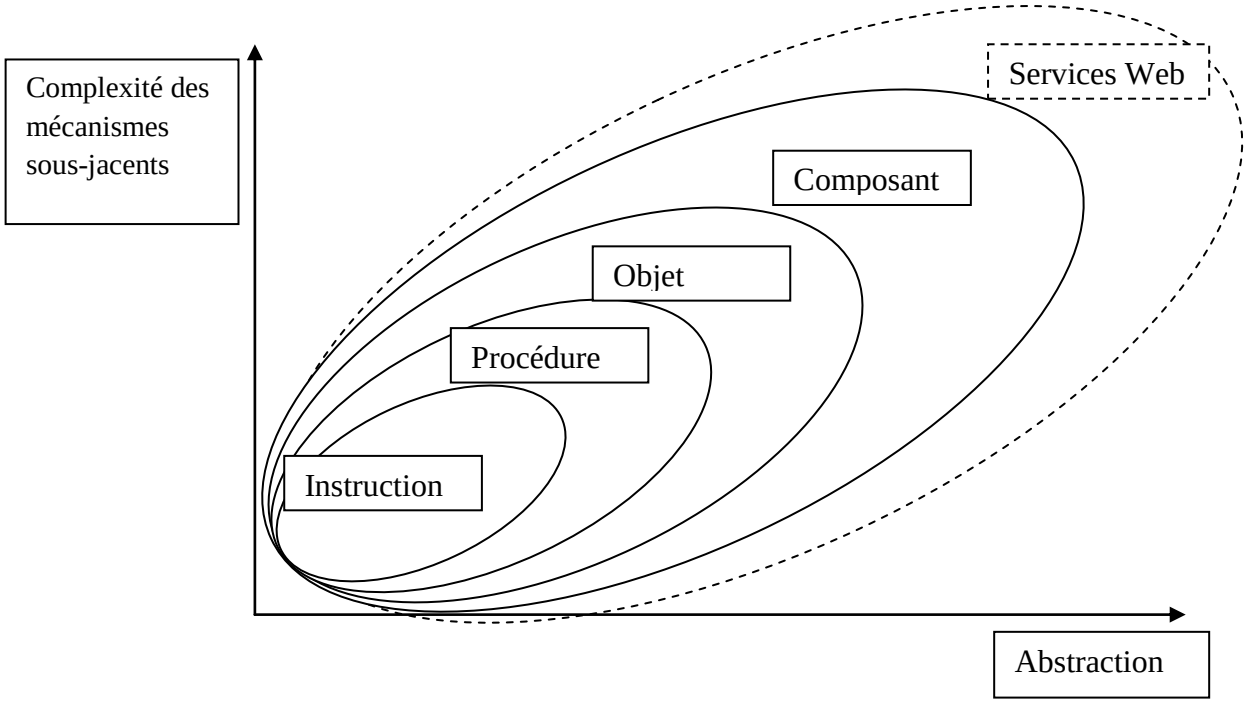


Figure 1: Comparaison des concepts **(Cyril, 2003)**

La **Figure 1** résume l'évolution des concepts de l'informatique dés ses débuts jusqu'à récemment, du plus simple concept qui est l'instruction plus proche de la machine jusqu'au concept plus évolué et plus complexe qui est le composant système et ensuite les Services Web qui sont plus récents et plus dynamique, en passant par la procédure (ou fonction) qui permet de s'éloigner un peu plus du cœur de la machine, qui permet une meilleur la portabilité, car la procédure peut être réutilisée à d'autres endroits du programme.

Les programmes sont plus pensés en termes d'algorithmes que de données manipulées dans les procédures qui manipulent des données.

Le développement de logiciels à base de composants s'est fait ressentir et ce par imitation de l'industrie qui avait su tirer profit de la réutilisation et standardisation de production par composant favorisant l'indépendance et la transparence des composants mis en contribution et des équipes engagés dans le processus de développement.

2.2 L'approche de développement par composants

Eli Whitney a proposé la fabrication de fusils avec des pièces remplaçables spécifiés clairement selon des modèles, au lieu de construire chaque fusil individuellement. Largement considéré comme le commencement de fabrication en série, puis Frederick Taylor introduit les principes du management scientifique du travail et la décomposition des tâches en unités parcellaires. Et d'autres après qui ont amélioré et développé le concept de réutilisation.

En ce qui concerne les Systèmes informatiques En 1968, (McIlroy D. , 1968) était précurseur en la matière en posant l'idée sur la place scientifique en suggérant la réutilisation dans l'industrie du logiciel.

Des années 70 aux années 80, les ingénieurs utilisaient une méthodologie de développement logiciel structurée pour diviser un système en un certain nombre de modules. Après que les modules aient été développés, ils les intègrent pour former un système.

Puis les objets ont fait leur apparition considérés comme un moyen puissant pour résoudre le problème de la crise logicielle grâce au niveau élevé de réutilisation et de maintenance atteint par ces derniers.

Nierstrasz qui est en relation avec la composition logicielle pour les systèmes ouverts distribués, peut être considéré comme une importante contribution aux futurs concepts et idées du CBD.

2.3 Concepts et notions de base

Suite aux nouvelles contraintes et obstacles auxquelles doit faire face le génie logiciel, les facteurs « coût » et « temps » sont de plus en plus difficiles à maîtriser dans le cycle de vie d'un logiciel. Parmi ces nouvelles contraintes, deux sont centrales: la taille croissante des applications et leur répartition qui est due à l'essor des réseaux informatiques.

Par rapport aux objets les composants se réclament d'une granularité moins fine et font apparaître explicitement leurs dépendances. Les applications à base de composants sont assemblées et non plus développées.

Un composant est un ensemble cohérent de données et de code informatiques. Mais aussi le composant est plus une entité architecturale qu'une façon de programmer. L'approche à base de composants est considérée comme un nouveau paradigme de développement des systèmes d'information (Barbier, Cauvet, Rieu, Bennasri, & Souveyet, 2002). Les activités de programmation ont été les premières concernées par ce nouveau paradigme.

L'émergence de l'industrie du composant n'a été vraiment que dans les années 90. En effet, le développement des logiciels à base de composants (Component-Based Software Engineering) est devenu une réalité avec l'avènement des technologies web et java qui ont facilité la distribution, la recherche, l'interopérabilité des composants sur internet et en particulier des COTS (Commercial Off The Shelf).

Et par cela la réutilisation n'a jamais été aussi poussée : une application paramètre le composant pour s'en servir conformément à ses besoins spécifiques. Par contre, la complexité sous-jacente n'a jamais été aussi grande.

Cependant la conception d'application est devenue plus exigeantes et plus pointue vue la taille et la complexité des applications et des équipes de développement. Surtout en ce qui nous concerne l'assemblage cohérant et répondant aux exigences.

À partir de là le développement à base de composant doit répondre à une certaine qualité de développement. (Meyer, 2000) Définit sept (07) critères pour une bonne conception et une meilleure définition d'un composant :

- Utilisable par d'autres systèmes (pas que d'humains).
- Utilisable par des éléments logiciels dont les auteurs sont inconnus aux auteurs du composant.
- Un composant doit inclure une spécification de toutes ses dépendances: plate-forme matérielle et logicielle, les versions et les autres composants.
- Pour la même raison, un composant doit fournir une spécification précise des fonctionnalités qu'il offre.

- Le composant doit être utilisable sur la seule base de ce cahier des charges, sans accès aux autres informations (comme le code source, même s'il est disponible).
- Les composants doivent être composable avec d'autres composants.
- Le processus d'intégration d'une composante dans les systèmes qui l'utilisent doivent être rapides et en douceur.

2.3.1 Définition d'un composant

Il existe de nombreuses définitions des composants. Parmi elle et l'une des plus admise est celle de (Szyperski, 1998) :

« 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 est l'objet de la composition par des tiers».

Ou encore et plus intéressant:

"Components are for composition"(Szyperski, 2002) C'est la phrase sur laquelle beaucoup de gents sont d'accord.

Un composant réutilisable est défini selon (Zitouni, 2008) 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 générale du concept de composant admet la possibilité d'utiliser le concept de composant dans des niveaux d'abstraction autres que le niveau d'implantation.

Le projet ACCORD (ACCORD, 2002) définit le composant comme *« une entité logicielle qui réalise des interactions avec d'autres composants via des connecteurs »*.

En 2000, une notion plus large et générale de composants logiciels a été définie par (Brown, 2000): « Une pièce indépendamment livrable de fonctionnalités permettant d'accéder à ses services via des interfaces ».

« 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.».

La **Figure 2** résume ce qui est un composant:

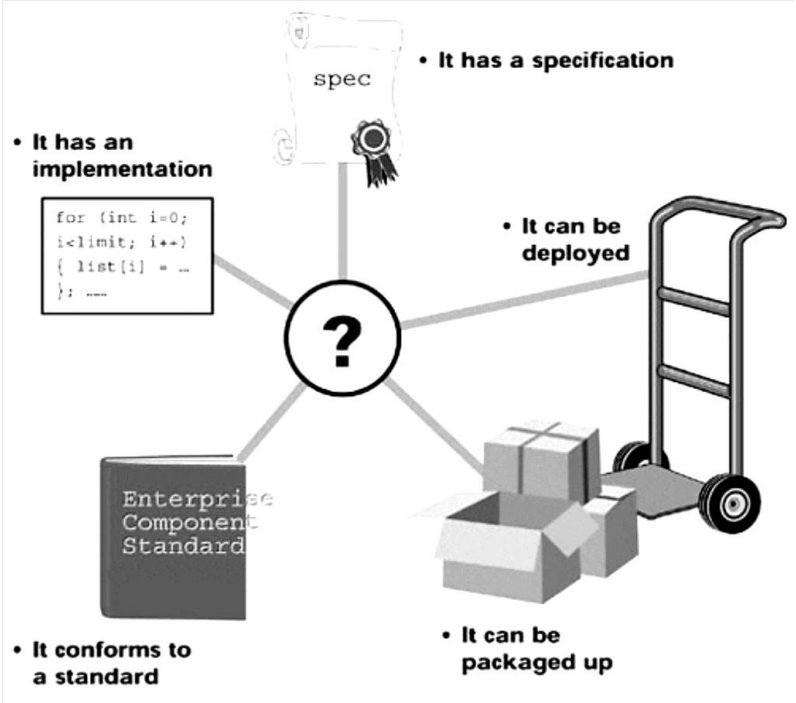


Figure 2: Ce qui est un composant **(Brown, 2000)**

Une spécification ; S'appuyant sur le concept d'interface, un composant nécessite une description abstraite des services qu'elle offre à agir en tant que contrat entre les clients et les fournisseurs des services. En plus d'un avantage pas des moindre qui est la transparence des auteurs des composants vis-à-vis de leurs clients (ceux des composants) et ce grâce à l'abstraction et la généricité permises par le paradigme composant.

Un composant selon une bonne partie des auteurs est défini par au moins :

- Le code qui est la concrétisation des services proposés par le composant
- La spécification du composant qui décrit les interfaces et leurs types et les propriétés du composant (fonctionnelle comme les contraintes, et non fonctionnelles)

Plus programmatiquement, les composants sont en général des structures de données et de code se présentant souvent sous forme de boites noires mais pas seulement, il y aussi ce qu'on appelle boites blanches et entre elles les boites grises **Figure 3**. Ils n'interagissent avec leur environnement que via les interfaces qu'ils exposent, qu'ils requièrent ou fournissent.

Dans les systèmes à composants, les entités impliquées sont essentiellement les composants et leurs connecteurs, organisés dans une architecture généralement explicite.

On peut synthétiser qu'un système à base de composants ressemble à un puzzle réalisé à partir d'éléments autonomes et transparents par rapport au contexte des composants, ces éléments de différents niveaux de granularité et de complexité offrent leurs services à travers des interfaces, ces interfaces ont des spécifications bien décrites pour qu'elles permettent une analyse adéquate de leurs comportements permettant une vérification de l'assemblage de manière automatique.

2.4 Catégories des composants

Les composants se composent de différents éléments et se voient de différentes manières, leurs classification dépend des différents points de vue (niveau de transparence, niveau d'abstraction, spécialisation..), la classification des composants permet de mieux entourer leur utilisation,

2.4.1 Niveaux de Transparence des composants

La notion de composant divise les auteurs, de par leur degré de transparence en matière de visibilité et des détails apparents des composants :

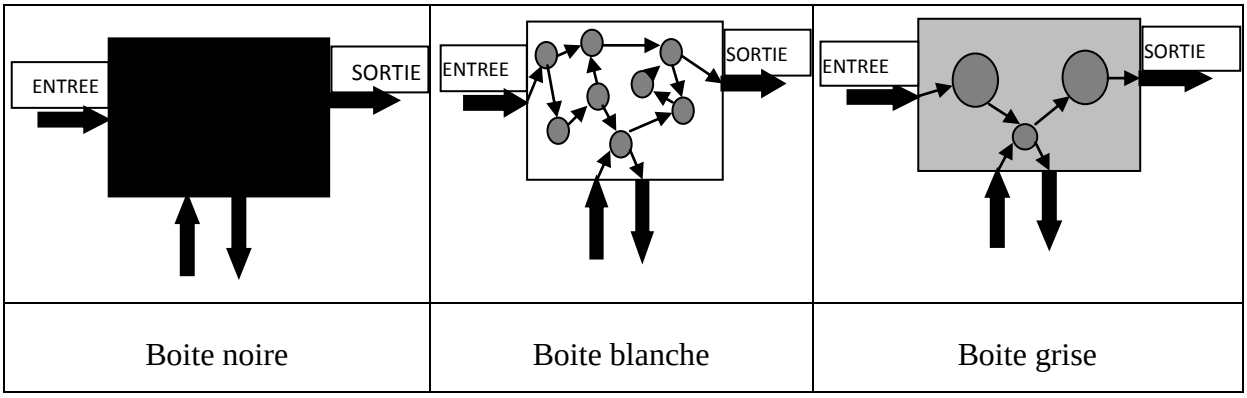


Figure 3: Représentations d'un composant **(ACCORD, 2002)**

- Certains auteurs font du composant une boîte noire **(Figure 3)** dont le code binaire est vendu avec un mode d'emploi et des spécifications. (la Réutilisation de boîte noire se réfère à la réutilisation d'une mise en œuvre sans compter sur autre chose que son interface et les spécifications).
- Dans un point de vue totalement opposé, d'autres auteurs ont préféré la solution de la boîte blanche **(Figure 3)** (solution où l'on fournit tout le code du composant). (la réutilisation des boîtes blanches rend improbable que le logiciel réutilisé peut être remplacé par une nouvelle version. Un remplacement cassera probablement certains

des clients de réutilisation, car ceux-ci dépendent des détails d'implémentation qui peuvent avoir changé dans la nouvelle version).

- Une solution intermédiaire est celle dite de la boîte grise (**Figure 3)**. (Des composants sont ceux qui révèlent une partie contrôlée de leur mise en œuvre. une mise en œuvre partiellement révélée dans le cadre de la spécification).

Sachant que cette notion intéresse beaucoup plus ce qu'on appelle logiciel pour la réutilisation (**Figure 5**).

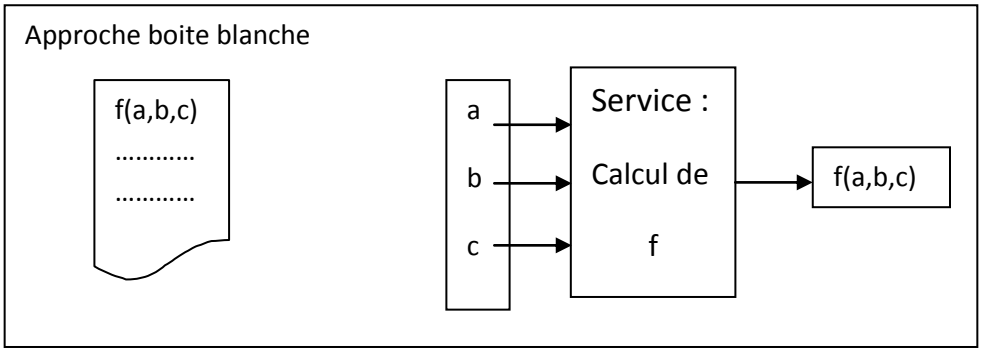


Figure 4: Différence entre le concept de boite blanche et boite noire **(Khayati, 2005)**

2.4.2 Spécialisations des composants

Trois types principaux de composants peuvent être distingués (Zitouni, 2008) (Khayati, 2005): les composants conceptuels, les composants logiciels et les composants métier.

- Les composants conceptuels : Sous la forme d'un modèle (ou une partie d'un modèle) destinée à être réutilisée qui est une solution à un problème conceptuel.
- Les composants logiciels : est un paquetage cohérent d'implantations logicielles qui peut être indépendamment développé et délivré.
- Les composants métier: Le concept de composant métier résulte de celui d'objet métier OMG *« Business Objects are representations of the nature and behavior 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 »*.

Le CBSE (Component Based Software Engineering) utilise les méthodes, outils et principes de l'ingénierie logicielle classique. Cependant, la CBSE distingue deux cycles de

vie: un pour le développement du composant (Design for reuse) et un autre pour le développement d'un système à base de composants (Design by reuse).

Dans la phase de développement du composant, la réutilisabilité est l'enjeu principal ; Un composant doit être spécifie précisément et formellement, facile à comprendre, assez générique, facile à adapter, à délivrer, à déployer et à replacer. La spécification d'un composant est donnée par ses interfaces sous forme de méthodes externes, séparées de l'implantation du composant.

L'implantation du système à base de composants n'est pas la partie la plus longue. Le plus laborieux est le choix des composants : les localiser, sélectionner le plus approprie, les tester, les vérifier etc.

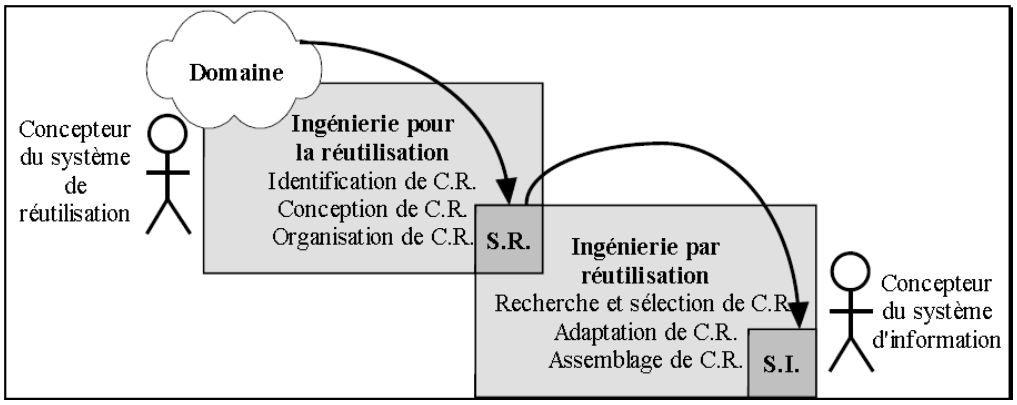


Figure 5: Conception des systèmes par réutilisation et pour réutilisation **(Ramadour, 2000)**

2.5 Représentation des fonctionnalités des composants

Dans la plupart des modèles actuels, un composant possède plusieurs fonctionnalités qui sont représentées par des services. Un service est généralement une fonction (ou opération) définie par un composant. L'invocation de service est le terme désignant le mécanisme permettant à un composant d'exécuter un service suite à la réception d'une invocation et émettre des invocations de services à d'autres composants (**Figure 6**).

2.5.1 Définitions liées à la notion de service

Définition 1: (Service fourni) fonction définie dans le code source d'un composant et offerte aux autres composants pour qu'ils puissent l'invoquer. Un service fourni peut être assimilé à une méthode publique dans le monde objet. Un

composant peut aussi posséder des services qu'il ne fournit pas afin de factoriser son implémentation par exemple.

Définition 2: (Service interne) service non-accessible à l'extérieur du composant qui le définit. Un composant exprime aussi les services qu'il requiert d'autres composants. Ces services requis ne sont pas définis par le composant qui peut tout de même les invoquer dans son code.

Définition 3: (Service requis) service nécessaire au fonctionnement d'un composant (invoqué dans le code de ses services fournis) et fourni par d'autres composants. Ces services sont fournis à travers l'une des plus importantes parties des composants qui doit être clairement différenciée des autres composants et de leur contexte d'exécution.est l'Interface qui représente l'élément clé de la manipulation, la découverte et la l'accès des composants. Et qui permettent un travail de vérification des assemblages moyennant les contrats dans notre cas.

2.5.2 Les interfaces de composant

De façon générale, les interfaces sont un support de description des composants permettant de spécifier comment ils peuvent être assemblés ou utilisés au sein d'une architecture. Les interfaces se situent soit à un niveau local (associées à un port) soit à un niveau global (associées à un composant).

Les interfaces définissent des contrats généralement classés en quatre niveaux (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999) (expliqué dans la section 3.4.3 Contrats).

Du point de vue de l'interface un composant est défini dans (Eriksson & Penker, 2000) « *Une unité de code exécutable qui fournit un ensemble de services physiquement encapsulés selon une architecture de type boîte-noire* ».

(Booch, Rumbaugh, & Jacobson, 1998) Définissent le composant comme *« une entité physique remplaçable d'un système qui se conforme à et réalise un ensemble d'interfaces »*.

On voit bien que le composant est indissociable des interfaces qu'il utilise pour fournir leur services, de ce fait (Sommerville, 2007) définit[2] « Les services offerts par un composant sont disponibles via une interface et toutes les interactions se font par cette interface.

2 Section 19.1 Components and component models, Page 444

L'interface de composant est exprimée en termes d'opérations paramétrées et son état interne ne soit jamais exposé».

Le composant est défini en relation étroite avec les interfaces, (Herzum & Sims, 1999) Le définissent *« Un composant est une unité de programmation auto-suffisante qui peut être indépendamment déployé et placé dans un environnement qui fournit des points de connexions compatibles. Il a un ensemble d'interfaces bien définies et accessibles par le réseau et peut coopérer hors de son support d'accueil avec d'autres composants».*

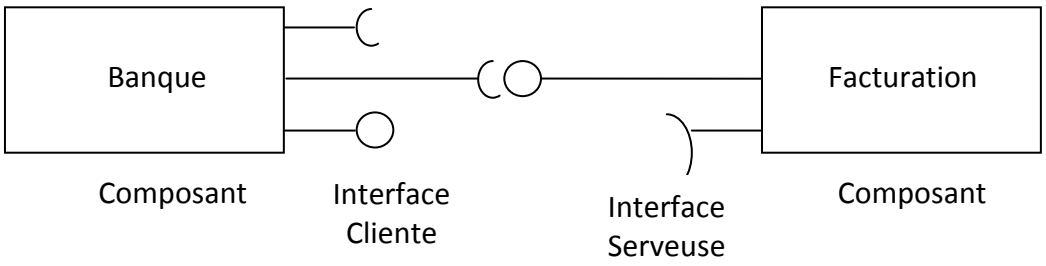


Figure 6: Deux composants avec leurs interfaces connectées (UML)

L'interface d'un composant résume les propriétés qui sont visibles depuis son environnement extérieur (voir exemple **Figure 6**). Elle va lister les différentes signatures des opérations fournies par le composant. Cette liste va permettre d'éviter les erreurs de typage au niveau des connexions du composant puisque les composants se connectent via leurs interfaces (via leurs ports).

Les informations contenues dans les interfaces permettent la vérification de la compatibilité et d'interopérabilité des composants et ceci à travers des descriptions des fonctionnalités des composants qu'on exploitera dans notre approche permettant aux système d'être conforme aux spécifications dans sa composition.

Les interfaces des composants sont décrites par une signature et une description comportementale. Ce qui permet de traiter la compatibilité par le typage et de ce fait arriver à un système le plus cohérant possible et surtout qui réponde au mieux aux exigences fonctionnelles requises en d'autres termes de la Conformité[3] à la spécification.

Un composant lors de son existence, suit un cycle de vie spécifique, qui caractérise dans quel ordre et sous quelles contraintes il est susceptible de répondre aux séquences de services qui vont lui être soumises par ses clients. Pour pouvoir utiliser un composant, il faut connaître

[3] La non-contradiction entre la réalisation d'un élément et la spécification qui le décrit

non seulement son interface (les services qu'il offre et leur signature) mais également son comportement.

La description de l'interface devient donc le médiateur qui permet aux deux parties (composants) de travailler ensemble, Le fournisseur et le client de services sont transparents l'un vis-à-vis de l'autre. On distingue les Interfaces Offertes qui forment des services que le composant fournit à son environnement, des Interfaces Requises qui sont requises pour rendre les services que le composant propose.

Le caractère contractuel des spécifications d'interface est important: donc le composant et ses clients sont développés dans l'ignorance mutuelle, c'est la notion de contrat (voir section 3.4.3 Contrats) standardisé qui constitue un terrain d'entente pour une interaction réussie.

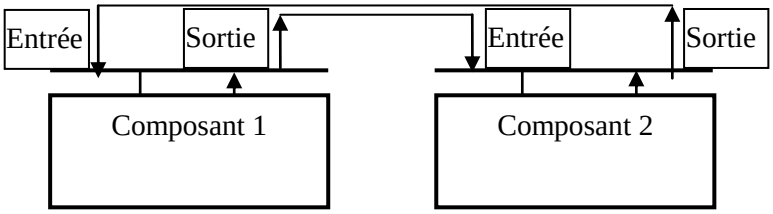


Figure 7: Les composants et leurs interconnexions

2.6 Propriétés du composant

Clemens Szyperski définit trois principales propriétés pour un composant (Szyperski, 2002):

- Un composant est une unité de déploiement indépendante ;
- Un composant est une unité de composition ;
- Un composant n'a pas d'état observable au niveau de l'environnement qui l'entoure.

L'indépendance des composants du déploiement implique que celui-ci doit être entièrement séparé de l'environnement et des autres composants. Il doit avoir une spécification précise de ses besoins mais aussi de ce qu'il peut fournir. Pour pouvoir être composable avec d'autres composants.

Et la notion d'état observable au niveau de l'environnement qui ne l'est pas à ce niveau d'encapsulation implique qu'on ne pourra pas le distinguer des autres composants qui offrent des services identiques. Qui sera au service de la substituabilité (voir 3.5.3 Substituabilité).

La gestion des éventuelles évolutions s'avère plus facile. Plusieurs notions sont mises en jeu dans cette conception (Cyril, 2003) :

- Deux vues existent: le comportement interne du composant (son code), et une description sur la façon dont il doit être utilisé (sa spécification).
- La manière dont les composants sont interconnectés, interaction entre composants, d'où survient le problème de la vérification de compatibilité entre composants.
- Le support d'exécution, c'est-à-dire le lieu où le composant va s'exécuter.

D'autres définitions existent, *Bertrand Meyer*: écrit dans (Meyer, 2000) "I appreciate both Szyperski's points and the Software Development editors insights in setting up this multi-voice column. It's not a fight, but an opportunity for each reader to gain (I hope) a better understanding of the issues and to arrive at his or her own conclusions." Qui montre qu'il n'y avait pas un consensus total et jusqu'aujourd'hui, et que lui il définit les composants comme des logiciels orientés clients. Un composant est au service des autres, en d'autres termes un système compositionnel ou composition de systèmes.

Deux catégories de composants existent, selon qu'il soit atomique ou composé :

- Un composant atomique est un composant élémentaire. Il n'inclue aucun autre composant en son sein ; Les composants atomiques apparait donc comme une restriction (un 'sous type') du composant général. Et c'est l'élément autonome le plus petit possible qui puisse être déployé.
- Un composant composite est un composant réalisé par un assemblage de sous éléments composants ; Le service qu'il fournit est le fruit d'un ensemble d'interactions entre son propre code et ses sous éléments. Il se peut d'ailleurs que chacun des sous éléments utilisés soient eux aussi des éléments composites, de manière récursive.

Un composant logiciel dans CBSE[4] doit avoir des propriétés pour garantir un bon système (Gao, Jacob Tsao, & Wu, 2003) définissent deux types de propriétés, celles de base qui se résument en :

- L'hétérogénéité : qui est une des principales caractéristiques de base des composants logiciels. elle permet une implémentation en utilisant différents langages, l'exécution

[4] Component Based System Environnement

sur différentes plates-formes, même dans des endroits multiples, sur de grandes distances.

- Modularité et encapsulation: Chaque composant encapsule un ensemble d'éléments de données qui sont étroitement liés et met en œuvre une logique fonctionnelle cohérente pour effectuer une tâche spécifique.
- Disponibilité du code source: Lors du développement de logiciels à base de composants, possibilité d'adopter des composants COTS chaque fois que des composants appropriés sont disponibles.
- Évolutivité: Le Plug-and-Play caractéristique des systèmes à base de composants permet aux composants d'être dynamiquement ajoutés, supprimés ou mis à jour sans être recompilés ou reconfiguré.
- Distribution: Avec le développement de l'Internet, des logiciels de plus en plus basé sur des composants sont maintenant distribués à travers les réseaux.
- Réutilisation: Un des principaux objectifs de l'ingénierie logiciel à base de composants est de promouvoir la réutilisation du logiciel, de sorte qu'il puisse améliorer la qualité des produits futurs et, en même temps, réduire les coûts de développement.

2.7 Avantages et limites

De tout temps la recherche évolue et innove sans cesse pour améliorer notre quotidien, dans tout domaine qui soit, et particulièrement dans les nouvelles technologies et l'informatique spécialement, et ce à travers des technologies nouvelles mais pas seulement, des méthodes, méthodologies[5] ou encore processus de développement. Et toute chose a ses avantages et ses limites, le développement à base de composant a les siens :

2.7.1 La Réutilisation

La réutilisation a de tout temps constitué une action clé de l'ingénierie des systèmes logiciels et des systèmes d'information. En 1968, (McIlroy M. , 1968) écrivait : « Ma thèse est que l'industrie logicielle est faiblement fondée et qu'un aspect de cette faiblesse est l'absence d'une sous-industrie de composants logicielle. Je vais aussi faire valoir qu'une industrie des composants pourrait être extrêmement utile, et expliquer pourquoi elle ne s'est

[5] La cartographie des méthodes ou tout simplement la méta méthode ou méthode des méthodes,

pas matérialisée. Enfin, je vais poser la question de la mise en service d'une «unité pilote» pour les composants logiciels».

Rendre réutilisable un produit de développement nécessite de disposer de langages conceptuels et opérationnels permettant l'expression de solutions génériques, adaptables, intégrables, composables, etc.

La réutilisation est un des objectifs phares du génie logiciel. De nombreux concepts et mécanismes associés ont été inventés pour réutiliser le code :

- Fonction et appel de fonction,
- Module et importation de module,
- Classe et héritage,
- Framework et paramétrage de Framework,
- Composant et assemblage de composants.

La réutilisation s'effectue en deux étapes (Fabresse, 2007): abstraction et utilisation. L'abstraction consiste à définir sous une forme abstraite et indépendante de tout contexte ce qui peut être réutilisé.

L'utilisation consiste à réutiliser dans un contexte donné une abstraction, ce qui peut nécessiter des adaptations et/ou du paramétrage. Un composant réutilisable est un fragment de produit ou de processus de développement réutilisable dans le développement de plusieurs systèmes. Ces composants peuvent être des composants logiciels comme les EJB (Enterprise Java Beans) de SUN ou être des composants conceptuels tels que les composants de domaine (Ramadour, 2000) (Ramadour, 2001).

La réutilisation fait augmenter la qualité et la productivité du logiciel, ainsi que la confiance dans ce logiciel. En plus, elle fait diminuer l'effort de production du logiciel. Ces caractéristiques ne sont pas indépendantes les unes des autres, elles font interactions entre elles.

Des limites existent aussi comme le manque du support de gestion.

- Le logiciel réutilisable n'est pas gratuit,
- La difficulté de trouver des ressources réutilisables,
- Quelques composants ne s'accordent pas avec la réutilisation,

- Ou encore et quelques fois un problème sérieux qui est la manque de certification (i.e. le composant n'est pas sur de manière formelle et standard)

2.7.2 La composition de composants

Dans le processus de développement par composants, chaque fois que des composants et leurs interactions sont définis, les composants réutilisables sont consultés pour identifier les composants potentiellement réutilisables. Il est possible que la fonctionnalité d'un composant issu de la conception ne puisse pas être assurée par un seul composant, mais par une combinaison de plusieurs composants. Dans ce cas, les composants doivent être assemblés de façon à ce que la structure générée apparaisse comme un seul composant.

La composition est l'idée forte qui a amené au concept de composant. En effet, chaque composant est considéré comme une brique de base, que l'on va relier à d'autres briques afin de concevoir un ensemble évolué capable d'effectuer des opérations complexes. L'idée de base est que l'on peut construire une application (un logiciel) comme on construit un circuit électronique. On connecte les composants les uns aux autres afin d'obtenir le comportement désiré.

Le but ultime du développement logiciel basé composant est l'assemblage effectué par les tierces parties. Pour ce faire, il est nécessaire d'être en mesure de spécifier les composants de façon à ce que nous pouvons raisonner sur leurs constructions et leurs compositions.

2.7.3 La structuration

Une limitation sérieuse survient dans les paradigmes objet dès que l'on veut pouvoir étendre une référence Tout ceci nécessite de modifier la représentation interne de l'objet .donc une reformulation et recompilation et restructuration.

La notion de composant logiciel apporte une amélioration notable à ce problème en externalisant les références, et en les décrivant explicitement sous la forme d'interfaces de sortie, par opposition, ou plutôt en complément, des interfaces d'entrée traditionnelles des objets et des procédures.

Les composants apportent une vision architecturale (Briot & Seghrouchni, 2009) (architecture logicielle qui présente une structuration) explicite de l'application, où l'on

s’intéresse à la logique du couplage entre les composants, indépendamment de leur implantation interne.

Parmi les quatre (04) niveaux de (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999) le niveau structurel ce qui implique la possibilité d’intégrer des contrats De manière à exprimer des indications sur les types de données (typage de données).

2.8 Modèles de composant

Le modèle de composant s'approche plus du modèle mental du développeur que les autres. On peut voir trois outils essentiels dans un composant (Cyril, 2003) :

- Des données,
- Des éléments qui transforment ces données,
- Des éléments qui assurent une connexion entre eux.

À première vue, un composant ressemble à un objet ou à un objet réparti, Mais contrairement à l’objet l’interface fait son apparition qui représente l’aspect extérieur visible et identifiant du composant, Le composant ne présente qu’une partie de tous les services qu’il peut offrir à l’entité qui l’invoque.

Un modèle de composants consiste en un ensemble de conventions à respecter dans la construction et l’utilisation des composants. L’objectif de ces conventions est de permettre 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.

Un modèle de composant est l'épine dorsale d'un système à base de composants. Il fournit un appui essentiel pour le développement des composants, la composition, la communication, le déploiement et l’évolution. Le modèle de composant est un facteur clé pour atteindre l'interopérabilité, l'évolutivité, la maintenabilité, ainsi que de nombreux autres attributs de qualité. Actuellement, il existe trois grands modèles de composants commerciaux : NET/COM/COM+, CORBA et EJB.

2.8.1 Un modèle de processus pour le logiciel à base de composants

De manière générale, le processus global d'ingénierie pour les logiciels à base de composants peut être divisé en plusieurs phases, comme le montre la **Figure 8**:

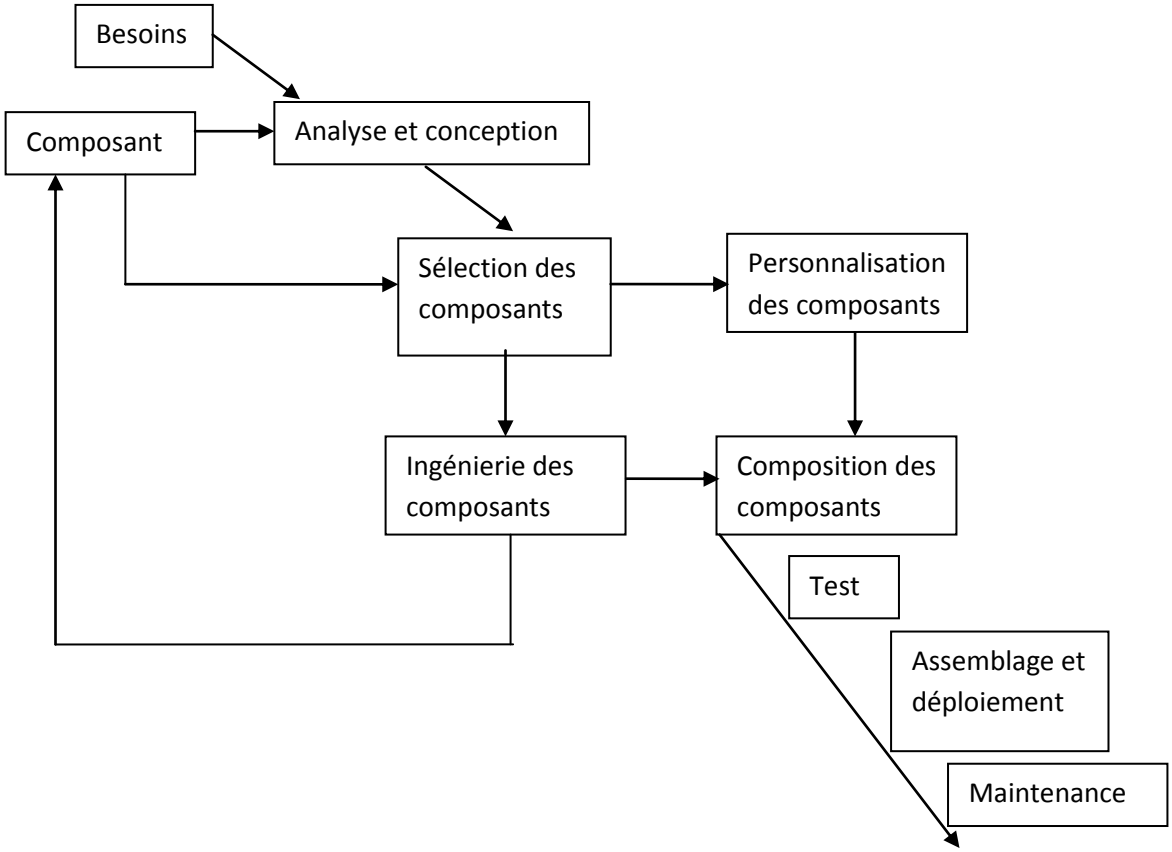


Figure 8 : Processus d'ingénierie pour les logiciels à base de composants **(Gao, Jacob Tsao, & Wu, 2003)**

Ces phases comprennent la sélection composante, la personnalisation et la composition, les tests, le déploiement et la maintenance.

2.8.1.1 La phase Analyse et conception

Cette phase est très importantes vue qu'elle mette en évidence les composants et leurs caractéristiques avec les interfaces qui servent d'interlocuteur des composants et dont on se sert pour une analyse structurelle ou comportementale permettant la phase d'identification des interactions des composants qui s'agit d'un processus visant à affiner les définitions des interfaces existantes, d'identifier comment les interfaces seront utilisées, et de découvrir de nouvelles interfaces et les opérations.

À ce niveau ou notre approche affine plus et ajoute une spécification permettant une analyse sémantique adaptée et un traitement adéquat avec notre modèle d'assemblage assisté et validé par les Agents.

2.8.1.2 Phase de spécification

Durant la phase de spécification des composants, nous allons définir précisément l'interface de composant et ainsi que les spécifications des dépendances entre les différentes interfaces du composant. Les spécifications des interfaces comprennent généralement les opérations qui seront effectuées dans les interfaces. Outre les spécifications pour chaque interface individuelle, la spécification d'un composant doit également préciser les contraintes et les dépendances entre les différentes interfaces.

Compte tenu de l'architecture des composants et les spécifications des composants, la sélection des composants est le processus de recherche de composants d'adaptation du référentiel de composants réutilisables et COTS. Il s'agit en fait d'un processus beaucoup plus complexe. Pour déterminer si deux composantes correspondent les uns avec les autres, les exigences fonctionnelles, les exigences non fonctionnelles et les contraintes de déploiement doivent tous être pris en considération.

2.8.1.3 Sélection de composants

Déterminer la liste des composants candidats, une liste des composants potentiels -en parcourant tous les composants réutilisables- sert à évaluer chaque composant candidat. En examinant les spécifications d'interfaces de manière basique (syntaxiquement) en plus sémantiquement à travers sa spécification et sélectionner le(s) composant(s) le(s) mieux adapté(s) à l'assemblage.

Même si le processus de sélection des composants pourrait être en mesure de trouver un groupe de composants candidats, une adoption directe nécessite souvent une correspondance parfaite des composants candidats à l'architecture des composants, la spécification des composants, et de l'environnement externe.

Nous nous intéressons essentiellement à cette phase dans le but d'assister la collecte d'informations sur les composants disponibles et d'améliorer la sélection de ceux d'entres eux qui répondent aux mieux au service requis par le client (le développeur) qui se matérialise par un l'assemblage du système global à partir des spécifications fournies en entrée.

L'implémentation dite aussi phase d'intégration. Cette phase d'intégration du système est la phase durant laquelle les composants sélectionnés sont assemblés pour créer le système.

2.9 Les principales familles d'approches à composants

Bien qu'il n'existe pas de classification générale des approches à composants, la littérature distingue souvent (Fabresse, 2007) (Pessemier, 2007) :

- Les approches industrielles comme COM(+)/.NET ou EJB,
- Les approches académiques comme les langages de description d'architectures (ADLs) (Unicon, C2, WRIGHT, ACME, etc.) Ou encore d'autres propositions telles que ArchJava et ComponentJ,
- Les normes (ou modèles de référence) comme CCMou UML 2.0.

Toutefois, elle ne permet pas de classer facilement toutes les approches à composants comme par exemple l'approche Fractal qui peut être classée dans les trois catégories puisqu'elle est réalisée dans le cadre d'un groupement entre industriels et chercheurs, que le modèle Fractal est une norme et que son implémentation de référence Julia est un langage.

2.9.1 Les modèles académiques de construction d'applications à base de composants

À l'instar des travaux sur l'ingénierie des composants, de nombreux modèles académiques sont apparus pour le développement d'applications à base de composants logiciels. Ces modèles académiques représentent des expérimentations sur des fonctionnalités précises dans le but de mieux comprendre les composants et de contribuer par la suite à la définition de modèles industriels.

2.9.1.1 Selon UML2.0

UML2.0 (OMG, 2002) spécifie un composant comme étant une unité modulaire, réutilisable, qui interagit avec son environnement par l'intermédiaire de points d'interactions appelés ports. Les ports sont typés par les interfaces : celles-ci contiennent un ensemble d'opérations et de contraintes ; les ports (et par conséquent les interfaces) peuvent être fournis ou requis. Le comportement interne du composant ne doit être ni visible, ni accessible autrement que par ses ports. Enfin, le composant est voué à être déployé un certain nombre de fois, dans un environnement à priori non déterminé lors de la conception (excepté au travers des ports requis).

Deux types de modélisation de composants dans UML2.0 : le composant basique et le composant composite (ou packagé). La première catégorie définie le composant comme un élément exécutable du système. La deuxième catégorie étend la première en définissant le composant comme un ensemble cohérent de parties (appelées parts, chacune représentant une instance d'un autre composant).

Dans UML 2.0, Les concepts de port, de connecteur et de diagramme d'architecture ont été introduits. UML permet aussi bien les descriptions structurelles des architectures que les descriptions comportementales via les protocoles notamment.

2.9.1.2 Selon ACCORD

Projet ACCORD (Projet ACCORD, 2003) et (ACCORD, 2002) définit le composant comme une entité logicielle qui réalise des interactions avec d'autres composants via des connecteurs, décrite selon trois niveaux:

- Le type du composant,
- La classe de composant. Un port (et aussi les interfaces associées),
- L'instance de composant qui est le résultat de l'activation du composant,

Et éventuellement :

- Un port (optionnel) permet d'accéder au connecteur comme s'il s'agissait d'un composant,
- Une prise sera reliée à un port d'un composant ou d'un connecteur.

2.9.1.3 ArchJava

Le modèle de composant d'ArchJava définit, comme la plupart des ADLs, les trois concepts : composant, connecteur et configuration.

ArchJava se situe à la frontière entre le langage de description d'architectures et les modèles de construction d'applications à composants. Un obstacle de taille qui se manifeste par l'absence de modèle abstrait de description du comportement des composants. Celui-ci est directement exprimé en Java, et ceci ne lui permet pas de faire l'objet d'une abstraction d'analyse et spécification avec test abstraitement soit hors du contexte environnemental.

ArchJava spécifie uniquement la structure d'une application. Son comportement ne l'est pas. Dans ce sens, ArchJava ne propose qu'une description des interfaces de niveau un (01) dans la taxonomie de (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999). Ainsi ArchJava est l'une des rares approches à tenter d'intégrer dans l'implémentation les descriptions architecturales normalement décrites par les ADLs (Fabresse, 2007).

2.9.1.4 Fractal

Malgré l'adoption des modèles à composants industriels comme EJB ou CCM (OMG, 2002), la nécessité d'avoir un modèle plus léger et plus proche des concepts des langages de programmation s'est fait ressentir. Le modèle de composant **Fractal** (Bruneton, Coupaye, & Stefani, 2004) réalisé dans le cadre du consortium ObjectWeb par France Telecom R&D et par l'INRIA cherche à combler ce besoin.

Les interfaces jouent un rôle central dans Fractal. Elles appartiennent à deux catégories distinctes : les interfaces métier et les interfaces de contrôle.

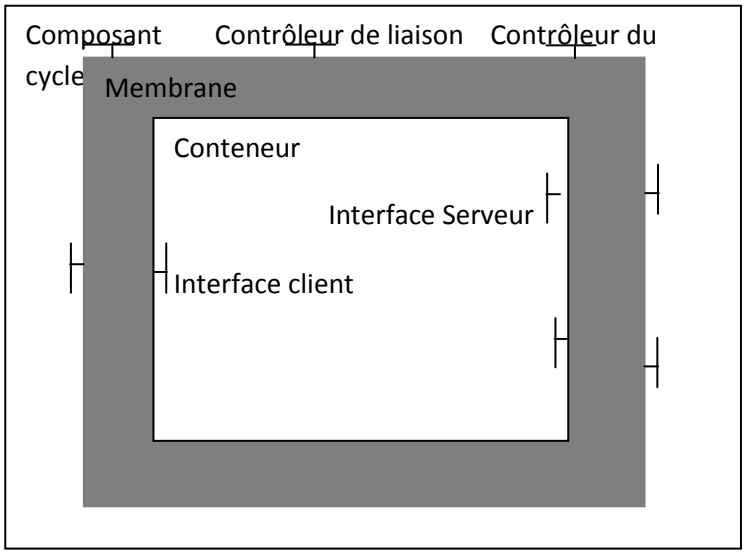


Figure 9 : Modèle de composant Fractal **(Bruneton, Coupaye, & Stefani, 2004)**

Le modèle étant fortement typé, le type d'une interface serveur doit être obligatoirement du type ou du sous-type de l'interface cliente à laquelle elle est reliée. La liaison est assimilable à un connecteur implicite, elle ne possède pas de comportement propre.

Les types de composants sont simplement des ensembles de types d'interface de composant. Le système de type est équipé d'une relation de sous-typage qui incarne les contraintes pour s'assurer substituabilité des composants.

Fractal est sans conteste une contribution majeure dans le domaine des composants. Ce modèle abstrait est indépendant de toute technologie. Néanmoins, le modèle Fractal n'intègre pas explicitement la notion de port. Elle est en réalité complètement intégrée dans la notion d'interface.

2.9.1.5 SOFA

SOFA (*SOFtware Appliance*) est un modèle de composant développé par l'université Charles (Prague). Son objectif est la construction d'application à partir d'une composition de composants.

À partir des descriptions de comportement des éléments d'une architecture SOFA, il est possible de vérifier la compatibilité sémantique d'un code source, c'est-à-dire, vérifié si celui-ci respecte la description de comportement du frame. De même, il est possible de vérifier si une architecture respecte la description de comportement de son frame.

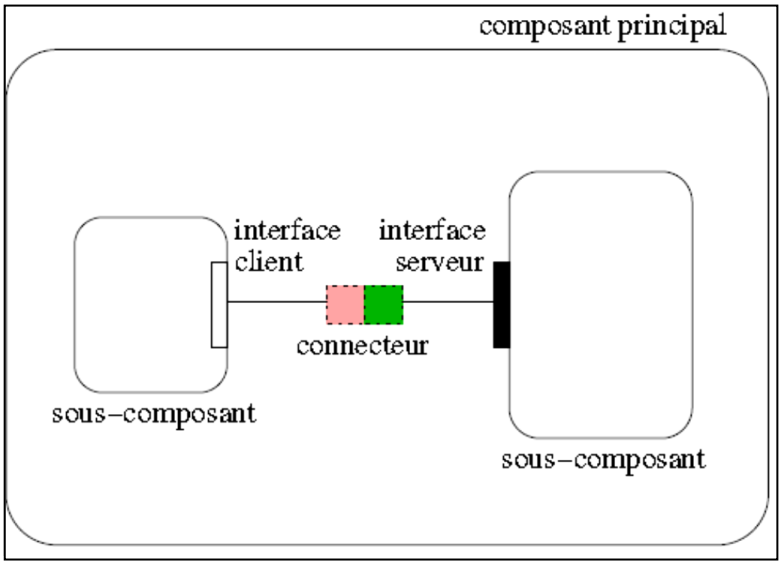


Figure 10: Deux composants SOFA reliés par un connecteur **(Fabresse, 2007)**

Le principal attrait de SOFA repose sur une description du comportement des composants permettant une analyse de l'assemblage et une analyse de la conformité d'une implantation (architecture) par rapport à son type (frame). Il propose dès lors un contrat de niveau 3 selon la taxonomie de (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999) pour la description des interfaces de composant. Cependant, le contrat de niveau 2 est inexistant dans SOFA, il n'y a pas de possibilité de préciser ce que s'engage à respecter le composant. Aussi il propose une approche statique de la description d'une architecture logicielle.

Le modèle SOFA ne distingue toutefois pas les concepts de port et d'interface qui sont pourtant assez différents et complémentaires comme dans d'autres modèles.

2.9.1.6 BILAN SUR LES MODELES DE COMPOSANT ACADEMIQUES

L'étude des différents modèles à composants récents montre l'intérêt de conserver la notion de composant logiciel et d'architecture logicielle dans le développement des applications. Les composants logiciels apportent une unité de structuration du code de l'application, une unité de réutilisation et de déploiement ainsi qu'une unité granulaire offrant une vue plus ou moins précise des éléments constitutifs de l'application. La prise en compte des connecteurs permet de séparer les aspects liés à la mise en œuvre de la communication des aspects métiers de l'application. Les capacités d'un modèle de composant à pouvoir reconfigurer dynamiquement une application, c'est-à- dire modifier les éléments constitutifs d'une application et leur interaction est un pas en avant vers la construction d'applications plus flexibles.

Finalement, on constate que chacune de ces approches a redéfini son propre langage de description d'architecture bien que l'on trouve facilement de nombreuses similitudes entre ces langages. Comme ces langages ont souvent une place centrale dans ces modèles de composant académiques, il serait intéressant de disposer d'un modèle de description d'architecture abstrait commun afin de favoriser l'interopérabilité entre ces approches de construction d'applications à base de composants.

La plupart des approches à composants ne couvrent pas l'ensemble du cycle de développement d'une application, depuis sa spécification, considérée comme le plus haut niveau d'abstraction, jusqu'à son code binaire, en plus certaines approches se spécialisent pour un domaine particulier.

2.10 Conclusion

Les composant représentent une avancée très sensible surtout en matière de réutilisation qui sert un assemblage plus cohérant et surtout une vérification automatisée vue la taille des systèmes distribués actuels, et malgré la diversité des types et des niveaux d'abstraction cela ne diminue pas de l'intérêt des développeurs et concepteurs envers ce paradigme de programmation qui à travers des standards et des méthodes formelles de plus en plus efficaces et sures résiste aux complexités de plus en plus persistantes (inexistence de méthodologies

globales, manque de standards tout le long du cycle de vie, variation des niveaux d'abstraction donc soit trop spécialisé ou trop abstrait…).

Et dans ce contexte fluctuant nous cherchons à avoir une vision plus pragmatique et certainement formelle pour ne pas rester évasif et implicite dans un domaine qui exige d'être explicite et formel. C'est ce que le nous décrivons dans le chapitre suivant.

Chapitre III : Etat de l'art sur les formalismes de description des composants

3.1 Introduction

La description formelle représente un moyen important dans le développement des systèmes informatiques, dans le génie logiciel la description comportementale permet la simulation du comportement dans les toutes premières phases de conception ce qui rend l'anticipation de détection d'erreur possible et de l'analyse des différentes possibilités d'aménagement et d'implémentation des systèmes.

Plusieurs langages et technologies de programmation permettent la description comportementale (la façon dont les méthodes sont enchainées). La spécification comportementale des systèmes est un champ de recherches à part entière. En ce qui concerne plus spécifiquement les systèmes distribués, de nombreux chercheurs ont reconnu l'importance de fournir des spécifications comportementales adaptées au modèle comme le modèle objet de l'OMG. L'usage de méthodes formelles peut au moins aider à maintenir « le niveau actuel de qualité du logiciel », alors que la complexité des systèmes à construire s'accroît du fait de la présence de composants distribués.

Cependant, ces méthodes ne sont pas assez bonne et adaptées pour la validation et l'assurance de qualité des composants réutilisables et des blocs de construction en génie logiciel à base de composants, car ces méthodes de validation, les critères et les processus de contrôle de la qualité ne portent pas toujours sur les questions que posent (Gao, Jacob Tsao, & Wu, 2003):

- Comment peut-on valider la réutilisation des composants?
- Comment peut-on valider l'interopérabilité des composants?
- Comment peut-on assurer qu'un composant développé suit un modèle de composant?
- Comment vérifier qu'un composant soit correctement déployé et paqueté?
- Comment peut-on vérifier si un composant est personnalisé convenablement et correctement?

3.2 Spécifications formelles dans le processus de développement logiciel

L'analyse des besoins ou spécifications de tout système informatique est une définition abstraite qui ne contient aucune information concrète des choix technique et technologiques d'implantation (choix technologiques, variantes algorithmiques, architecturales, etc.). Elle

décrit une description abstraite, selon plusieurs auteurs cette description concerne trois aspects du produit : l'aspect **fonctionnel**, l'aspect **structurel**, et l'aspect **comportemental**.

- Description fonctionnelle. La description fonctionnelle spécifie les propriétés du produit ainsi que la sémantique statique de chaque fonction du produit (service offert et requis pour un composant par exemple).
- Description structurelle. La description structurelle spécifie les interfaces (services offerts et requis pour un composant exemple) du produit. Elle spécifie également les relations à travers lesquelles le produit interagit avec son environnement ainsi que toute structure interne du produit visible au niveau de ses interfaces.
- Description comportementale. La description comportementale complète les descriptions structurelle et fonctionnelle du produit en définissant les mécanismes de gestion des contraintes temporelles et de synchronisation et en spécifiant la sémantique dynamique des fonctions du produit.

La séparation des spécifications se fait de plusieurs manières dont celle du projet Accord (ACCORD, 2002) qui utilise les connecteurs comme moyen de reléguer les spécifications non fonctionnelles. Ce qui facilite leur traitement (spécificité fonctionnelle) et sépare les préoccupations (diviser pour mieux régner).

Par exemple, Les exigences et la conception des systèmes -comme le développement des systèmes basés sur le modèle en cascade- sont exprimés en détail et soigneusement analysés et contrôlés avant leur mise en œuvre. On peut voir que la spécification devient formelle après même le commencement de la partie conception et ce prouvant un prolongement de la spécification jusqu'à une étape avancée dans le cycle de développement, c'est dire sa pertinence.

La **Figure 11** montre les étapes de spécification du logiciel et son interface avec le processus de conception. Les étapes de spécification ne sont pas indépendantes et ne sont pas nécessairement développées dans l'ordre indiqué.

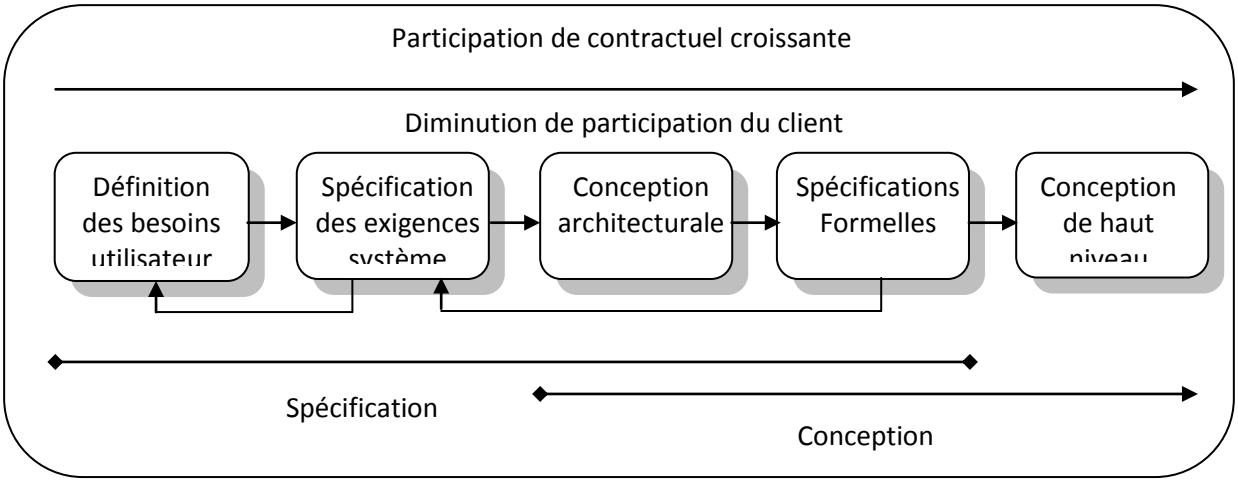


Figure 11: Spécification et conception **(Sommerville, 2007)**

Lorsque l'on élabore les spécifications en détail, la compréhension de cette spécification augmente. La création d'une spécification formelle nous force à faire une analyse détaillée des systèmes qui révèle souvent des erreurs et des incohérences dans la spécification informelle des exigences. Cette détection d'erreur est probablement l'argument le plus puissant pour développer une spécification formelle (Hall, 1990). Elle vous aide à découvrir les problèmes d'exigences qui peuvent être très coûteux à corriger plus tard.

3.3 Spécifications formelles pour les composants logiciels

La recherche dans les langages formels de spécification a commencé au début des années soixante (60). De nombreux langages résultent de ces efforts de recherche. Certains d'entre eux ont conservé l'attention de la communauté de la recherche au fil des ans, et a eu des essais réussis dans certaines applications industrielles.

De manière générale, un composant est un logiciel qui rend un service au moyen d'une ou de plusieurs *interfaces* externes. Ceci implique d'associer au composant des spécifications précises tant fonctionnelles que non fonctionnelles, Les spécifications doivent également porter sur les comportements du composant.

Les spécifications formelles s'imposent progressivement comme une condition essentielle à un bon développement logiciel. Leurs avantages principaux sont la rigueur dans la spécification du problème, la preuve de propriétés et une conception méthodique, automatisable en partie.

Dans les années 1980, de nombreux chercheurs en génie logiciel ont proposé que l'utilisation de méthodes formelles de développement fût la meilleure façon d'améliorer la qualité des logiciels (Sommerville, 2007). Ils ont fait valoir que la rigueur et l'analyse détaillée qui sont une partie essentielle de méthodes formelles pourraient conduire à des programmes avec moins d'erreurs et qui étaient plus adaptés aux besoins des utilisateurs. Ils ont prédit qu'au 21e siècle, une grande partie des logiciels serait développé en utilisant des méthodes formelles.

Deux approches fondamentales de la spécification formelle ont été utilisées pour rédiger des spécifications détaillées pour les systèmes de logiciels industriels. Ce sont:

1. Une approche algébrique où le système est décrit en termes d'opérations et de leurs relations.

2. Une approche basée sur un modèle où un modèle du système est construit en utilisant des constructions mathématiques telles que des ensembles et les séquences et les opérations du système sont définis par la façon dont ils modifient l'état du système.

Différents langages de ces familles ont été développés pour spécifier des systèmes séquentiels et simultanés. Là ci-dessous montre des exemples des langues dans chacune de ces classes.

	Séquentiel	Concurrent
approche Algébrique	Larch, OBJ	LOTOS
approche basée sur un modèle	Z, VDM, B	CSP, réseaux de Pétri

Figure 12 : Langages de spécification formelle **(Sommerville, 2007)**

Une spécification formelle exige d'être exprimée dans un langage à syntaxe et sémantique précises, baptisée souvent sur des théories mathématiques ce qui leur confère une base solide et formelle permettant la validation automatisée ou du moins assistée.

On utilise des langages formels, car il semble être le moyen le plus facile d'écrire des spécifications qui sont à la fois précises, claires et concises. Cela n'est guère surprenant. Il

n'est pas fortuit que ces activités aussi diverses que la chimie, échecs et la musique ont tous leurs propres notations formelles.

Un grand avantage de spécification formelle est que les outils peuvent être utilisés pour aider à détecter et isoler nombre d'erreurs.

3.4 La classification des méthodes formelles

La classification de ces méthodes est subjective et plusieurs auteurs ont fait l'effort de les ranger selon plusieurs critères (type de raisonnement (structure/propriété), aspect du système (statique/dynamique/fonctionnel), langage…).

Ceci dit un semblant de consensus s'est formé avec la classification de (Alagar & Periyasamy, 1998) basée sur (Guttag, Horning, Garland, Jones, Modet, & Wing, 1993), qui définissent quatre (04) concepts principaux sur des bases mathématiques pour un langage de spécification formelle qui sont: l'algèbre, la logique, la théorie des ensembles et algèbre relationnelle et leurs combinaisons éventuellement.

Notre travail se penche sur les composants systèmes et ceci nous guide vers une classification orientée plus vers ce paradigme et qui donne trois axes principaux qui sont définis pour les spécifications formelles des composants ceux issus d'une sémantique à base d'automates à états finis, ceux dérivant du π-calcul, et les contrats dérivant des pré- et postconditions (Cyril, 2003).

3.4.1 Automates d'interface et supposition/garantie

Des automates I/O qui les définissent, qui sont des automates avec des transitions qui indiquent les entrées ou sorties effectuées ; ainsi, les séquences d'entrées/sorties autorisées sur l'interface du composant sont données par ces automates.la vérification si un composant implante bien sa spécification donnée par des automates par une relation de raffinement, cette vérification est faite en termes de simulation. Un automate P raffine un automate Q si toutes les entrées de Q peuvent être simulées par les entrées de P (ou : un message reçu par Q peut être reçu par P), et si toutes les sorties de P peuvent être simulées par les sorties de Q (si P envoie un message, Q peut envoyer le même message).

La compatibilité entre interfaces est obtenue en composant les automates correspondant de telle sorte que les actions duales (une entrée correspondant à une sortie) deviennent des

transitions internes. Les interfaces sont considérées comme compatibles tant qu'il existe un environnement qui peut interagir avec l'automate résultant de la composition.

La sémantique des automates d'interface est similaire aux suppositions/garanties (assume/guarantee) : les entrées correspondent aux suppositions quant aux actions de l'environnement, et les sorties sont les garanties que le composant doit respecter.

L'idée des pré- et postconditions est appliquée aux processus, Les notions de suppositions/garanties sont d'ailleurs à la base d'un raisonnement pour la composition de spécification (Abadi & Lamport, 1993). La compatibilité des spécifications de deux processus doit montrer que les garanties de l'un assurent les suppositions de l'autre.

Il est à noter que le paradigme suppositions/garanties mène naturellement à la spécification d'un contrat.

3.4.2 Typage comportemental

Un type comportemental représente une abstraction du comportement de l'entité considérée (objet, composant ou autre). Le service proposé par le type est alors appelé non uniforme, car il varie selon le contexte. Le typage comportemental permet donc de prendre en compte ces aspects dynamiques. L'algèbre des processus comme (π-calcul, CSP) est la source d'inspiration pour la syntaxe des types, Elle s'en distingue par la définition de séquences valides d'appels de méthodes sur toutes les interfaces de composants.

L'algèbre de processus est fondée sur une expression décrivant un ensemble de traces (séquences d'évènements). Appliqué à un composant, un événement est une abstraction d'un appel de service ou d'une réponse à un appel (Saudrais, 2010).

Deux tendances principales qui dominent l'orientation en terme type comportemental, La première tendance dérive des types réguliers, en général pour typer les ports ; La deuxième tendance des types comportementaux s'intéresse aux processus. Des recherches sur la première tendance qui dérive des types réguliers ont débouché sur des systèmes de type qui assurent que tout message envoyé sera compris (et consommé) par le récepteur traitant les erreurs de message, et qui traitent aussi la visibilité des interfaces (publique/privées).

Une extension d'une idée fondée par (Honda, Vasconcelos, & Kubo, 1998) et reprise par (Gay & Hole, 1999) puis (Gay & Hole, 2003) et (Vallecillo, Vasconcelos, & Ravara, 2005)

qui est les types de session qui sont associés aux canaux de communication, et spécifient le protocole d'interaction qui a lieu sur ce canal , cette piste représente notre effort à l'amélioration d'une modélisation à base de session type qui va servir de base pour une meilleure prise en charge d'un assemblage assisté et validé par le biais d'agents qui fourniront le moyen pragmatique de cette tache.

3.4.3 Contrats

Concevoir un système de nos jours est un défi tant méthodologique, conceptuel ainsi que réalisation, et ceci est lié à plusieurs facteurs et critères (exigences de fiabilité, cohérence, maintenabilité, évolutivité, sécurité….) ; en permettant la prévision et l'optimisation des propriétés fonctionnelles et non-fonctionnelles des systèmes conçus est toujours un défi ouvert, Une théorie globale est toujours inexistante aujourd'hui.

La logique de Hoare (Hoare, 1969). En 1969, a inspiré plusieurs auteurs qui se sont penchés pour en extraire des applications dans le géni logiciel et des techniques de conception, vérification et validation des systèmes. Destinées à fiabiliser les applications et faciliter leur développement.

La conception par contrat est basée sur la théorie et les techniques de spécification formelle et de vérification qui remonte aux années 1960.et Bertrand Meyer est parmi les premiers si ce n'est pas le premier qui a introduit les contrats dans le développement des systèmes, et qui a publié un premier manifeste (*Design by Contract*) en 1992 (Meyer, 1992)

Meyer dans (Meyer, 1992) précise qu'il faut une description adéquate et précise des propriétés des interfaces qui se définit en contrat, de cette manière clients et serveurs de services (un composant propose un ou plusieurs services selon son degré de granularité) sauront manipuler et traiter de manière transparente mais formelle donc précise. Dans ce sens qu'il doit être possible de les analyser et de les interpréter sans ambiguïté.

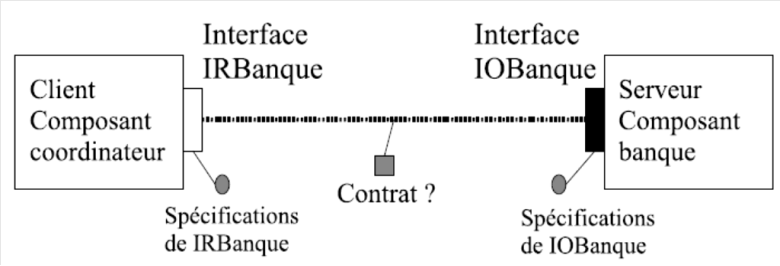


Figure 13: Format d'un contrat d'interaction **(Legond-Aubry, 2005)**

3.4.3.1 La description d'un composant

La description d'un composant est un moyen essentiel dans la conception des systèmes surtout de grande envergure, et à travers les contrats elle permet de préciser les fonctionnalités des contractants, sachant que les composants de par leur principe de réutilisation s'utilisent en assemblage.

L'assemblage et les contrats sont indissociables du paradigme composant, Lorsqu'un composant est assemblé avec d'autres, leurs contrats d'utilisation sont confrontés, négociés et parfois adaptés pour satisfaire aux différentes hypothèses et contraintes. Et ce selon le niveau (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999) dans lequel on travail.

Un contrat peut en effet se décomposer en quatre niveaux (Projet ACCORD, 2003) (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999) qui sont : syntaxique, comportemental, synchronisation et qualité de service.

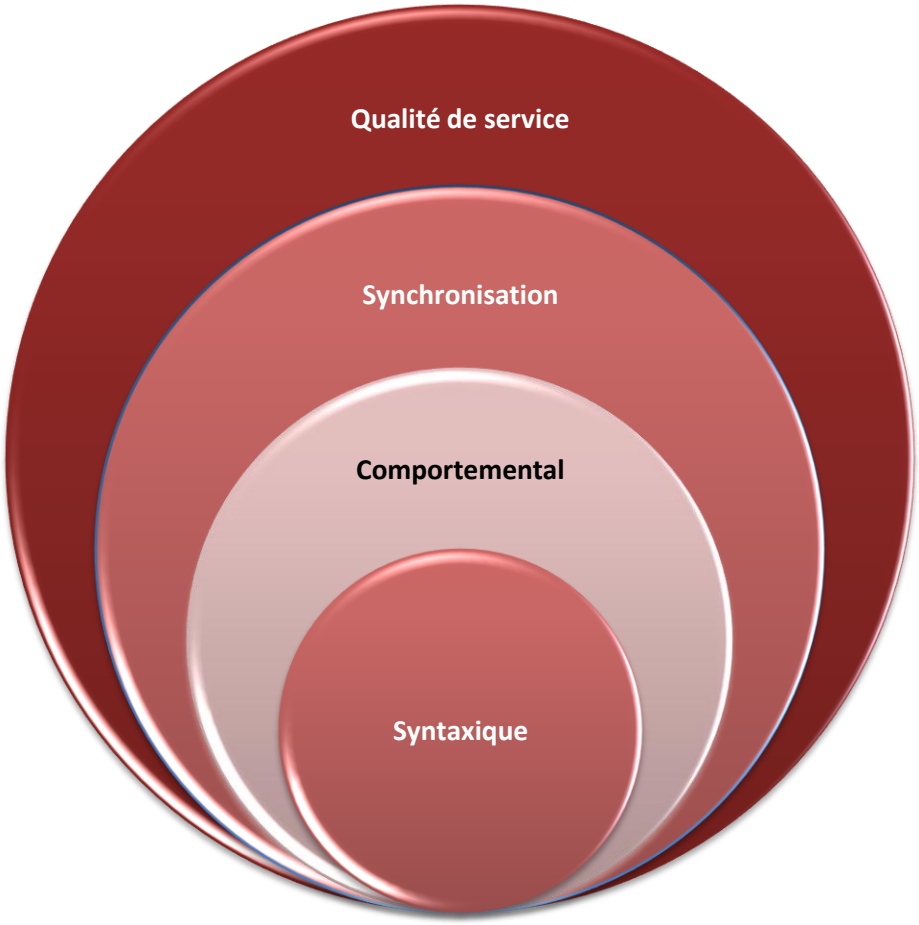


Figure 14: Les quatre niveaux de description d'un contrat **(Beugnard, Jézéquel, Plouzeau, & Watkins, 1999)**

Et c'est ce qui nous intéresse dans notre approche à base de contrats et que nous développerons avec plus de détails dans la suite de ce mémoire. L'utilisation des concepts contrats dans le développement d'une application entraîne de très nombreux bénéfices (Saudrais, 2010) :

- Une meilleure séparation des préoccupations : si les approches par contrats cherchent avant tout à favoriser la réutilisation de modules logiciels, permet de clarifier le code des méthodes, Ce code contient alors uniquement le code fonctionnel lié aux méthodes.
- Possibilité d'outillage : grâce à une formalisation précise des langages de contrats, il est possible d'outiller convenablement l'approche afin de garantir statiquement certaines propriétés, de générer du code pour garantir à l'exécution le respect des contrats.
- Documentation : le contrat est une documentation naturelle, car il répond à la question « qu'est-ce que cet élément requérant un service est supposé faire ? » en plus de la description des interfaces.

L'établissement d'un contrat a lieu lors de l'assemblage de composants. Ce contrat est construit à partir d'informations fournies par les composants. Ces informations sont contenues dans un contrat de composition attaché aux interfaces du composant.

Lors de l'assemblage de composants, les contrats de composition présents sont comparés et forment un contrat s'ils sont compatibles. La vérification de compatibilité se fait niveau par niveau (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999). Si l'une des parties ne satisfait plus le contrat, celui-ci est violé et peut être renégocié selon le niveau du contrat falsifié.

3.4.3.2 Cycle de vie du contrat

Le contrat a un cycle de vie que plusieurs auteurs ont essayé de définir, la notion de demi contrat utilisée par (Legond-Aubry, 2005) avec propagation de ce dernier comme dans le modèle ACCORD figure ci-dessous

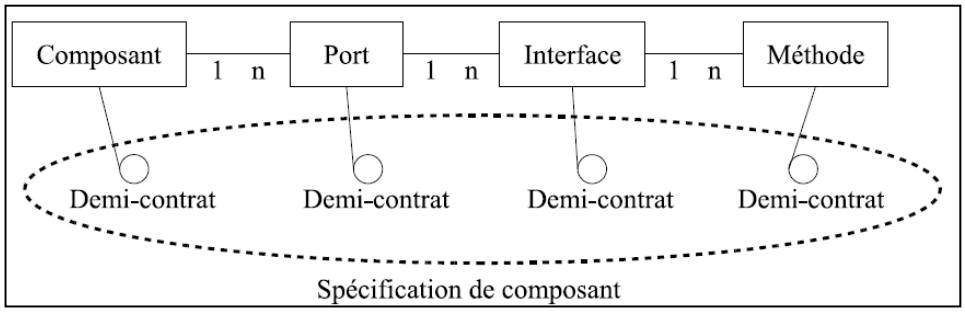


Figure 15: Propagations des contraintes au sein d'un composant ACCORD **(ACCORD, 2002)**

(Saudrais, 2010) a défini un cycle de vie général intégrant tous les niveaux et qui utilise la négociation comme moyen d'atteindre l'assemblage souhaité ou dans le cas échéant le plus proche possible avec des compromis que nous nous étudierons pas vu que nous travaillons sur une Vérification d'assemblage fonctionnel automatisée.

L'accès à un système de fichiers par exemple (*open* + *read* + *close*) | (*open* + *write* + *close*) impose une restriction qui est d'interdire une lecture et une écriture simultanément ce qui évite un blocage mais qui n'assure pas l'ordre souhaité (la sémantique comportementale de la description).

Les modèles de contrat appliqués aux composants adressent le problème de la garantie de leur composition bien qu'ils ne la contraignent pas toujours explicitement. Il s'agit en effet parfois d'évaluer la compatibilité de contrats portés par des composants associés et non de confronter au sein d'un contrat leurs spécifications. Ils présentent toutefois souvent à des degrés variables des considérations architecturales.

3.4.4 Sessions et types de session

Les types de session ont été initialement formulés pour les langages étroitement basé sur le calcul de processus. Depuis, l'idée a été appliquée aux langages fonctionnels (Gay, Vasconcelos, & Ravara, 2003), (Vasconcelos, Gay, & Ravara, 2006), systèmes d'objets à base de composants (Vallecillo, Vasconcelos, & Ravara, 2005) (Vallecillo, Vasconcelos, & Ravara, 2003), les langages orientés objet (Dezani-Ciancaglini, Yoshida, Ahern, & Drossopoulou, 2005), (Hu, Yoshida, & Honda, 2008), les services des systèmes d'exploitations et les systèmes axés sur les services plus généraux . Les types de session ont également été généralisés à partir des systèmes bipartis aux systèmes multi-partis (Gay S. J., Vasconcelos, Ravara, Gesbert, & Caldeira, 2010),

(Honda, Vasconcelos, & Kubo, 1998) Initialement introduisent les types de session pour décrire des protocoles de service des objets. La Compatibilité et la substituabilité des tests de protocoles sont définies par la relation de sous-typage définie par (Gay & Hole, 1999) pour les types de session.

Un exemple d'un serveur pour les opérations mathématiques est pris pour présenter le formalisme, qui offre un choix entre l'addition et la négation. Toutes les communications ont lieu sur un seul canal de session appelé X, dont le type de session est (Gay & Hole, 1999).

Exemple: S = &<plus : ?[int]. ?[int]. ![int]. end, negate: ?[int] . ![int]. end>. (Le constructeur &<..> Précise que le choix est offert).

Dans (Carbone, Honda, & Yoshida, 2007) rapportent les deux paradigmes différents pour la description de la communication comportementale (globale et locale) en utilisant un calcul formel basé sur les types de session. Ils parviennent à cartographier ces différents points de vue tout en préservant les structures de type en explorant une théorie de la projection du point final. Sur un papier plus tard (Carbone, Honda, & Yoshida, 2008), ils créent un système de type qui permet des exceptions structurées basées sur des types de session qui garantit la coordination de la gestion des exceptions entre les pairs communicants.

En suivant les travaux (Hu, Yoshida, & Honda, 2008) présentent une implémentation d'un langage et d'exécution pour la programmation distribuée basée sur la session avec des fonctionnalités supplémentaires comme la délégation de la session et le sous-typage.

(Vallecillo, Vasconcelos, & Ravara, 2005) Proposent une extension de la description des interfaces de composants logiciels d'inclure des informations comportementales grâce à l'utilisation de types de sessions. Par conséquent, ils décrivent comment une interaction complexe entre les composants deux-à-deux peut être fournie dans une spécification de haut niveau.

3.4.4.1 Principes de base

Deux processus prêts à échanger en toute sécurité des messages doivent s'accorder sur un protocole à adopter lors de l'interaction, pour être sûr que le déroulement de la communication ne soit pas interrompu.

L'utilisation de types de sessions a quelques possibilités intéressantes car elles permettent de réduire la complexité lorsqu'on vérifie la compatibilité de communication, même si cela se traduira par une perte de flexibilité en raison de leur composition par paires. En raison du fait que chaque élément de la paire contient la définition du comportement inverse (duale), la nécessité de décrire explicitement cela peut devenir un fardeau pour les utilisations de l'objet (local) et que ces types sont mieux adaptés à des compositions comportementales plus globales. Dans la théorie des types de sessions la spécification du protocole est un type associé à au canal de communication (Giachino, 2009), qui décrit la séquence et de la direction des données échangées.

Nous adaptons les types de sessions qui sont des protocoles d'interfaces partiels pour le modèle comportemental que nous proposons composé de contrats permettant ainsi de vérifier par cela le sous typage du service requis par l'utilisateur et le système composé à travers l'assemblage de composants sélectionnés auparavant.

Dans les systèmes distribués, il est commun pour la communication entre deux processus qui consiste en une composition d'un dialogue structuré décrit par un protocole qui spécifie le format et la direction de chaque message dans une séquence. Ce point de vue de la communication structurée ne correspond pas bien à un système de type qui exige de chaque canal de diffuser des messages d'un seul type. Pour résoudre ce problème, (Honda, Vasconcelos, & Kubo, 1998) ont introduit les types de sessions, qui permettent aux séquences non-uniformes, mais structurées de messages d'être spécifiées. Par exemple, le type (? [INT].! [Bool].Fin) décrit un canal qui peut être utilisé pour recevoir un nombre entier, puis envoyer un booléen, puis doit pas être utilisé à nouveau.

SERVEUR MATHEMATIQUE BASIQUE

Prenons l'exemple de (Giachino, 2009), un service qui donne deux nombres entiers, les réponses si ils sont égaux ou non. Nous pouvons décrire le comportement de communication de ce service dans la façon dont nous venons de le faire:

"Le service (EGALITE) prévoit deux entiers et retourne un booléen.". Ainsi, un client approprié pour interagir avec le service ci-dessus est celui qui est capable de donner les deux chiffres et attend une réponse booléenne. "Un client du service (EGALITE) doit fournir deux entiers et obtiendra un booléen".

Le service et le client savent que, s'ils se comportent comme spécifie par la description ci-dessus, c'est qu'ils suivent le protocole d'interaction, leur communication sera couronnée de succès. Et nous disons que le client dispose d'un protocole dual au regard du protocole du service, Ainsi, ce genre de description est leur comportement public, tout client avec le comportement ci-dessus peut interagir avec n'importe quel service avec le comportement dual.

Les types de session spécifient exactement ce genre d'information comme un type associé au canal sur lequel le service/client est à l'écoute. Il enregistre le type des données échangées et leur direction, dire si les données doivent être envoyées ou reçues. Par exemple, un canal de type (!Int) précise que le processus enverra un entier sur ce canal. Dualement, un canal de type ?Int précise que le processus s'attend à recevoir un nombre entier sur ce canal.

Les types de session spécifient une information importante au sujet des données à échanger et qui offrent des garanties sur la communication entre deux partenaires qui communiquent suivant des comportements duaux. Cette communication est en fait appelée session et il s'agit d'un échange de données sûr, privé et structuré. Notons que le type de session spécifie seulement le comportement d'interaction, chaque informations sur le calcul local ne se produit pas dans le type de session.

3.4.4.2 Propriétés des types de session

Un autre élément important de l'interaction en session est le choix, qui représente la possibilité de l'un des deux processus communicants pour choisir une option particulière parmi celles qu'offre son partenaire. Les choix sont basés des étiquettes et permettent aux processus de s'accorder sur le chemin que la communication doit suivre. Deux concepts s'y ajoutent :

***Le branchement*:** le branchement symbolise l'union des services proposés, le processus offre différentes suites possibles en fonction du choix de son partenaire. Chaque branche est étiqueté, et le processus s'attend à recevoir une étiquette de son partenaire, de savoir comment continuer.

La sélection: symbolise la possibilité pour le client de choisir tel ou tel message à envoyer, le processus sélectionne une branche et communique son choix pour le partenaire, afin que les deux puissent continuer à interagir en toute sécurité. Le choix se fait par l'envoi de l'étiquette correspondant à la branche choisie.

Le système de type des sélections et branchements assure que les canaux de communication sont utilisés correctement (ce qui revient à la détection d'erreurs (message non compris).

3.4.4.3 Types de session et composition de composants

Nous utilisons le concept de types de session pour décrire le comportement dynamique des composants (au sens séquencement). Les Sessions sont des spécifications partielles de protocole, dans lesquelles nous prêtons seulement attention à l'interface comportementale qu'un composant présente à un autre. Cela permet la spécification modulaire du comportement des composants, offrant plus que juste des informations de signature et de permissions des définitions précises des tests de compatibilité et de substituabilité, et à un coût de calcul plus faible que d'autres vérifications sémantiques (et donc de l'utilité pratique).

En outre, les types de session sont des types, et donc pris en charge par une discipline de type. C'est un élément clé de la méthode de structuration qui effectue des contrôles de type entre les sessions, permettant ainsi des tests de compatibilité puissant entre les composants.

La notion de «typabilité[6] » d'un programme développée par (Vallecillo, Vasconcelos, & Ravara, 2003) veille à ce que les deux composants éventuellement communicants possèdent toujours des modes de communication compatibles. Aussi, permet la définition de (contrôle de substituabilité des composants) basé sur le concept de sous-typage de session, qui est un simple calcul dans la plupart des cas, à la différence des tests exponentielles connus comme problème NP-complets qui résultent de l'utilisation des traces ou des algèbres de processus dans la description du protocole.

3.5 Le système à base des types de session

Initialement (Honda, Vasconcelos, & Kubo, 1998) introduisaient les types de session pour une description des interfaces permettant une communication structurée La compatibilité et la substituabilité des tests de protocoles ont été défini par (Gay & Hole, 2003) par la relation de sous-typage définie pour les types de session.

(Vallecillo, Vasconcelos, & Ravara, 2003) Puis (Vallecillo, Vasconcelos, & Ravara, 2005) ont complété l'œuvre par la théorie de sous-typage pour les types de session et adapté pour spécifier les composants logiciels distribués. En introduisant la notion de compatibilité

[6] Qui a trait au type

entre les différents types de sessions, et de prouver certaines de ses propriétés puis ils ont étudié comment les types de session peuvent être appliqués avec succès (terminaison), non seulement sur le plan théorique, mais aussi dans un environnement commercial.

(Gay S. J., Vasconcelos, Ravara, Gesbert, & Caldeira, 2010) Étendaient les travaux antérieurs sur les types de session pour langages orientés objet distribués.et la communication entre les processus dans un système réparti est souvent constituée d'un dialogue structuré, selon un protocole qui définit le format des messages échangés.

3.5.1 La compatibilité

Sur la base de la relation de sous-typage, nous sommes enfin en mesure de définir les concepts de substituabilité et la compatibilité entre les différents types de sessions.

3.5.1.1 Définition 1 (Vallecillo, Vasconcelos, & Ravara, 2005)

Soit T et S des types de session. Tel que:

i T peut se substituer à S en toute sécurité, si $T \leq S$ qui se lit T est sous type de S;

ii T est compatible avec S, et on écrit $T \bowtie S$, si $T \leq \overline{S}$.

3.5.1.2 Propositions 1(Vallecillo, Vasconcelos, & Ravara, 2005)

i La relation duale est symétrique.

i Le sous-typage $\leq$ est un préordre[7].

ii $T \leq \overline{S}$ si et seulement si $S \leq \overline{T}$

iii la compatibilité $\bowtie$ est symétrique.

iv Si $T \leq S$ et $U \bowtie T$, alors $U \bowtie S$.

3.5.2 Vérification des composants

Compte tenu des relations de **substituabilité** et de **compatibilité** des sessions, il est facile de vérifier si deux éléments sont substituables (ou compatible): il faut simplement vérifier que chaque session dans le protocole de l'un des composants est substituable (ou compatible avec) certaines sessions dans le protocole de l'autre composant.

---

[7] Un préordre (ou préordre) est une relation binaire *réflexive* et *transitive*.

3.5.3 Substituabilité (Wegner & Zdonik, 1988)

Un type t1 est un sous-type d'un type t2 si toute valeur du type (dynamique) t1 peut être substituée, à l'exécution, à toute expression du type (statique) t2, sans déclencher d'erreur de type. Une instance de sous-type peut toujours être utilisée dans n'importe quel contexte dans lequel une instance de super type était attendue (Wegner & Zdonik, 1988)

Plusieurs auteurs se sont penchés sur le sujet parmi eux (Wegner & Zdonik, 1988) et (Brada, 2001) et bien d'autres avant, ce qui ressort de leurs classifications est qu'il existe au moins deux niveaux de compatibilités (qui elle-même va de soi avec la substitution) dont

- Le niveau de compatibilité contextuelle, est plus souple et ne prend pas en compte la relation de sous-typage
- Le niveau de compatibilité stricte qui équivaut au sous-typage exige que le composant remplaçant soit un sous-type du composant qu'il remplace et qu'il ait des spécifications contra variantes et assure la substituabilité quel que soit le contexte et celui qui nous intéresse le plus.

3.5.4 Définition 2 (Règle de contra-variance)

La redéfinition d'une méthode doit généraliser le type des paramètres et spécialiser le type de retour de la méthode redéfinie, Sachant que les types de session sont du domaine de la théorie des types et des systèmes de type, alors que les contrats sont davantage liés à l'étude des équivalences comportementales, ce qui est intéressant à nos yeux puisque l'utilisation combinée des deux techniques nous permet une complémentarité, une faiblesse dans l'un est componée par l'autre et vice versa, et ainsi on a aussi un mécanisme prenant les deux préoccupations dans la vérification et validation des systèmes qui sont : préoccupation structurelle et le préoccupation comportementale (niveaux 3 et 4 du classement de (Beugnard, Jézéquel, Plouzeau, & Watkins, 1999)

La grammaire suivante est proposée par (Vallecillo, Vasconcelos, & Ravara, 2005) pour une description des protocoles d'interfaces utilisant les types de sessions :

Protocol ::= **Protocol** X {Sessions}

Sessions ::= **session** X = T . . . **session** X = T

T ::= &{m1 : T1 | · · · | mn : Tn} | +{m1 : T1 | · · · | mn : Tn} |

?(˜ T); T | ![˜ T]; T | ?(˜ sort); T | ![˜ sort]; T | X | μX.T | **end**

sort ::= **string** | **float** | **boolean**

Figure 16: Une grammaire de description des protocoles **(Vallecillo, Vasconcelos, & Ravara, 2005)**.

3.5.5 Dualité de Type pour les types de session

La relation de compatibilité fait penser à une relation de dualité, Le dual se calcule simplement en inversant les réceptions et émissions respectivement en émissions et en réceptions, et les branchements en sélections et vis versa.

(Vallecillo, Vasconcelos, & Ravara, 2005) Définissent les duals de types comme suit :

$\overline{?(\tilde{S});T} = ![\tilde{S}] ; \overline{T}$

$\overline{![\tilde{S}];T} = ?(\tilde{S}) ; \overline{T}$

$\overline{\&\{m1 : T1 \mid \cdots \mid mn : Tn\}}$=+{m1 : $\overline{T}$1 | · · · | mn : $\overline{T}$n}

$\overline{+\{m1 : T1 \mid \cdots \mid mn : Tn\}}$=&{m1 : $\overline{T}$1 | · · · | mn : $\overline{T}$n}

$\overline{\boldsymbol{end}}$=**end**

$\overline{X}$=X

$\overline{\mu X.T}$=μX.$\overline{T}$

Figure 17: Les duals des types **(Vallecillo, Vasconcelos, & Ravara, 2005)**

3.5.6 Le sous-typage pour les types de session

(Wegner & Zdonik, 1988) Définissaient quatre types de compatibilité dont la compatibilité comportementale qui est le « Type »**.** Et ajoutaient que la vraie compatibilité comportementale qui satisfait le principe de substitution est plus restrictive que généralement supposée. Trois types de sous typage existent toujours selon (Wegner & Zdonik, 1988) :

1 Sous-type des sous-ensembles : exemple les entiers (0..10) et un sous ensemble du sous type entiers

2 Sous-type isomorphiquement[8]intégré, les entiers sont un sous-type isomorphiquement intégré aux réels.

3 Sous-typage Orienté objet : Etudiant est Sous-type Orienté objet de Personne.

3.6 Le système à base de contrats

Plusieurs auteurs ont utilisé les contrats pour la vérification et la validation des systèmes à construire, en utilisant des grammaires et langages nécessaires pour leurs descriptions (Laneve & Padovani, 2008) (Padovani, Session Types = Intersection Types + Union Types, 2010).

La composition de processus -donc nécessairement des contrats- utilise des grammaires qui les décrivent (Laneve & Padovani, 2008), (Padovani, 2009), (Padovani, 2010) et (Bernardi & Hennessy, Giovanni Tito Bernardi), Les règles d'inférence de la (**Figure 18**) on est une qu'on va utiliser pour translater avec les types de session et conclure une description de système sémantiquement.

Nous nous basons sur la grammaire de contrat de (Bernardi & Hennessy, Giovanni Tito Bernardi), prenant σ un contrat qui est comme suit :

$$\sigma ::= \begin{cases} nil & \text{terminaison} \\ 1 & \text{succès} \\ \alpha.\,\sigma & \text{action} \\ \sigma + \sigma & \text{choix externe} \\ \sigma \oplus \sigma & \text{choix interne} \\ x & \text{variable de contrat} \\ \mu x.\,\sigma & \text{récursivité} \end{cases}$$

Figure 18: Grammaire de contrat **(Bernardi & Hennessy, 2012)**

[8] Un isomorphisme est un morphisme bijectif (soit E un ensemble muni d'une loi de composition * et F un ensemble muni d'une loi de composition x. On dit qu'une application f :E→F est un morphisme lorsque pour tous éléments a et b de E, on a l'identité : f(a*b)=f(a)xf(b).

Les auteurs définissent un système d'inférence pour la sémantique des contrats et leurs translation à travers un ensemble fini d'actions semblable à celui défini par (Vallecillo, Vasconcelos, & Ravara, 2005) pour les types de session, et ce pour le déroulement de système de vérifications comportementales.

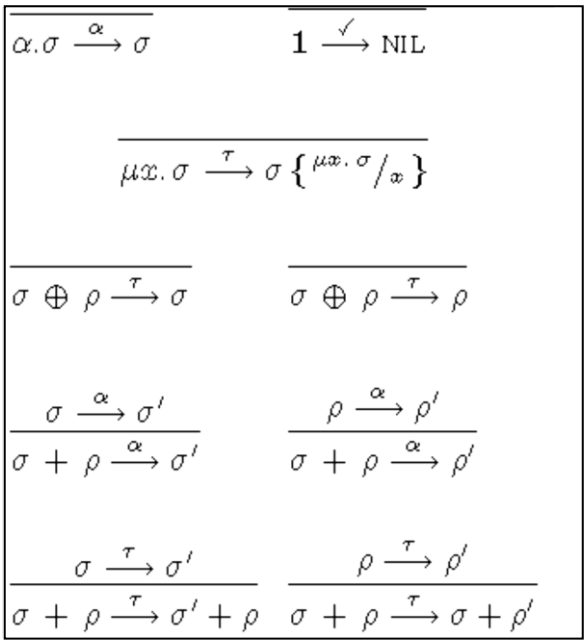


Figure 19: Les règles d'inférence sémantique **(Bernardi & Hennessy, 2012)**

Deux systèmes interagissent par contrats ce qui fait que pour pouvoir vérifier une interaction entre deux systèmes (composants dans notre cas) il faut vérifier la compatibilité de leurs contrats successifs et là on est dans la comparaison de types duaux.

Et pour vérifier la conformité entre le système requis et celui proposé il nous faut vérifier la substitution de système. sachant que les types de session sont à leur tour un moyen de vérification de validation de type de manière à assurer durant une session limitée par une interaction qui peut servir directement (Vallecillo, Vasconcelos, & Ravara, 2003), (Vallecillo, Vasconcelos, & Ravara, 2005) et (Gay S. J., Vasconcelos, Ravara, Gesbert, & Caldeira, 2010) comme moyen de vérification sémantique d'une composition, mais pour plus d'assurance dans les systèmes à bases de composants, intégrer les types de session aux contrats peut avoir un effet plus raffiné et plus cohérant aussi bien en structure qu'en comportement.

3.7 Le système combinant contrats et types de session

La communication entre les processus dans un système réparti est souvent constituée d'un dialogue structuré, selon un protocole qui définit le format des messages échangés, et, au moins pour une communication binaire, le sens de ces messages. Les types de session, ont été introduits comme une approche pour l'analyse statique des participants à ces dialogues. Ils permettent des séquences structurées de messages non uniformes d'être échangées entre les participants.

Un contrat peut décrire le comportement d'un serveur offrant un service spécifique. Dualement un contrat peut décrire le comportement attendu d'un client qui souhaite bénéficier d'un service particulier. La théorie des contrats est l'idée de respect entre ces contrats, formalisée comme une relation asymétrique (Bernardi & Hennessy, 2012), et vu les contraintes sur le comportement, les types de sessions sont beaucoup plus contraignantes que les contrats; (Bernardi & Hennessy, 2012) Montrent que les types de sessions de premier ordre (qui respectent soit la contra variance ou la covariance) peuvent être facilement intégrés dans la théorie des contrats, qui apparaît donc comme un formalisme plus général.

(Bernardi & Hennessy, 2012) Ont proposé les contrats de sessions dans un système Client-serveur, À travers les deux mécanismes et leurs systèmes de preuves et de déroulement des règles (grammaires, règles d'inférence). Qui ont donné les contrats de sessions :

$$\rho, \sigma = 1 | \sum_{i \in M} ?mi.\sigma i | \bigoplus_{i \in M} !mi.\ \sigma_i \mid ?t.\sigma \mid !\ t.\sigma \mid x \mid \mu x.\sigma$$

Figure 20: Grammaire des contrats de session **(Bernardi & Hennessy, 2012)**

Voici les équivalences structurelles des contrats :

$\sigma \oplus \rho = \rho \oplus \sigma$ $\qquad$ $\sigma \oplus (\sigma' \oplus \sigma'') = (\sigma \oplus \sigma') \oplus \sigma''$

$\sigma + \rho = \rho + \sigma$ $\qquad$ $\sigma + (\sigma' + \sigma'') = (\sigma + \sigma') + \sigma''$

Une translation à travers un mappage des deux systèmes contrat et types de session) qui donne les contrats de sessions (CS).

Notez que CS est un sous-ensemble du langage plus commun des contrats C, mais

- Choix externes sont limitées à des entrées sur les labels.

- Choix internes sont limités à des sorties sur des labels.

(Bernardi & Hennessy, Giovanni Tito Bernardi) (Bernardi & Hennessy, 2012) Ont proposé des définitions pour permettre de vérifier et traiter les contrats de sessions dont on va se baser dans la partie de démonstration de la substituions et la compatibilité des services de composants à assembler dans notre approche à base de contrats et des types de session.

Définition 3 [Préordre Restreint de serveur]

Pour $\sigma_1, \sigma_2 \in$ CS (contrat de session) (voir Le système combinant contrats et types de session) sachant $\sigma_1 \sqsubseteq_{srv}^{SC} \sigma_2$ chaque fois que le processus P du client répondant au contrat de serveur σ_1 répond aussi à σ_2. (Bernardi & Hennessy, Giovanni Tito Bernardi).

Définition 4 [Préordre Restreint de client]

Pour $\rho_1, \rho_2 \in$ CS (contrat de session) sachant $\rho_1 \sqsubseteq_{CLT}^{SC} \rho_2$ chaque fois que le contrat ρ_1 du serveur répondant à σ, le contrat ρ_2 l'est aussi. (Bernardi & Hennessy, Giovanni Tito Bernardi).

Définition 5 [Préordre des contrats de Session]

Pour $\sigma_1, \sigma_2 \in$ CS (contrat de session) $\sigma_1 \sqsubseteq^{SC} \sigma_2$ chaque fois que $\sigma_1 \sqsubseteq_{srv}^{SC} \sigma_2$ et $\sigma_1 \sqsubseteq_{CLT}^{SC} \sigma_2$. (Bernardi & Hennessy, 2012).

Ce qui signifie que les conditions des contrats en (Définition 3 [Préordre Restreint de serveur] assurant une substitution sure du serveur et de l'autre coté une substitution sure pour les clients (Définition 4 [Préordre Restreint de client].

3.8 Mappage entre contrats et types de session

L'interprétation des types de sessions comme des contrats est exprimée en fonction du langage des types des sessions **Figure 16** et celui des contrats de session **Figure 20**, et la fonction M définie par (Bernardi & Hennessy, Giovanni Tito Bernardi) et améliorée par (Bernardi & Hennessy, 2012) qu'on adapte pour notre approche comme suit :

$$M(T)= \begin{cases} 1 & \text{si } T=End \\ !sort.M(T) & \text{si } T =![sort]; T \\ ?sort.M(T) & \text{si } T =?[sort]; T \\ \sum_{i\in(1..n)}? mi.M(Ti) & \text{si } T=\&\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \\ \bigoplus_{i\in(1..n)}!mi.M(Ti) & \text{si } T =+\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \\ \mu X.M(T') & \text{si } T= \mu X.T' \\ x & \text{si } T=X \end{cases}$$

Figure 21: Fonction d'interprétation M **(Bernardi & Hennessy, 2012)**

Théorème 1 [Full abstraction]

Pour tous les types de sessions, T1 $\leq$T2 si et seulement si M (T1) $\sqsubseteq^{SC}$M (T2) (Bernardi & Hennessy, 2012).

Ce théorème permet d'utiliser le sous-typage des types de session($\leq$) pour la autoriser la substitution et la compatibilité des types de session avec la prise en charge des contrats à travers les contrats de session permettant ainsi une double vérification sémantique comportementale (contrats) et de type (types de session).

3.9 Synthèse

- Aspects formels : Des propriétés peuvent être établies par raisonnement formel, ce qui n'est pas le cas des autres formes de spécification.
- Les outils de preuve calculent et vérifient automatiquement et uniquement ce qui a été décrit.
- Précision : Un langage commun lève les ambiguïtés et facilite la communication entre les acteurs (humains ou numériques).
- Abstraction : Seules les caractéristiques essentielles sont retenues, Les composants logiciels sont plus ables et plus généraux donc plus réutilisables.

Un problème lié à l'approche de développement orientée composant est de savoir comment définir la visibilité extérieure d'un composant, c'est comment spécifier son interface. D'autre part, les utilisateurs de composants doivent comprendre un composant au moyen de ce genre de description. Ainsi, l'absence de normes pour décrire efficacement ce qu'est un composant et sur la façon d'interagir avec lui est un obstacle pour cette approche.

Habituellement, la spécification d'interface est utilisée pour décrire un composant. Des approches comme description de l'interface langues (IDL) décrivent les services fournis par un composant. Ceci est approprié pour des composants fabriqués à fournir des services. Souvent, les signatures de fonction sont suffisantes pour établir comment un client accède à la fonctionnalité. Une classe à partir d'une bibliothèque qui implémente une liste est un exemple d'un composant qui peut être décrit de manière adéquate par le biais de ses services fournis. Donc une description appropriée des interfaces est primordiale si l'on veut vérifier que l'assemblage est correct.

3.10 Conclusion

La spécification formelle débouche d'un besoin accru en matière de conception et développements logiciel et surtout ce qui concerne la vérification en amont et en aval des propriétés du produit final, et dans notre choix du paradigme composant l'accent est donné sur l'assemblage assisté par les agents et la vérification par ces derniers du bon déroulement du processus, Ceci implique l'usage d'un langage clair en plus d'être compréhensible par la machine qui jouera en partie le rôle de l'humain (concepteur et développeur) par le biais des agents. Et à travers l'introduction des contrats, le processus d'assemblage et de vérification aura une assurance d'une vérification comportementale formelle.

Les types de session ont aussi prouvé leur efficacité dans un certain créneau de vérification des systèmes que plusieurs auteurs ont ressèment améliorés et enrichis pour répondre aux exigences de plus en plus accrues et pressantes pour les concepteurs et développeurs de systèmes, Ainsi en combinant ces deux formalismes qui sans doute ont chacun ses limites et ses atouts, on a essayé de les rendre complémentaires vu que les contrats sont plus comportementaux et moins strictes et les types de session d'un niveau plus bas (types) mais plus exigeants.

Notre but est de définir un formalisme adapté à la spécification du système. Nous voulons pouvoir décrire le comportement d'un ensemble d'éléments qui interagissent, et pas seulement le comportement d'un composant isolé.

Combiner ces deux formalismes qui ont prouvé leur efficacité mais qui ont chacun ses faiblesses peut nos porter une combinaison réussie. L'assemblage de composants et la vérification de son validité fonctionnelle, cela nous à poussé à élargir le mécanisme qui était restreint à l'architecture client-serveur, qui par la nature des agents exige une liberté de

décisionnelle donc un assemblage par paires de composants est possible, une méthode de mappage entre contrats et types des sessions est utilisée pour obtenir un contrat de session, sémantique. Des définitions et règles d'inférences pour le déroulement et le passage par preuve d'une étape à une autre jusqu'à confirmer ou infirmer une compatibilité ou substituabilité de composants.

Chapitre IV : Agents et systèmes multi agents

4.1 Introduction aux agents

Dans des applications complexes, l'expertise que l'on cherche, c'est-à-dire le savoir-faire, les compétences et les connaissances diverses, est détenue par des individus différents qui, au sein d'un groupe, communiquent, échangent leurs connaissances et collaborent à la réalisation d'une tâche commune. Dans ce cas, la connaissance du groupe n'est pas égale à la somme des connaissances des individus: chaque savoir-faire est lié à un point de vue particulier et ces différentes visions, parfois contradictoires, ne sont pas nécessairement cohérentes entre elles.

La réalisation d'une tâche commune nécessitera alors des discussions, des mises au point, voire même des négociations pour résoudre les conflits éventuels. Une résolution efficace de ces problèmes doit s'intéresser à mettre la solution dans des entités autonomes reflétant la réalité distribué des systèmes naturelles.

La résolution des problèmes par systèmes multi agents consiste à pouvoir étudier, concevoir et réaliser des univers ou des organisations d'agents artificiels capables d'agir, de collaborer à des tâches communes, de communiquer, de s'adapter, de se reproduire, de se représenter l'environnement dans lequel ils évoluent et de planifier leurs actions, pour répondre soit à des objectifs définis extrinsèquement (par un programmeur humain par exemple), soit intrinsèquement à partir d'un objectif général de survie. On dira aussi que la solution procède par la construction de systèmes multi-agents, c'est-à-dire par la réalisation de modèles composés d'entités artificielles qui communiquent entre elles et agissent dans un environnement.

4.2 Agents et système multi agents

Que ce qu'un agent ? La programmation orientée agent est donc aujourd'hui beaucoup plus une manière de penser un système informatique qu'une technique d'implémentation particulière. Ainsi, la plupart des définitions qui ont été proposées pour le terme agent s'attachent à décrire des principes généraux lies à cette approche.

Parmi ces principes, certains semblent aujourd'hui faire l'objet d'un consensus au sein de la communauté multi-agents. Nous utiliserons donc la définition exposée dans

(Wooldridge & Jennings, 1995) et (Ferber, 1995) qui résument ces principes. Par le terme agent, un système est une entité physique ou virtuelle qui possède les propriétés suivantes :

1. Qui est capable d'agir dans un environnement,
2. Qui peut communiquer directement avec d'autres agents,
3. Qui est mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),
4. Qui possède des ressources propres,
5. Qui est capable de percevoir (mais de manière limitée) son environnement,
6. Qui ne dispose que d'une représentation partielle de cet environnement (et éventuellement aucune),
7. Qui possède des compétences et offre des services,
8. Qui peut éventuellement se reproduire,
9. Dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit".

Un agent est un processus cyclique composé de trois phases : perception délibération action.

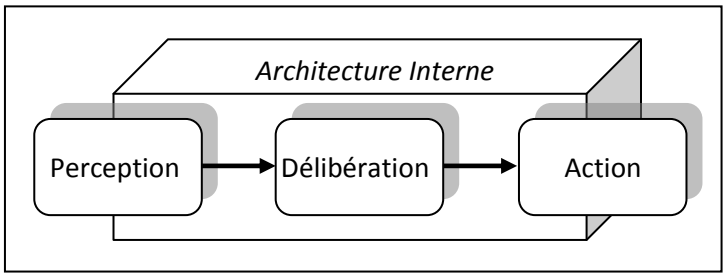


Figure 22: L'agent, un processus à trois phases : perception, délibération puis action **(Michel, 2004)**

4.2.1 Un système multi agents

"On appelle système multi-agent (ou SMA), un système composé des éléments suivants:

1. Un environnement E, c'est-à-dire un espace disposant généralement d'une métrique.
2. Un ensemble d'objets O. Ces objets sont situés, c'est-à-dire que, pour tout objet, il est possible, à un moment donné, d'associer une position dans E. Ces objets sont passifs, c'est-à- dire qu'ils peuvent être perçus, créés, détruits et modifiés par les agents.

3. Un ensemble A d'agents, qui sont des objets particuliers (A inclut O), lesquels représentent les entités actives du système.
4. Un ensemble de relations R qui unissent des objets (et donc des agents) entre eux.
5. Un ensemble d'opérations Op permettant aux agents de A de percevoir, produire, consommer, transformer et manipuler des objets de O.
6. Des opérateurs chargés de représenter l'application de ces opérations et la réaction du monde à cette tentative de modification, que l'on appellera les lois de l'univers. "

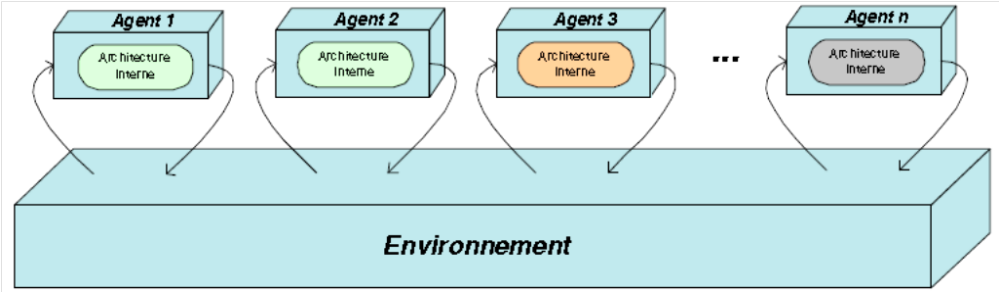


Figure 23: Représentation schématique d'un système multi-agents.

4.3 L'agent et son environnement

L'environnement est un aspect très important dans l'étude des systèmes multi agents, puisque dans cet environnement ils vivent, ils le perçoivent à travers ses capteurs et agit sur lui à travers ses effecteurs.

Bien que les environnements sont différents, et il y a beaucoup de critères qui entre dans leurs propriétés, il est difficile d'avoir une liste exhaustive de leurs caractéristiques. Une classification des environnements est celle de (Russell & Norvig, 2003).

Accessible versus inaccessible. Un environnement est accessible si l'agent peut avoir une vue complète, exacte et actualisée sur l'état de l'environnement. Et la plupart des environnements réels ne sont pas accessible dans ce sens.

Déterministe versus non déterministe. Un environnement est déterministe si toute action sur cet environnement a un seule effet, c à d qu'il n y a pas une ambiguïté sur l'état qui résulte après l'achèvement d'une action.

Statique versus dynamique. Un environnement est statique s'il est supposé inchangé sauf par les actions de l'agent. Par contre un environnement dynamique peut avoir des changements au-delà de la portée de l'agent.

Discret versus continu. Un environnement discret est un environnement où les perceptions de l'agent et ses actions sont au nombre limité à l'inverse d'un environnement continu.

4.4 Architectures d'agents

Un agent se caractérise essentiellement par la manière dont il est conçu et par ses actions, en d'autres termes par son architecture et par son comportement. L'architecture correspond à un point de vue de concepteur, qui peut se résumer par la façon d'assembler les différentes parties d'un agent de manière qu'il accomplisse les actions que l'on attend de lui?

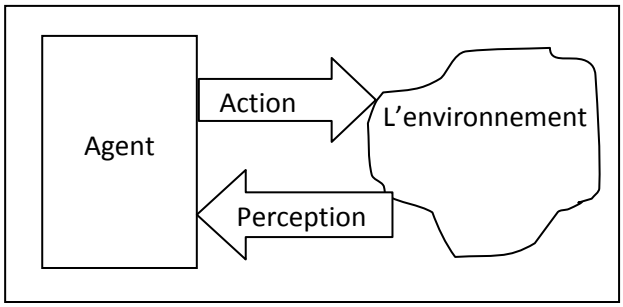


Figure 24: L'agent et son environnement

Suivant la façon dont un agent prend les décisions de ses réactions on peut distinguer plusieurs types d'agents.

L'architecture d'un agent caractérise ainsi sa structure interne, c'st-`à-dire le principe d'organisation qui sous-tend l'agencement de ses différents composants.

- Agent basé sur la logique
- Agent réactif
- Agent délibératif
- Agent BDI (Belief-Desire-Intention)
- Architecture hybride ou architecture couches

4.5 Les agents mobiles

Le schéma de mobilité des agents mobiles dérive de deux domaines différents, i.e. les agents venant de l'intelligence artificielle avec les systèmes multi-agents et des systèmes

distribués avec la migration de processus (Cubat dit Cros, Agents Mobiles Cooperants pour les Environnements Dynamiques, 2005).

Les agents mobiles ont été introduits initialement en 1994 avec l'environnement **Telescript** qui permettait à des processus de choisir eux-mêmes de se déplacer sur les sites d'un réseau afin de travailler localement sur les ressources. Dans ce cas, la migration était forte et proactive. L'intérêt suscité par cette méthode fut confirmé avec le succès de Java et de son utilisation dans la programmation d'applications basées sur le Web.

Il existe différents types de migration de code (Cubat dit Cros, Agents Mobiles Cooperants pour les Environnements Dynamiques, 2005) qui peuvent être classifiés selon deux grandes caractéristiques : la première est la prise de décision de la migration (réactive/proactive) et la seconde est la capacité de déplacement du contexte d'exécution (forte/faible).

Un agent réalise une fonctionnalité précise et relativement simple. Une application réalisant une tâche plus complexe sera construite grâce à plusieurs agents, comme dans le cadre de la conception orientée objet où une application est un ensemble d'objets ou composants et de relations.

4.5.1 Définition

Un agent statique est un agent qui s'exécute seulement dans les systèmes où il commence son exécution. Il utilise un mécanisme de communication tel que RPC. Par contre, un agent mobile n'est pas lié au système dans lequel il débute son exécution.

Un agent mobile (Fuggetta, Picco, & Vigna, 1998) est un agent logiciel qui peut se déplacer d'un site à un autre en cours d'exécution pour se rapprocher de données ou de ressources.

Il se déplace avec son code et ses données propres, mais aussi avec son état d'exécution. Selon (Xu & Wims, 2000) Les agents mobiles ont pour trait la définition de la capacité de se déplacer de machine en machine. Lorsqu'il se déplace, l'agent emballe son code, et son état de fonctionnement et se déplace vers un nouveau site. Une fois arrivé sur le nouveau site, l'agent continue à exécuter son code là où il s'était arrêté.

Donc un agent est un processus, incluant du code et des données, pouvant se déplacer entre des machines pour réaliser une tâche et communique potentiellement avec d'autres agent.

4.5.2 Motivation

L'utilisation d'agents conduit à la décentralisation de la connaissance et du contrôle. Développé aux frontières du génie logiciel et des systèmes répartis, le concept d'agent mobile est à priori destiné à la mise en œuvre d'applications dont les performances varient en fonction de la disponibilité et de la qualité des services et des ressources, ainsi que du volume des données déplacées : par exemple, un agent mobile peut adapter un service aux capacités du poste client et à la bande passante disponible.

L'agent mobile est capable de se déplacer d'un hôte à un autre hôte dans le réseau. Il peut transporter son état et son code d'un environnement vers un autre dans le réseau où il poursuit son exécution. L'idée de la mobilité du code et des agents mobiles trouve plusieurs motivations dans les applications modernes de l'informatique. Nous décrivons ci-dessous certains de ces motivations (Bahri, 2009) :

• **Plus d'efficace:** au lieu d'avoir des interactions à distance avec un serveur (en générale coûteuse) c'est-à-dire appel de procédure à distance (RPC : Remote Procedure Call), le mieux est de déplacer le code vers le serveur et avoir des interactions locales moins coûteuses.

• **Robustesse et tolérance aux fautes:** lorsqu'un système/machine hôte est en difficulté (rupture de communication), les agents mobiles visiteurs prévenus ont la possibilité de se dispatcher ailleurs dans le réseau. En plus, l'exécution asynchrone et autonome des agents mobiles rend l'application tolérante en vers les ruptures possible entre hôte source et hôte destination.

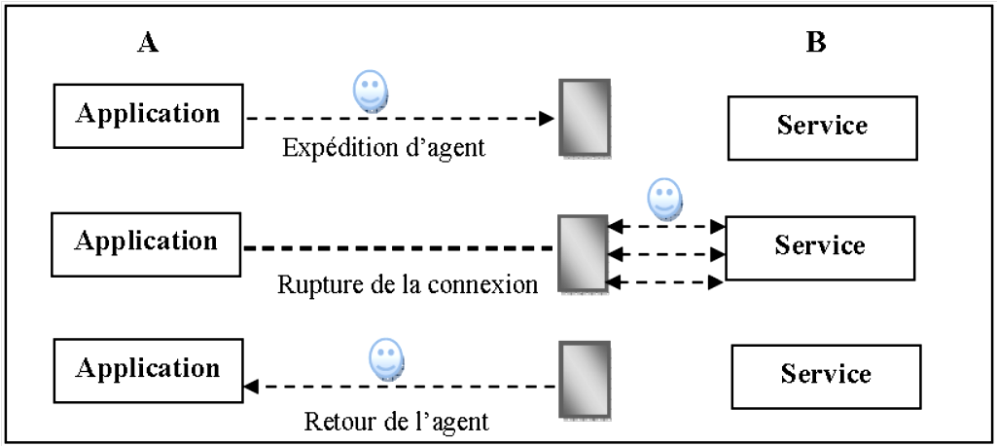


Figure 25: Exécution asynchrone et autonome

• **Réduction de la charge dans le réseau :** les agents mobiles réduisent le trafic du réseau, en déplaçant le traitement vers les données brutes plutôt que de déplacer les données vers le traitement.

• **Adaptation dynamique à l'environnement :** les agents mobiles sont autonomes, ils peuvent « découvrir » le réseau, c'est-à-dire qu'ils pourront visiter des hôtes qui leurs étaient inconnus. À chaque nouvel hôte parcouru, ils pourront prendre des décisions basées sur leurs propres connaissances ainsi que sur les connaissances acquises des hôtes visités.

• **Une solution stratégique :** au lieu d'avoir trop de services sur un certain serveur dès le début, on peut adapter ces services selon les besoins spécifiques des clients.

• **Plates-formes hétérogènes :** l'informatique dans les réseaux à grand échelle, est fondamentalement hétérogènes (logiciel, matériel). Les agents procurent une configuration unique, dans laquelle les applications réparties peuvent être implémentées facilement et de manière efficiente.

• **Equilibrage de charge :** la mobilité d'agent lui permet de déplacer la charge de calcul d'un site à un autre, afin d'essayer d'équilibrer l'utilisation des ressources dédiées à l'application sur les différents sites participant.

4.5.3 Migration d'un agent

Les migrations d'agents mobiles peuvent s'effectuer selon deux modes :

1. Migration forte

La migration forte, où la totalité de l'agent (c'est-à-dire code, données et unité d'exécution) migre vers le nouveau site. Pour cette migration réelle, l'agent est suspendu ou capturé avant d'être transféré. Une fois arrivé sur le site distant, il redémarre son exécution au point de contrôle précédent, en conservant l'état du processus.

2. Migration faible

La migration faible ne fait que transférer avec l'agent son code et ses données. Sur le site de destination, l'agent redémarre son exécution depuis le début en appelant la méthode qui représente le point d'entrée de l'exécution de l'agent, et le contexte d'exécution de l'agent est réinitialisé.

4.6 Les normes

L'émergence des nombreuses plates-formes expérimentales a rendu nécessaire la proposition d'une harmonisation grâce à la standardisation des différents concepts communs pouvant être identifiés. Cette normalisation devrait permettre à terme de rendre compatibles les différents systèmes. On peut trouver deux normes principales (Cubat dit Cros, Agents Mobiles Cooperants pour les Environnements Dynamiques, 2005):

• FIPA (Fondation for intelligent physical Agents)

En revanche, la communauté d'origine de FIPA étant celle des systèmes multi-agents, plus proche de l'intelligence artificielle, FIPA s'intéresse plus particulièrement à l'interopérabilité des agents intelligents (les efforts sont placés au niveau du langage, des protocoles et des infrastructures de communication), elle va se situer à un niveau plus élevé c'est à dire le niveau applicatif, en décrivant les éléments nécessaires à la réalisation d'une application et principalement en détaillant la communication entre les agents.

• MASIF (Mobile Agent System Interoperability Facility)

La norme MASIF a été spécifiée par l'Object Management Group (OMG) qui se préoccupe généralement de l'hétérogénéité entre les systèmes, l'OMG s'intéresse à l'interopérabilité des agents mobiles à travers sa spécification appelée MAF.

La raison de créer l'accord de liaison OMG-FIPA est d'encourager des normes de technologie d'agent à se développer compatible avec des normes de technologie d'objet et à la nouvelle coordination entre OMG et le travail lié du FIPA vers des normes de technologie d'agent (Cubat dit Cros, Agents Mobiles Cooperants pour les Environnements Dynamiques, 2005).

4.7 Implémentations existantes

À l'heure actuelle, il existe beaucoup d'implémentation, incompatibles entre elles. Certaines de ces plates-formes existantes à nos jours sont présentées ci-dessous (Cubat dit Cros, Agents Mobiles Cooperants pour les Environnements Dynamiques, 2005) :

LIME : le premier système que nous étudions est l'intergiciel basé sur Java nommé LIME (Linda In Mobile Environnement) qui propose une couche de coordination pour les agents en réutilisant le modèle Linda. Ce n'est pas à proprement parler une plate-forme car elle ne prend pas en charge les différents éléments décrits par les normes. Elle se focalise plus sur la coordination qui peut être exploitée pour construire des applications réparties.

Aglets : utilise la notion de référence distante (AgletProxy), le passage de messages (multicast, broadcast), développé par IBM. Il offre en outre une certaine notion de sécurité en limitant les ressources allouées aux aglets.

JADE : (java Agent Development Framework), est un intergiciel construit sur java afin de permettre une programmation multi-agents simplifiée en prenant en compte la norme FIPA.

TACOMA : le projet TACOMA (Troms And COrnell Moving Agents), de Norway & Cornell University, a été développé dans le but d'offrir différentes abstractions de haut niveau pour les agents.

PLANGENT est une plate-forme basée sur le langage Java s'intéressant à l'adaptation des agents durant leurs déplacements au sein des environnements dynamiques.

4.8 Avantages et inconvénients des agents mobiles

Les agents mobiles ont rapidement suscité un intérêt tout particulier dans les domaines de recherche portant sur les applications réparties. Très rapidement, cette nouvelle méthode de

programmation a été évaluée afin de voir ce qu'elle pouvait apporter comme caractéristiques propres et ce qu'elle permettait d'améliorer par rapport aux méthodes de programmation plus classiques. Différents apports tout autant que les difficultés sont soulevées par les agents mobiles.

La performance : Les premières améliorations apportées par les agents mobiles portent sur le gain de performance due à une meilleure utilisation des ressources physiques mises à disposition.

La conception : Du point de vue de la conception, les agents mobiles représentent à la fois une méthode permettant de mieux caractériser certaines applications mais ils apportent aussi une complexité accrue par rapport au classique client/serveur bien mieux maîtrisé.

Le critère de performance des agents sera très utile dès que les applications vont comporter des communications intensives. Grâce à leurs interactions locales, les agents permettent de ne pas subir les ralentissements dus aux bandes passantes limitées et aux temps de latence des réseaux, surtout sur Internet. D'importantes difficultés de développement existent. Elles sont dues principalement au manque de standardisation des intergiciels qui ne permettent pas d'obtenir, par exemple, un langage homogène partagé par tous les agents. On peut ajouter à cela, les difficultés de mise au point dues aux déplacements de l'unité d'exécution qui sont difficiles à suivre et aux problèmes de test qui nécessitent d'importants efforts de coordination.

4.9 Composants logiciels et systèmes multi-agents

De nombreux modèles d'exécution sont proposés par les différents middlewares.

Le plus courant est le modèle client-serveur (appels de procédures à distance, objets répartis). Cette approche repose sur l'hypothèse d'une qualité de service relativement bonne qui permet à un objet distant de rester accessibles pendant une durée importante.

Cette hypothèse est réaliste dans le cadre d'un réseau local ou d'un réseau interne à une seule entité administrative. Dans le cadre de réseaux à grande distance faisant intervenir de nombreuses entités administratives (les particuliers par exemple), le coût du maintien des propriétés nécessaires à ce modèle est trop élevé. Actuellement, l'approche « pair à pair » vise à compenser les difficultés liées à la centralisation.

Dans ce travail, nous explorons l'intérêt d'associer les technologies d'agent (autonomie, mobilité, adaptabilité) et de composant logiciel pour la construction d'applications par assemblage de composants assisté par les agents, des applications adaptables aux changements et à la dynamique des différents intervenants (utilisateurs, développeurs, environnement…).

4.9.1 Dualité composants - agents

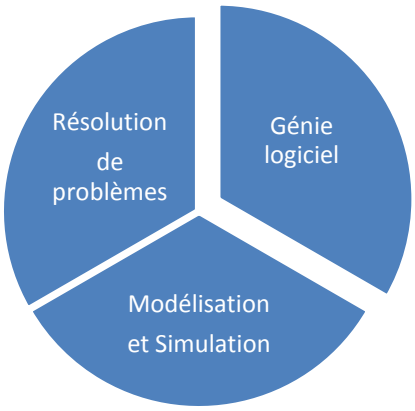


Figure 26: Classes de problèmes traités par les Agents systèmes

Les agents peuvent contribuer au génie logiciel et spécialement l'approche composant et des composants logiciels « adaptables ». En résumé :

- Les interactions non prévues deviennent la norme et non plus l'exception
- Possibilité d'objectifs en compétition, avec moyens de réconciliation
- A l'inverse (et complémentaire) de la programmation « sécuritaire » classique
- Le couplage entre composants est abordé au niveau des connaissances et non plus au niveau des types de données (ce qui est sûr mais rigide)
- Les composants logiciels peuvent en partie eux-mêmes organiser leur assemblage (appareillement, appel d'offre, négociation…)
- Vers un plus grand découplage, ou plutôt un couplage plus sémantique et plus flexible : objets -> composants -> agents (et ensuite ?!)
- A rapprocher du "Ever late binding" (C -> C++ -> Java -> ...)

4.9.2 Des composants plus autonomes et adaptatifs

L'approche composant a été toujours à la recherche de mécanismes souple et performant pour ce qui présente la phase d'assemblage et de vérification, plusieurs recherches ont tenté de résoudre ce problème, nous nous sommes penché sur les agents et l'opportunité qu'ils représentent pour ce type de problème,

1. ***Assistance à l'assemblage***

Le choix des composants logiciels pour constituer une architecture est à priori manuel. Le concepteur doit donc identifier les composants souhaités, souvent à partir de bibliothèques, internes ou externes (« sur étagères »), et les assembler en fonction des objectifs souhaités. L'approche peut être plutôt ascendante, par construction sont d'un niveau plus sémantique, mais restent plus focalisés sur les hypothèses de fonctionnement correct que sur une véritable description des services offerts ou requis par un composant.

Une direction est en conséquence d'offrir des services d'assistance à l'assemblage (identification, mise en correspondance, assemblage effectif) à partir d'informations sur « ce que font » les composants ou/et à partir d'objectifs plus globaux de l'application.

Un exemple d'une telle approche est le modèle de mise en correspondance (matchmaking) d'agents LARKS (Sycara, Widoff, Klusch, & Lu, 2002). Il est relativement élaboré et est fondé sur différents niveaux de descriptions (signatures-types, comportementaux-contraintes, ontologiques-concepts. . .), différents types de mesures de similarité, et différents types de politiques de mise en correspondance (exact, inclusion, relâché). LARKS a été conçu pour la recherche et la mise en correspondance d'agents logiciels sur Internet, mais l'approche suivie est en fait plus générale et peut aussi s'appliquer à des composants logiciels ou encore à des services.

Le projet et modèle de composant COGents (Bertrand Braunschweig, 2002) est un autre exemple d'assistance à l'assemblage de composants logiciels. Son domaine d'application est la conception assistée et la simulation de processus pétrochimiques. Le projet illustre bien la progression d'une approche au départ objet, vers une approche composant, puis vers une approche agent.

2. Auto-reconfiguration de l'architecture

La conception et la construction d'une architecture logicielle reste encore en général réalisée statiquement à la conception. Or les nouvelles applications dynamiques et ouvertes, telles que l'informatique nomade ou ubiquitaire[9], engendrent des besoins d'adaptation et de reconfiguration dynamiques, au niveau de l'architecture, comme au niveau d'un composant. Ceci amène une évolution actuelle de mécanismes d'adaptation, au départ essentiellement réactifs à partir d'évènements (Briot & Seghrouchni, 2009), vers des mécanismes de plus haut niveau, à partir de contraintes et de politiques, voire de buts et de plans. Une illustration d'un tel objectif est le modèle abstrait MaDcAr (Grondin, Bouraqadi, & Vercouter, 2006) d'assemblage et réassemblage dynamique et automatique d'applications à base de composants. À partir de configurations existantes de composants, de contrats de compatibilité et de politiques d'assemblage, MaDcAr gère la reconfiguration d'assemblages.

4.9.3 Apports des agents mobiles

L'utilisation d'agents conduit à la décentralisation de la connaissance et du contrôle. Développé aux frontières du génie logiciel et des systèmes répartis, le concept d'agent mobile est à priori destiné à la mise en œuvre d'applications dont les performances varient en fonction de la disponibilité et de la qualité des services et des ressources, ainsi que du volume des données déplacées : par exemple, un agent mobile peut adapter un service aux capacités du poste client et à la bande passante disponible.

Dans (Harrison, Chess, & Kershenbaum, 1995), les auteurs précisent le champ d'application : sources d'information multiples réparties, volumes importants, prise en compte des spécificités du client, interactions avec le client et la source. À priori, les avantages des agents mobiles sont nombreux :

- La mobilité d'agent permet à un client d'interagir localement avec un serveur, et donc de réduire le trafic sur le réseau qui ne transporte plus que les données utiles (éventuellement prétraitées). En outre, cela permet des transactions plus robustes que les transactions distantes.
- L'exécution d'agents spécialisés offre davantage de souplesse que l'exécution d'une procédure standard sur les sites serveurs.

[9] Omniprésence

- L'asynchronisme, l'autonomie et la réactivité des agents leur permet de réaliser une tâche tout en étant déconnecté du client, ce qui est particulièrement utile dans le cas de supports physiquement mobiles (clients ou serveurs d'information).

Le principal apport des agents mobiles se situe sur un plan du génie logiciel : les agents mobiles sont des unités de structuration des applications ; ils unifient en un modèle unique différent paradigme d'interaction entre entités réparties (client-serveur, communication par messages, code mobile). Ceci dit, que peuvent amener les agents aux composants ?

- Abstraction
- Autonomie
- Adaptabilité
- Interopérabilité
  - ✓ plus sémantique et plus flexible
  - ✓ (fondée sur la connaissance plutôt que sur des types)
- Coordination
  - ✓ Plus évoluée que les architectures logicielles classiques (statiques, typage...)
  - ✓ Notons cependant que la communauté des architectures logicielles et de la configuration/déploiement considère progressivement plus de dynamicité.

Outre les avantages de l'approche agent sur le plan de l'architecture du système de réutilisation (modularité et ouverture), son utilisation dans la formulation et le guidage de la mise en œuvre du processus de réutilisation présente plusieurs intérêts. Elle permet de:

1. Distribuer le processus de réutilisation en délégant à chaque agent composant des fonctions de recherche et de sélection, d'adaptation et d'assemblage.
2. Faciliter l'introduction et la suppression de composants et enfin.
3. Voir le processus d'assemblage comme un processus décisionnel qui se construit dynamiquement en fonction des composants existants.

4.10 Conclusion

Au lieu d'adapter l'esprit humain à la pensée de la machine en exprimant nos idées d'une façon séquentiel, les visions actuelles et future dans les systèmes informatiques vont changer de méthode en rendant les machines se comporter selon ce que nous pensons. Une

vision future de J Ferber est : « En l'an 2045, tout ce qui constitue notre culture informatique actuelle sera dépassé. L'intelligence artificielle n'existera peut-être plus en tant que discipline à part, si ce n'est comme étude de la "psycho-socio-biologie des artefacts". Elle sera totalement fusionnée et amalgamée avec tous les secteurs de l'informatique et en collaboration avec les sciences physiques, biologiques, sociales et psychologiques. La structure des ordinateurs aura changé. La notion même de processeur central aura totalement disparu. Chaque ordinateur sera constitué de plusieurs milliers de petits processeurs travaillant ensemble, constituant ainsi le "milieu" dans lequel les petites entités des programmes évolueront».

La résolution de problèmes ne va plus donc se reposer sur des visions classiques globales et centralisées, mais plutôt sur une vision locale, basée sur interactions entre entités autonomes, résolvant le problème global par émergence de fonctionnalités simples. Cette conversion va changer toute notre façon de concevoir et même de penser pour la résolution des problèmes actuels.

Cette vision du futur suppose selon (Ferber, 1995) que les logiciels puissent coopérer harmonieusement à l'instar des humains, qui apprennent par "osmose", c'est-â-dire par confrontation avec la réalité, en imitant, en essayant et en modifiant des comportements "pour voir ce que ça fait", les logiciels de l'avenir devront être en contact avec nous, en regardant et en apprenant à partir des flux d'informations qui circulent normalement entre les humains.

Chapitre V : Proposition d'une approche basée contrat, type de session et agents

5.1 Introduction

À travers l'hybridation de deux mécanismes de vérification comportementale, les contrats et les types de session qui donne les contrats de session, vus précédemment On souhaite augmenter l'efficacité des assemblages pour aboutir à un meilleur rendement qui à l'aide d'agents mobiles peut apporter plus de souplesse et d'adaptabilité sans pour autant ignorer l'essentiel qui est la vérification du (des) système(s) offert(s) au client.

Nous avons aussi proposé un mécanisme de classement et de sélection des assemblages valides possibles, qui est basé sur la distance entre un assemblage donné et la spécification du client. Et ce à travers un algorithme de classement des différents assemblages selon un principe simple mais qui peut être développé et amélioré pour aboutir par un mécanisme plus complexe et de ce fait plus efficace.

En exploitant le (Théorème 1 [Full abstraction]) de (Bernardi & Hennessy, 2012) avec la définition (3.5.1.1 Définition 1 (Vallecillo, Vasconcelos, & Ravara, 2005)) de substitution on propose :

Définition 6 (compatibilité et substituabilité)

Pour tous types T, S :

- T peut substituer S, Si et seulement si: M (T) $\sqsubseteq^{SC}$M(S) ($\sqsubseteq^{SC}$: Pré ordre de contrat de session).
- T est compatible avec S, si et seulement si: M(T)≤M($\overline{S}$) (≤ : Sous type)

Nous avons adapté le mécanisme de substitution de contrats de session (CS) (Définition 5 [Préordre des contrats de Session]) entre client et serveur pour l'appliquer à la substitution de composants deux à deux de telle sorte qu'ils répondent aux conditions des définitions précitées, et pour cela nous vérifions la compatibilité pour chaque couple de composants à la fois comme serveur et comme client.

Et en utilisant la fonction de mappage (transformation) M (voir **Figure 21: Fonction d'interprétation**), on a utilisé les méthodes (m) comme représentation du service basique qu'un composant offre ou requiert, en s'inspirant de la grammaire de description (**Figure 16**),

qui permet une combinaison des contrats avec les types de session donnant ainsi les contrats de session.

Par l'intégration d'une description de haut niveau représentée par les contrats de session dans les différents composants ainsi qu'aux agents on obtient un système capable de coopérer et surtout d'être vérifié sémantiquement de manière automatique, de cette manière le la vérification sémantique des applications est validée ou invalidée avant même la création des liens et de ce fait on gagne en temps et en coût.

La vérification de l'interopérabilité de deux systèmes (ou sous systèmes) passe par la capacité à les distinguer entre eux et les comparer, pour pouvoir le faire nous proposons la définition du sous typage partiel qui est un pré ordre, on note ≤& pour les branchements et ≤+ pour les sélections des composants (branchements et sélections vus plus haut) :

Définition 7 (Sous-typage partiel ou pré-ordre ≤&, ≤+)

j et k des entiers, ∀ j, k>=0, m=Méthode, T= Type.

1. $\leq_{\&}$: $\&\{m_1 : T_1 \mid \cdots \mid m_n : T_n\} \leq \&\{m_1 : T_1 \mid \cdots \mid m_{n+j} : T_{n+j}\}$ par extension (contra-variance)
2. $\leq_{+}$: $+\{m_1 : T_1 \mid \cdots \mid m_{n+k} : T_{n+k}\} \leq +\{m_1 : T_1 \mid \cdots \mid m_n : T_n\}$ par restriction (covariance)

Ce qui donne une covariance de type pour les branchements (plus restrictif) et contra-variance pour les sélections (extensif).

De ce fait on peut supposer pouvoir tirer profit de ces inégalités pour pouvoir jauger un système s'il est substituable ou pas par un sous système composé d'un ou plusieurs composants assemblés ensemble, et pour cela nous proposons ceci :

Proposition 2

Pour qu'un type soit substituable de manière totale par un autre il faut que les deux pré-ordres (1 et 2) de la **Définition 7 (Sous-typage partiel ou pré-ordre ≤&, ≤+)** soient vérifiés : T et S deux types :

T est sous type de S ($T \leq S$) Si et seulement si $T \leq_{\&} S$ et $S \leq_{+} T$.

Si les deux sous-typages partiels sont prouvés on a un sous-typage strict (complet qui permet une substitution totale. Dans le cas particulier où j= k=0, la substitution est bijective donc parfaite.

La relation de sous-type T≤ S incarne la notion de substituabilité sûre: chaque terme ayant le type S peut être remplacé par un terme ayant le type T sans affecter le contexte d'une manière sensible.

5.2 Système de composition

Un système de composition crée par le biais d'un SMA (Système Multi- Agents) à partir duquel nous nous intéressons aux assemblages qui répondent à une fonctionnalité prédéfinie. Formellement, un système de composition est un tuple <R,⊕, ⊨> (Santhanam, Basu, & Honavar, 2011)

- $R = \{W_1, W_2 \ldots W_r\}$ est un ensemble de composants disponibles,
- ⊕ désigne un opérateur de composition qui agrège les composants fonctionnellement et encode toutes les indications fonctionnelles de la composition. ⊕ est une opération binaire sur les composants W_i, W_j dans le référentiel qui produit une composition $W_i \oplus W_j$.
- ⊨ Est une relation de satisfaction qui renvoie **vrai** quand une composition satisfait certaines propriétés fonctionnelles pré-spécifiées.

À partir de ce système de modélisation nous construisons la définition des compostions possibles et faisables pour pouvoir vérifier ensuite leur validité fonctionnelle avec celle souhaitée.

Définition 8 (compositions, compositions faisables)

Compte tenu d'un système de composition <R, ⊕, ⊨>, et une fonctionnalité φ, une composition $C = W_{i1} \oplus W_{i2} \ldots W_{in}$ est une collection aléatoire de composants W_{i1}, $W_{i2}, \ldots, W_{in}$ i.e. $\forall j \in [1, n]: W_{ij} \in R$.

φ est la substitution par sous-typage du type (S) de la spécification du système attendu par le type (T) du système composé, φ=S≤T. qui donne donc C ⊨ T≤S

- C est une composition possible chaque fois que C ⊨ φ;
- CP est une composition partielle possible chaque fois que :

$\exists$ $W_{j1}, W_{j2}. . . W_{jm} \in R$ tel que $C= W_{j1}. . . \oplus W_{jm}$ est une composition possible. Et sachant que la grammaire du type contient une description compositionnelle donc on peut déduire que la vérification d'une composition de différente granularité est possible.

Nous proposons :

$$C= \begin{cases} W_i & \text{si } C \in R \\ C \oplus C & \text{sinon} \end{cases}$$

Qui donne une composition sous forme d'arborescence binaire de vérification ce qui nous permet d'appliquer notre approche basée sur les contrats et les types de session (contrats de session) déjà expliquée.

Chaque composant W_i a une description qui donne accès à un contrat d'interface σ_i, la composition des composants implique celle des contrats, et ce par le biais de la grammaire (**Figure 20**)

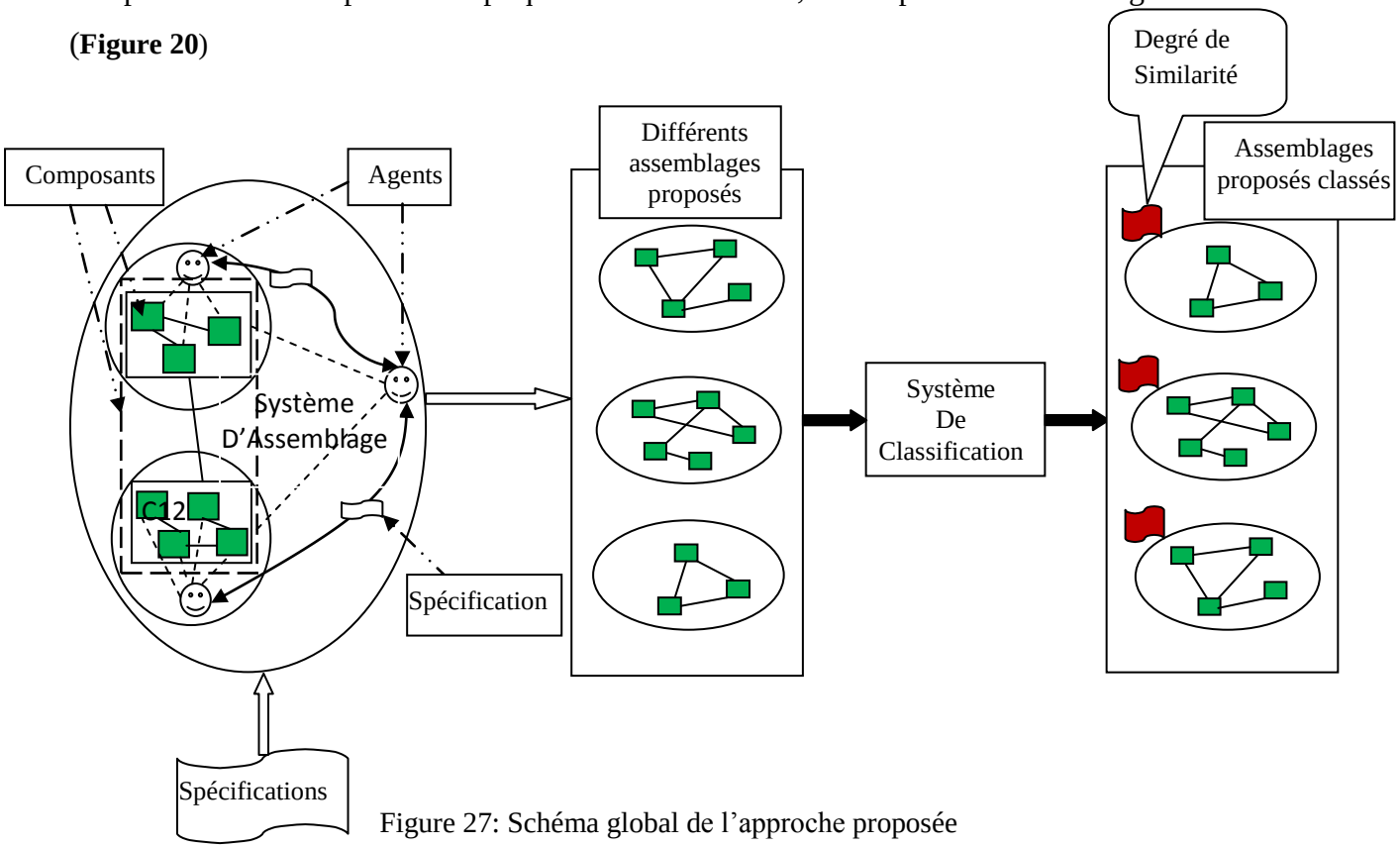


Figure 27: Schéma global de l'approche proposée

Exemple :

Nous supposons un système de trois composants et leurs contrats respectifs dont deux atomiques W_1 et W_2 (avec chacun son contrat *contr*) entrant dans la composition du composant composite C_{12}, et aussi le troisième C_3 quelconque.

La vérification se fait comme suit :

1ére étape :

En prenant d'abord W_1 et W_2 en déduisant leurs contrats de sessions (**Figure 20**) respectifs pour pouvoir vérifier s'ils sont compatibles en extrayant ensuite leurs translations par M (**Figure 21**) permettant de confirmer ou infirmer la validité du préordre de contrats de session($\sqsubseteq^{SC}$) entre eux, ceci nous permet de savoir de la compatibilité entre les deux composants en utilisant la **Définition 6 (compatibilité et substituabilité)Définition 6 (compatibilité et substituabilité).**

Si les deux composants sont compatibles ils composeront un nouveau composant C_{12} qui résulte des deux premiers en les combinant et de ce fait en combinant leurs interfaces et leurs contrats Contr1 et Contr2 pour à travers (**Figure 18: Grammaire de contrat**) et puis en suivant (**Figure 20: Grammaire des contrats de session**) pour extraire le nouveau contrat de session et on refait la boucle.

2eme étape :

Une nouvelle itération du même mécanisme cette fois avec comme composants le composite précédent C_{12} et un autre C_3, et ce de manière répétée jusqu'à n'avoir plus de composants disponible. En répétant le processus sur plusieurs combinaisons on aura éventuellement plusieurs compositions possibles.

En fin de parcours on n'aura soit aucune composition valide soit N composition possibles, N>=1.

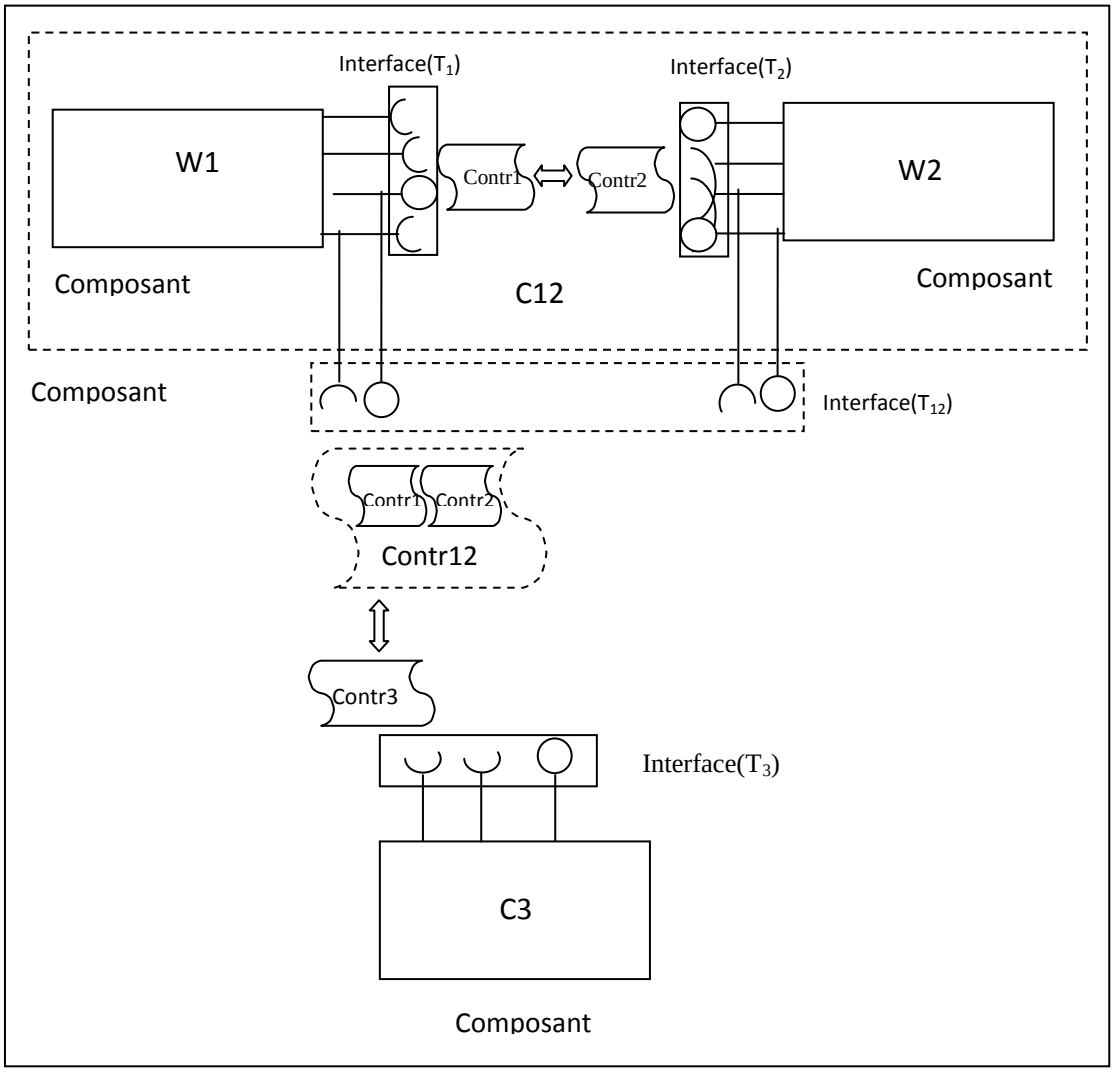


Figure 28: Modèle de l'approche proposée combinant contrat et type de session

5.3 Sous-typage et distance de Sous-typage

On a vu que plusieurs systèmes peuvent être valides et qui répondent aux exigences des spécifications des clients, ceci nous a poussé à penser donner une classification des compositions possibles, permettant ainsi à aider le développeur à avoir une vision pertinente et plus claire du système à concevoir.

Comparer deux systèmes implique l'utilisation d'un ou plusieurs critères mais pas de manière absolue, ce qui rend la comparaison contextuelle et donc personnalisée, plusieurs méthodes et techniques d'équivalences de modèles existent, plus au moins complexes, pour simplifier nous avons utilisé une comparaison en utilisant les branchements et sélections

comme indicateurs de distance entre deux systèmes de composition, que deux systèmes S1, S2 répondant aux spécificités du client, S1 est dit plus proche par rapport à S2 si il a moins de branchements et plus de sélections que S2.

Et pour pouvoir faire le calcul on a proposé ce qu'on a appelé un sous-typage partiel ce qui signifié un sous-type du coté des entrée (covariance) et inversement dans le principe coté sortie (contra variance).

Définition 9 (XOR)

XOR est une fonction qui vérifier la présence ou non d'une méthode dan une interface donnée.

On pose m_1 : méthode de type T1 et T' un type non nul (qui correspond à une interface non vide)

$$XOR(m_1:T_1, T') = \begin{cases} 1 \text{ si } \forall i \in (1..n_{T'})\ m_1 \neq m_i \\ 0 \text{ sinon} \end{cases}$$

Le calcul de la distance se base sur la **Définition 9 (XOR)** donc :

Définition 10 (Système de distance)

$$DS(T,T') = \begin{cases} -1 \text{ Ssi } T, \overline{T'} \text{ incompatibles} \\ 0 \text{ si } T \leq T' \text{ et } T' \leq T \\ \sum_i^n XOR[(mi:Ti), T'] \\ DS(T,(T1 \cup T2)) \qquad T' = T_1 \oplus T2\ \{T1 \cup T2 = \&(T1 \oplus T2) | + (T1 \oplus T2)\} \end{cases}$$

Définition 11 (ordre de substitution)

Si T, T' et S sont des types, et $T,T' \leq S$

$$\text{S est mieux substituable par } \begin{cases} T & \text{si } [T' \leq T \text{ et (not } (T' \not\leq T) \text{ ou } DS(T,S) < DS(T',S)] \\ T' & \text{sinon} \end{cases}$$

Et puisque le sous-typage est un Préordre (voir 3.5.1.2 Propositions 1(Vallecillo, Vasconcelos, & Ravara, 2005)) donc transitif cet ordre de substitution est aussi transitif. Ce qui

nous permet de proposer l'algorithme de classification des compositions possibles d'un système :

Nous proposons l'ensemble contenant toutes les compositions C= (C_1, C_2 …Cn) répondant aux spécifications du système de différentes manières $\forall i \in [1..n]$ $C(i) \models T_{Ci} \leq S$:

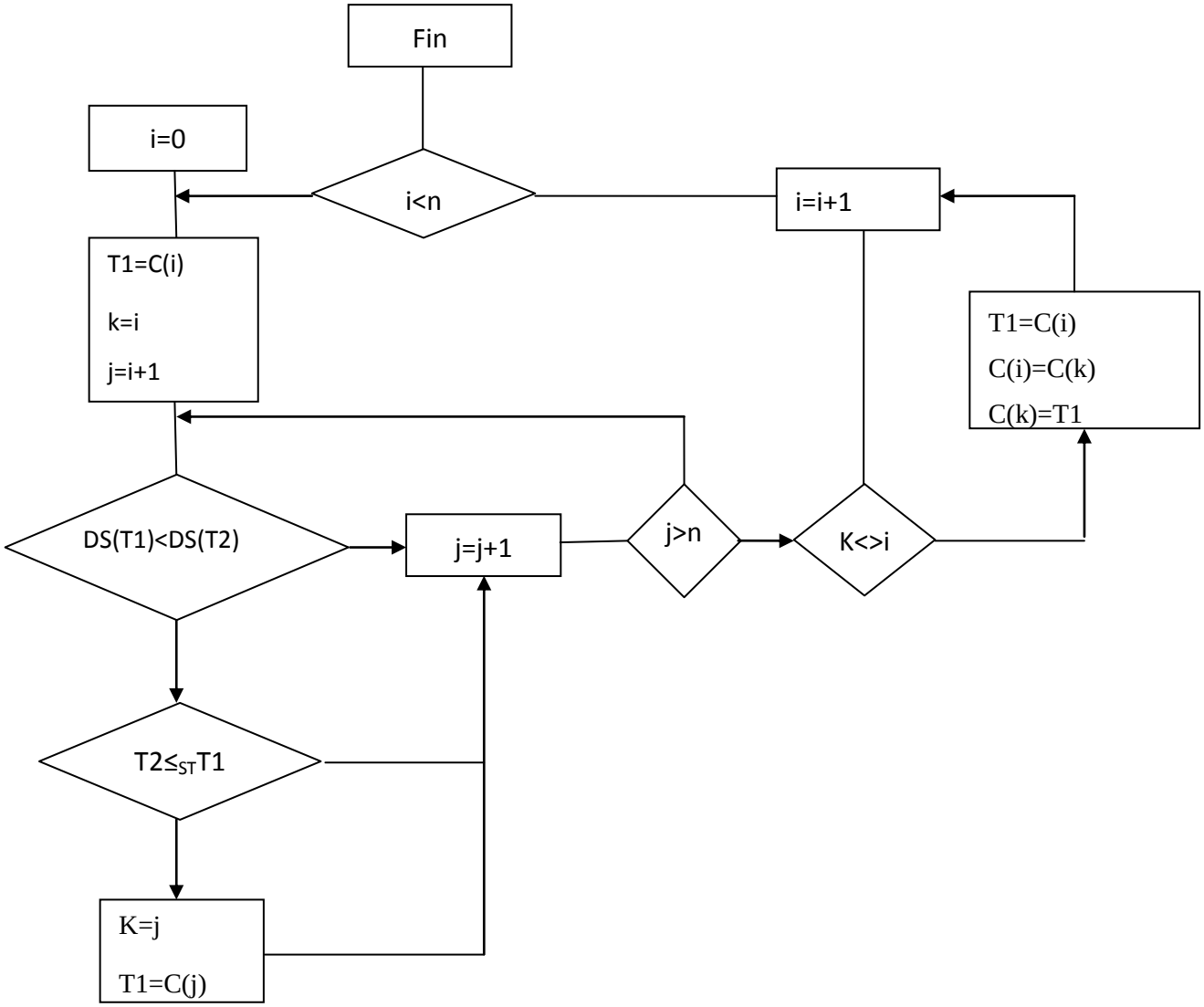


Figure 29: Algorithme de classification des assemblages proposés

En constatant la mobilité croissante des sites et la simultanéité apportée aux utilisateurs, nous supposons que les agents mobiles proposent le modèle le mieux adapté à la prise en compte de ces environnement dynamiques, ceci grâce à leur autonomie et leur mobilité.

5.4 Pour quoi les agents

Le processus d'assemblage et de vérification de composants fait partie d'un processus plus global qui lui se préoccupe de la conception et développement d'applications à base de composants, et sachant qu'il n'existe pas de modèle globale et complet permettant de prendre en charge tout le processus, nous nous sommes intéressé à une phase assez pertinente dans le développement de ces systèmes, donc un processus d'assemblage et de vérification est intrinsèquement interactif qui met en œuvre des mécanismes différents (la recherche, la collecte d'informations, la mise en correspondance d'interfaces…) et qui manipule différents types de connaissances (les connaissances sur les composants). La mise en œuvre d'un tel processus nécessite un système capable d'autonomie (capacité à agir indépendamment d'une intervention humaine), de réactivité (capacité à percevoir son environnement et à répondre aux événements qui y surviennent), de pro-activité (capacité à prendre des initiatives si la situation le demande) et doté de capacités sociales (capacités à interagir intelligemment et d'une manière constructive avec les autres composants du système et/ou les utilisateurs). Les agents possèdent ces caractéristiques. Nous proposons donc un système d'agents système pour assister la mise en œuvre de ce processus.

En utilisant la spécification du système à construire comme source d'informations en entrée conformément au formalisme commun entre agents et la description des composants basé sur les contrats et les types de session, la compréhension est facile et donc le processus d'assemblage et de vérification est immunisé contre les incompatibilités souvent responsables d'erreurs d'appréciations conduisant à des fautes qui peuvent engendrer des disfonctionnements couteux.

Les agents dispersés collectent des informations et par le biais de la spécification du système attendu traduisent les besoins en vecteurs de vérification et par des comparaisons syntaxiques d'abord puis sémantiques comportementales des composants découverts ils communiquent leurs résultats aux autres au fur et à mesure qu'ils avancent. Et ce processus est répété par de nouvelles demandes ou lors des changements des états de leurs environnements respectifs qui sont les composants.

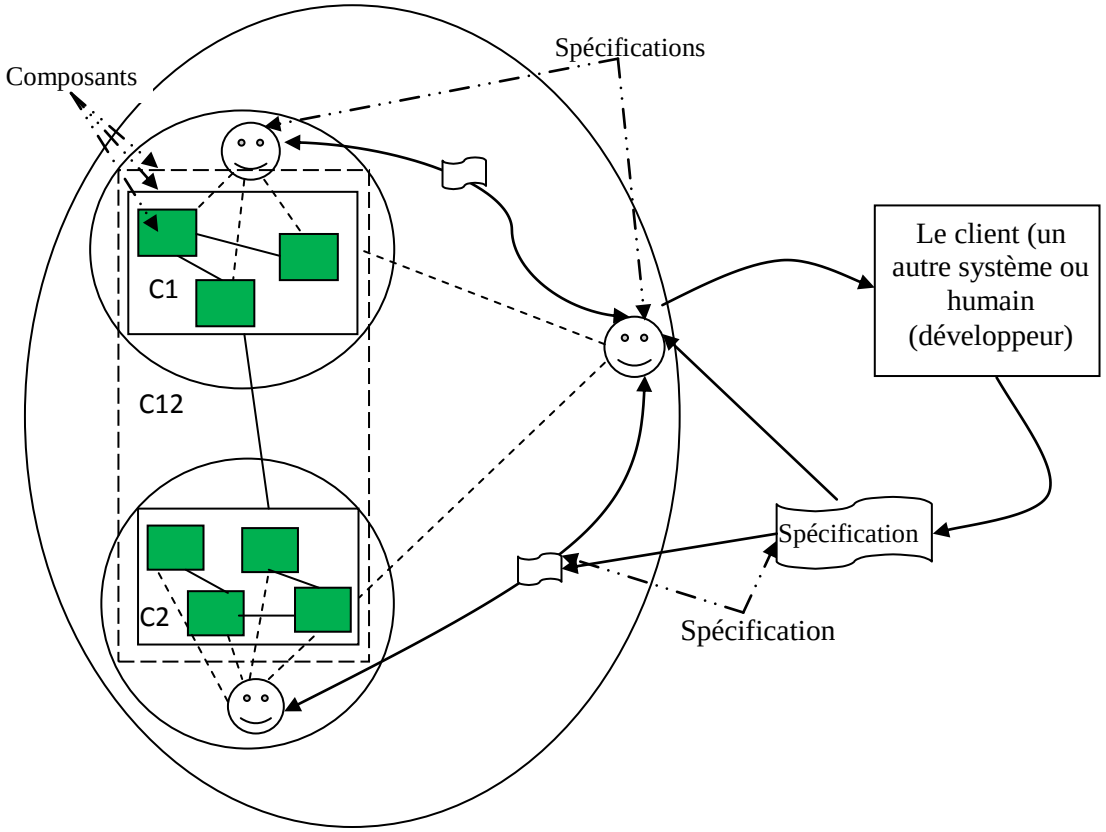


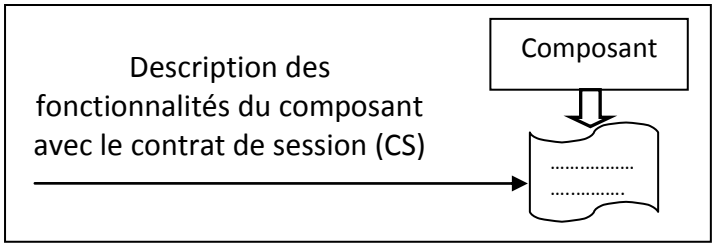
Figure 30: Coopération entre agents pour l'assemblage des composants

Ce système de vérification permet une mobilité physique et logique, ce qui nous autorise à utiliser les SMA[10] et spécialement les agents mobiles qui représentent un début de solution aux problèmes de reconfiguration, pannes, extension,… dans les applications modernes, avec une politique claire et adéquate du système multi agents nous construisons ce système coopératif entre agents qui communiquent leurs informations respectives des sous systèmes du voisinage de chaque agent.

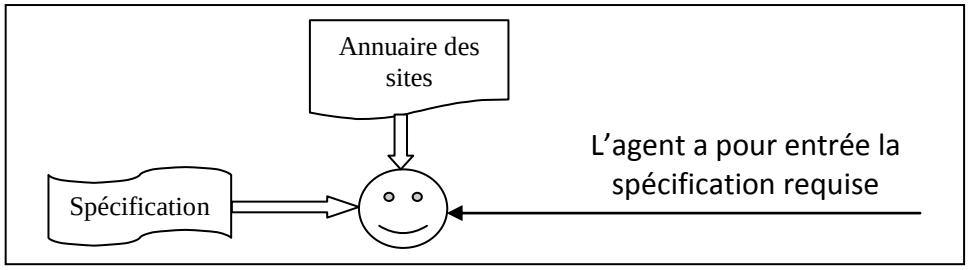
Vérifier un assemblage par le biais d'agents à travers la collecte d'informations sur les différentes parties du système et ce à travers un langage standard qui est le contrat de session, en comparant les différents sous systèmes en utilisant le concept de sous typage, qui filtre les bons et les mauvais assemblages c.à.d. qui ne répondent pas aux spécifications du client.

Un composant a par nature une description de ses services (offerts et requis), nous avons formalisé les contrats de session (CS) comme formalisme de haut niveau.

[10] Système Multi Agents



Les agents dispersés consultent l'annuaire des adresses des machines voisines et accessibles en embarquant les spécifications requises du système souhaité.



Le système est composé de plusieurs sites formant un réseau, ce réseau de machines comporte les composants de l'assemblage attendu, ce qui représente un champ d'action pour les agents qui inspectent ces sites et vérifient les différents assemblages possibles (une vérification fonctionnelle).

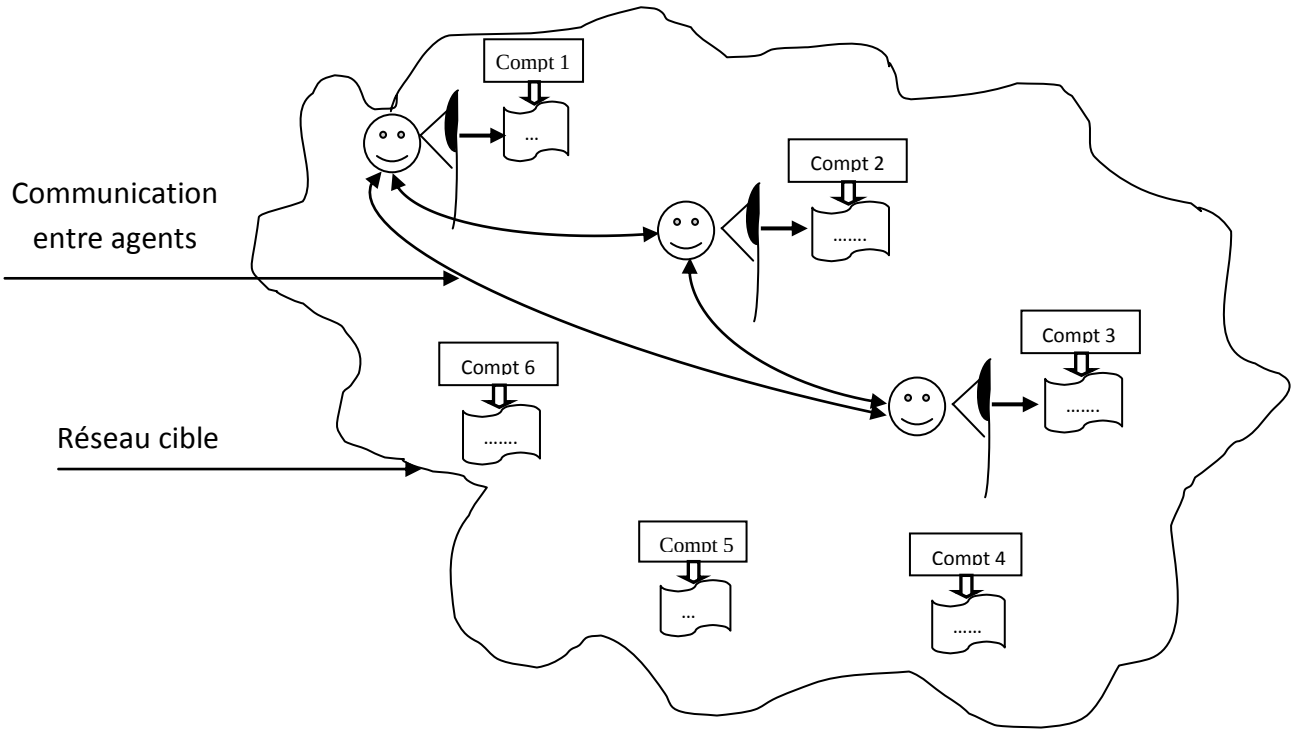


Figure 31: Implication des agents et leur rôle vérificateur sur le réseau

Pouvoir trier et ordonner les combinaisons possibles et qui satisfont les critères du développeur est réalisé en passant par l'algorithme (**Figure 29**).

5.5 Conclusion

Nous avons vu comment peut-on intégrer les contrats de session (combinaison des contrats et des types de session), et puis permettre un mécanisme d'assemblage à l'aide d'agents mobiles et nous avons vu le l'algorithme de classement des assemblages qu'on a proposé, l'idée même d'associer les agents au mécanisme d'assemblage à l'aide du formalisme proposé est récente. Ce qui nous donne une idée sur l'opportunité d'automatiser l'assemblage et faciliter par ça la tache des développeurs.

Chapitre VI: Etude de cas : sous contrat et sous typage à un système de vote

6.1 Introduction

Les types de session et les contrats trouvent leurs origines dans deux domaines différents (Laneve & Padovani, 2008): les premiers découlent du domaine de la théorie des types et des systèmes de type, alors que les contrats sont davantage liés à l'étude des équivalences de comportement, comme la bi-simulation et les tests d'équivalence. Les deux langages sont équipés avec des constructeurs similaires. L'exemple de (Laneve & Padovani, 2008) que nous avons pris dessous représente un service simple pour un vote en ligne.

Vu que ce genre de système est basé sur une architecture repartie et qui peut être constitué de plusieurs composants dispersés sur un ensemble de machine qu'on doit assembler pour former le bon système, ceci exige un effort considérable pour collecter les informations nécessaires pour pouvoir ensuite choisir les bons composants à assembler, Ce système nous permet d'illustrer l'utilisation de notre approche par l'intégration du formalisme de description de haut niveau utilisé (contrats de session) qui à l'aide d'agents permet de se substituer en partie aux développeurs dans la tache de la vérification des assemblages éventuels.

6.2 Description de l'étude de cas

Ce système de vote électronique (utilisé par exemple aux Etats Unis). Comme dans système de vote classique manuel, avant qu'un client (électeur) est autorisé à voter, l'électeur doit fournir un jeton de connexion valide que le système utilise pour faire en sorte que les préférences sont exprimées tout au plus une fois, faute de quoi l'électeur est identifié comme un tricheur. L'architecture générale est composée de :

- Un serveur qui sert à délivrer des droits de vote
- Plusieurs serveurs distincts servent à voter et qui sont des clients du serveur principal

Plusieurs composants sont nécessaires pour former ce système (interfaces d'électeur, authentification, enregistrement des choix…), des composants pour des taches centralisées et d'autres décentralisées. Une description fonctionnelle commune de ces composants représentée par les contrats de session est supportée par les agents qui sont responsables d'une collecte d'informations et qui constituent le mécanisme de la vérification des assemblages.

Un serveur propose au client électeur une interface d'authentification sert à vérifier l'identité de l'électeur, les données saisies et renvoyées, un refus est émis au électeur s'il a mal renseigné, le cas échéant un choix lui est offert, le vote qu'il choisisse entre plus d'un est renvoyé au serveur et enregistré.

La relation entre les types de session et des contrats concerne la sémantique, bien que, dans ce cas, leurs origines différentes parce que les relations sont souvent opposées.

Prenons le type de session suivant:

$S=\mu x.\ \&(?Login.(\ !Wrong.x \oplus !Ok.(?VoteA.x+?VoteB.x\ +?VoteC.x+?VoteD.x)\ )$

$T=\mu x.\ \&(?Login.(\ !Wrong.x \oplus !Ok.(?VoteA.x+?VoteB.x\ )\ )$

Et les contrats

$\sigma= \mu x.\ ?Login.(\ !Wrong.x \oplus !Ok.(?VoteA.x+?VoteB.x\ +?VoteC.x+?VoteD.x))$

$\rho= \mu x.\ ?Login.(\ !Wrong.x \oplus !Ok.(?VoteA.x+?VoteB.x))$

Les types de session et les contrats ci-dessus décrivent un service de bulletin qui accepte toujours les jetons de connexion quel que soit leur validité. Il s'avère que T est un sous-type S ($T \leq S$) et ρ est sous-contrat de σ ($\sigma \sqsubseteq \rho$), ce qui signifie que T peut se substituer à S offrant au moins les mêmes services, et le service à contrat conforme avec σ est aussi sûr avec ρ. $\&(\ !VoteA :end, !VoteB :end) \leq \&(\ !VoteA :end)$ (le service de bulletin de vote peut être étendu avec plus de candidats VoteB sans invalider les anciens électeurs). À l'inverse les contrats sont discriminatoires.

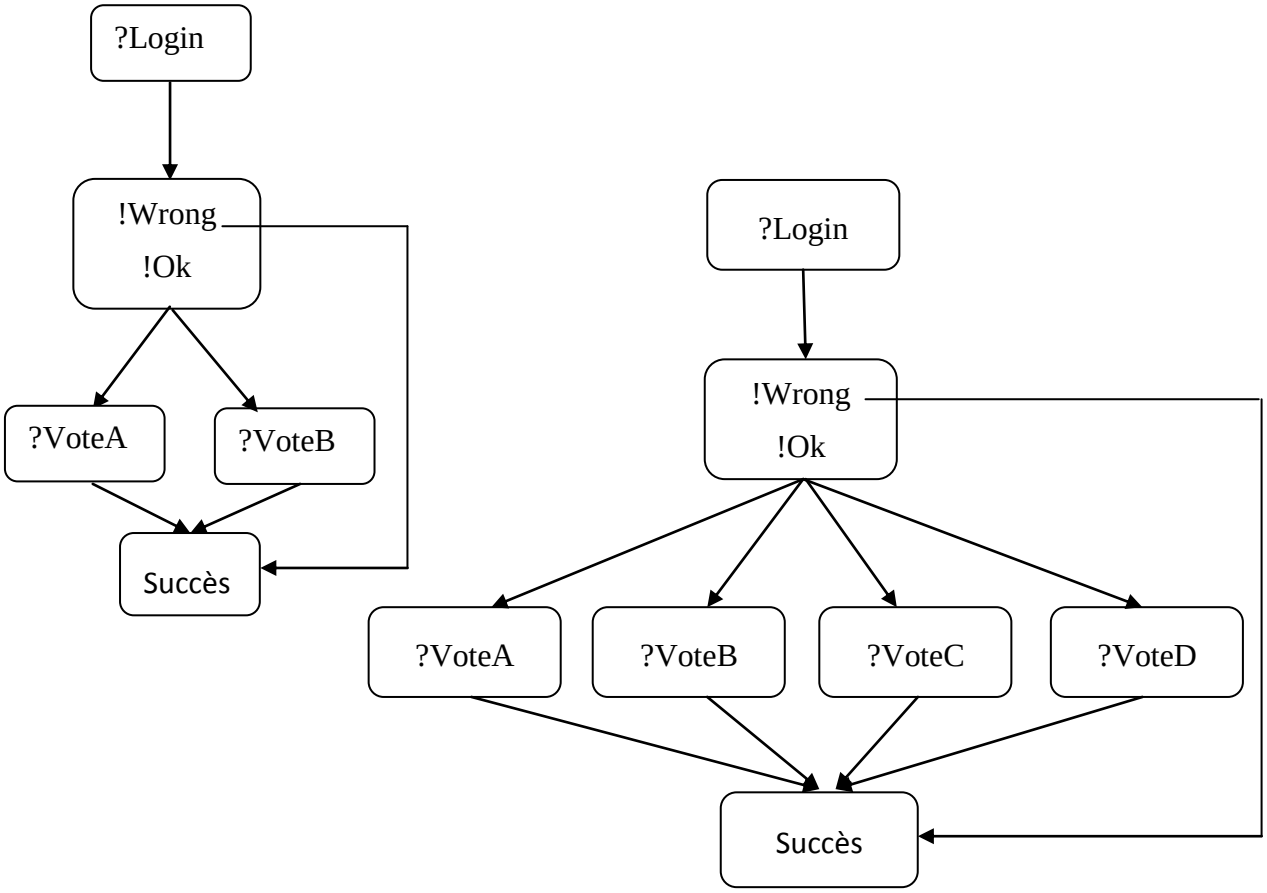


Figure 32: Schématisation des deux contrats σ et ρ

σ_1={VoteA , VoteB }[!VoteA] et σ_2={VoteA,VoteB }[!VoteA+ !VoteB]

et le client ρ={VoteA,VoteB }[?VoteA.end+ ?VoteB.?VoteB.end]

Là ρ peut communiquer avec σ_1 mais non avec σ_2. Donc $\sigma_1 \not\subseteq \sigma_2$ puisque voter deux fois VoteB puis terminer exigé par ρ n'est pas permis dans σ_2.

Prenons**: Ballot** qui représente le contrat du serveur de vote et **Voter** celui du composant électeur, NB : **.**=Successif, ?=Réception, !=Envoi, ⊕=Choix interne, +=Choix externe

Ballot = μx.?Login.(!Wrong.x⊕!Ok.(?VoteA.x+?VoteB.x))

Voter = μx. !Login.(?Wrong.x+ ?Ok.(!VoteA.1⊕!VoteB.1))

Un processus offrant le contrat Ballot implémente un service de vote électronique. Un tel service permet à un client de se connecter (Login) Si le Login échoue les services repartent à

zéro, alors que si le log réussit deux actions sont proposées à l'environnement, à savoir VoteA et VoteB.

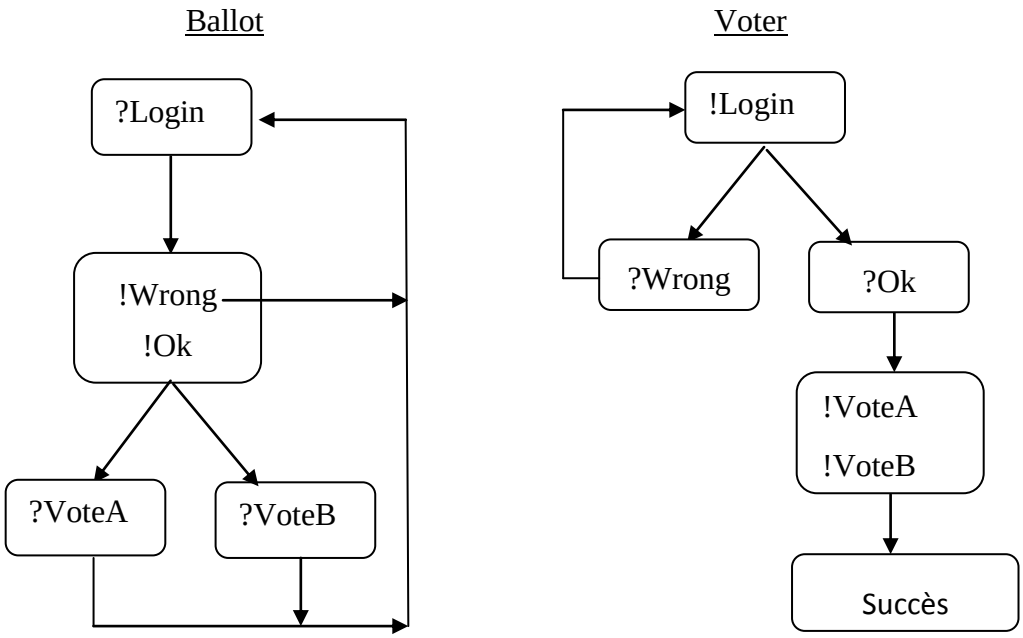


Figure 33: Graphe représentant les contrats "Ballot" et "Voter"

Les deux contrats **Ballot** et **Voter** sont compatibles puisque **Voter** répond à **Ballot** de manière duale. Donc Ballot⋈Voter

Le contrat (Voter) est un client récursif pour le protocole décrit par le contrat Ballot

Considérant le contrat de session déduit à partir de la grammaire de la **Figure 20**:

Ballot = μx.?Login.(!Wrong.x⊕!Ok.(?VoteA.x+?VoteB.x))

BallotB = μx.?Login.(!Wrong.x⊕!Ok.(?VoteA.1+?VoteB.1+?VoteC.1+?VoteD.1))

BallotB offre à un électeur (Voter) plus d'options que Ballot, et intuitivement, il devrait être possible d'utiliser un serveur qui garantit BallotB en place d'un serveur qui garantit Ballot. Ce n'est pas le cas si les contrats sont comparés de manière classique, parce que BallotB est plus général que Ballot donc moins précis. D'autre part, en limitant l'attention aux contrats de session (CS), et donc à la définition du pré-ordre $\sqsubseteq_{srv}^{SC}$, nous avons : Ballot $\sqsubseteq_{srv}^{SC}$ BallotB.

Donc BallotB peur remplacer Ballot en terme de contrat de session en assurant une compatibilité du coté du serveur Ballot qui reste compatible avec tous les anciens clients Voter.

Ce qui n'est pas le possible sans les contrats de session (plus en détaille dans Lemme 4.8 (Bernardi & Hennessy, Giovanni Tito Bernardi)).

Traitant maintenant le cas d'interopérabilité avec plusieurs clients (on parle des clients du serveur de vote)

Le contrat du serveur Ballot:

Ballot = μx.?Login.(!Wrong.x⊕!Ok.(?VoteA.x+?VoteB.x))

Et deux contrats de deux clients :

VoterA = μx. !Login.(?Wrong.x+ ?Ok.!VoteA.1)

VoterB = μx. !Login.(?Wrong.x+ ?Ok.(!VoteA.1⊕!VoteB.1 !VoteB .1))

Ballot répond à VoterA et a un choix en plus (+?VoteB.x) qui est neutre pour lui, mais pour VoterB un vote double VoteB n'est pas pris en charge par le serveur ce qui fait que VoterA$\sqsubseteq_{CLT}^{SC}$VoterB est invalide donc le clientVoterB ne peut par se substituer à VoterA.

On Considère aussi le contrat BallotA tel que: BallotA⊑SCBallotB (Définition 5 [Préordre des contrats de Session])

Et en supposant le type de session : $BallotA=M^{-1}(BallotA)$ et $BallotB= M^{-1}(BallotB)$ par la fonction inverse de mappage (**Figure 21**) en application du (théorème 5.7) (Bernardi & Hennessy, 2012) $BallotA \leq BallotB$

Ça peut donner : BallotA =?Login.(!Wrong.1⊕!Ok.(?VoteA.1+?VoteB.1))

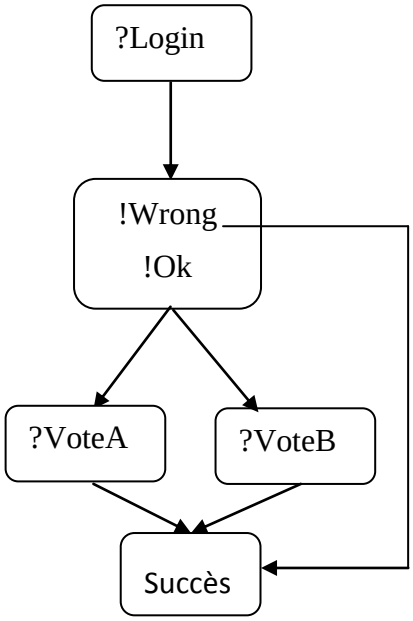


Figure 34: Graphe représentant les contrats " BallotA "

BallotB =?Login.(!Wrong.1⊕!Ok.(?VoteA.1+?VoteB.1+?VoteC.1+?VoteD.1))

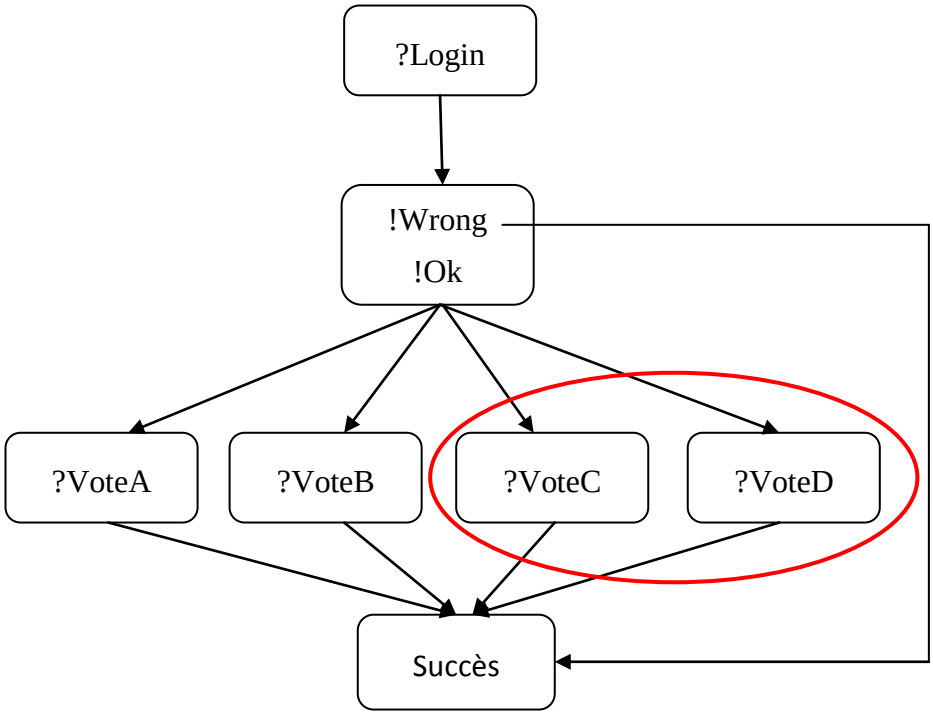


Figure 35: Graphe représentant les contrats " BallotB "

Voter = M^{-1}(!Login.(?Wrong.x+ ?Ok.(!VoteA.1+ !VoteB.1)))

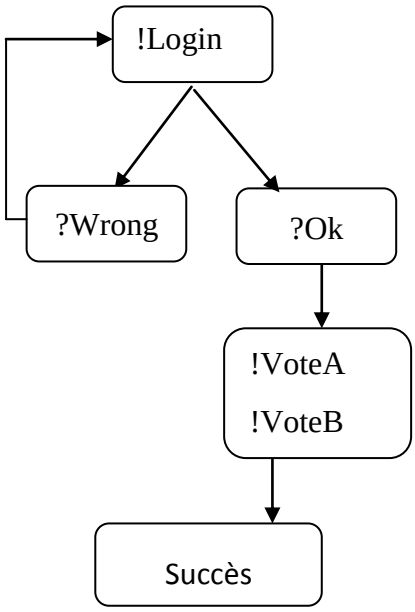


Figure 36: Graphe représentant le contrat " Voter "

Qui satisfont les hypothèses de sous-typage des types de session *BallotA* ≤ *BallotB* sans pour autant violer les contrats (théorème 5.7) (Bernardi & Hennessy, 2012).

Par transformation inverse de M on obtient le type de session *BallotA* à partir du contrat BallotA précédant :

BallotA=M^{-1}(BallotA) = ?(Login). M^{-1}(!Wrong.1⊕!Ok.(?VoteA.1+?VoteB.1))

=?Login;(M^{-1}(!Wrong.1) | M^{-1}(!Ok.(?VoteA.1+?VoteB.1)))

=?Login; + (!Wrong; end |!OK; M^{-1}(?VoteA.1+?VoteB.1))

=?Login; + (!Wrong;end | !OK; &(M^{-1}(?VoteA.1)+ $M^{-1(}$?VoteB.1)))

=?Login; + (!Wrong;end | !OK; &(?VoteA;end+ ?VoteB;end))

=?Login; + (!Wrong;end | !OK; &(?VoteA+ ?VoteB);end)

BallotB=?Login; + (!Wrong;end , !OK; &(?VoteA+ ?VoteB+ ?VoteC+ ?VoteD); end)

Et puisque BallotA=*M(BallotA)* $\sqsubseteq^{SC}$ BallotB= *M(BallotB)* ceci implique *BallotA*≤*BallotB* et donc BallotB peut en toute sécurité remplacer BallotA,

Appliquant ce mécanisme en généralisant sa mise en œuvre et en vérifiant les composants deux à deux de telle sorte qu'on puisse évaluer des compositions de manière compositionnelle et en commençant par deux composants ensemble qui soient compatibles puis leur combinaison en tant qu'un seul composant avec un troisième et ainsi de suite jusqu'à atteindre le système (ou les systèmes) possibles requis par la spécification du client.

En prenant le même exemple de vote et on ignore l'idée qu'il y a un service en qualité de serveur (Ballot, BallotA, BallotB….) et un autre en qualité de client (Voter), et on applique les règles et définitions précédemment expliquées de façon évolutive de telle sorte qu'à chaque nouvelle étape d'ajout de composant(atomique ou composite) au système qu'on a déjà, on vérifie les deux parties (sous-systèmes) de manière à satisfaire toutes les exigences de types et de sureté du système assemblé.

Pour simplifier nous utilisons une codification simple qui va de soi avec la codification utilisée dans nos propositions. Nous illustrons l'application de ce calcul par un exemple, l'ordre de distance entre trois composants :

6.2.1 Calcul de distance entre composants

Suivant le système de modélisation de la composition (voir5.2 Système de composition) nous supposons : C, C'== (w1⊕w2), C"= (w3⊕w4⊕w5), trois composants

Tel que : w1=&(m1 :T1, m2 :T2)|+(m3 :T3) ;

w2=&(m4 :T4)|+(m5 :T5) ;

w3=& (m1 :T1)|+(m3 :T3) ;

w4=&(m2 :T2)|+(m5 :T5) ;

w5=&(m6 :T6))|+(m7 :T7)

Ce qui nous donne en typage des sessions :

C=&(m1 :T1,m2 :T2)|+(m3 :T3, m5 :T5, m6 :T6)

C'=&(m1 :T1,m2 :T2,m4 :T4)|+(m3 :T3, m5 :T5)

C"=&(m1 :T1,m2 :T2, m6 :T6)|+(m3 :T3, m5 :T5, m7 :T7)

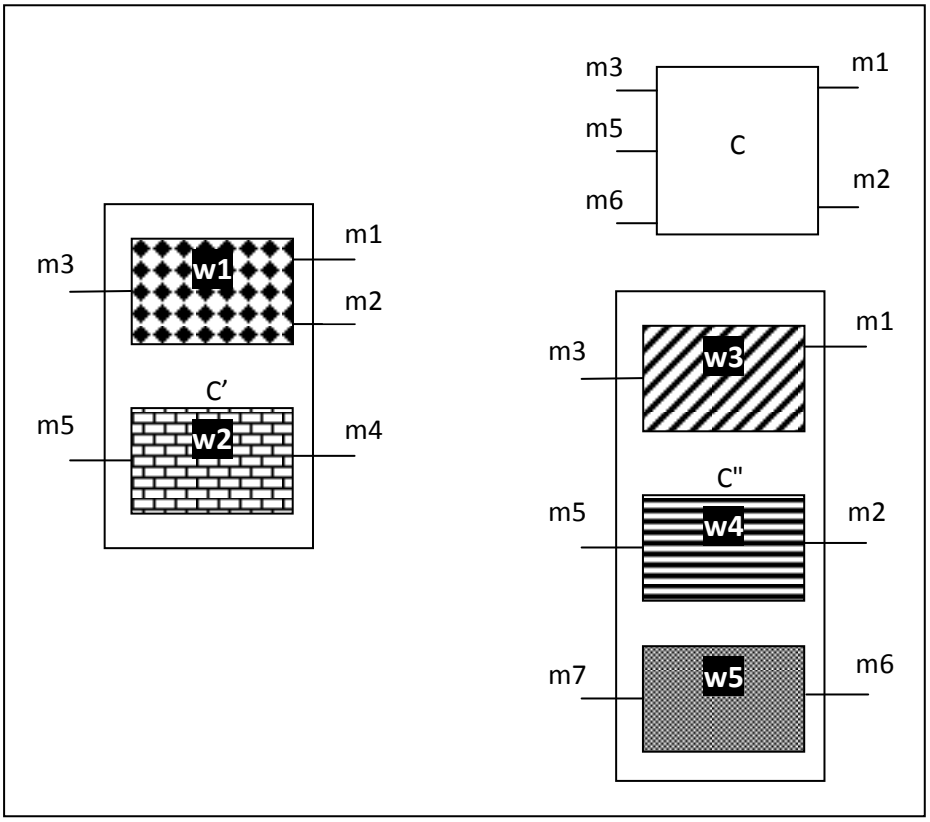


Figure 37: Différents composants décrits dans l'exemple

C= &(m_1 :T_1,m_2 :T_2)|+(m_3 :T_3, m_5 :T_5, m_6 :T_6)

C'=($w_1 \oplus w_2$)=&(m_1 :T_1,m_2 :T_2,m_4 :T_4)|+(m_3 :T_3,m_5 :T_5)

C"=(w3⊕w4⊕w5)+&(m_1 :T_1,m_2 :T_2,m_7 :T_7)|+(m_3 :T_3, m_5 :T_5, m_6 :T_6)

On voit les deux pré-ordres partiels entre les interfaces de Cet C' :

- &(m1 :T_1,m_2 :T_2,**m_4 :T_4**) $\leq_{\&}$=&(m_1 :T_1,m_2 :T_2)
- +(m_3 :T_3, m_5 :T_5, **m_7 :T_7**)$\leq_{+}$+(m_3 :T_3,m_5 :T_5) donc (C'≤C)

Et entre C et C"

- &(m1 :T1,m2 :T2,m6 :T6) ≤&=&(m1 :T1,m2 :T2)
- +(m3 :T3, m5 :T5)≤++(m3 :T3,m5 :T5) donc (C"≤C)

Et on calcule la distance entre les deux paires (C',C) et (C",C):

DS (C', C)=1+1=2 (la différence est m_4 :T_4 et m_6 ;T_6)

DS (C", C)=1 (la différence est m_7 :T_7)

Donc C" est plus proche de C que C’ donc il est plus pertinent de ce point de vue.

6.3 L’environnement ECLIPSE

Eclipse IDE est un environnement de développement intégré libre (le terme Eclipse désigne également le projet correspondant, lancé par IBM) extensible, universel et polyvalent, permettant potentiellement de créer des projets de développement mettant en œuvre n'importe quel langage de programmation. Eclipse IDE est principalement écrit en Java (à l'aide de la bibliothèque graphique SWT, d'IBM), et ce langage, grâce à des bibliothèques spécifiques, est également utilisé pour écrire des extensions.

La spécificité d'Eclipse IDE vient du fait de son architecture totalement développée autour de la notion de plug-in (en conformité avec la norme OSGi) : toutes les fonctionnalités de cet atelier logiciel sont développées en tant que plug-in.

La base de cet environnement de développement intégré est l'Eclipse Platform qui est composée de :

- Platform Runtime démarrant la plateforme et gérant les plug-ins
- SWT la bibliothèque graphique de base de l'EDI
- JFace une bibliothèque graphique de plus haut niveau basée sur SWT
- EclipseWorkbench qui est la dernière couche graphique permettant de manipuler des composants tels que des vues, des éditeurs, des perspectives...

Ces composants de base peuvent être réutilisés pour développer des clients lourds indépendants d'Eclipse grâce au projet Eclipse RCP (Rich Client Platform).

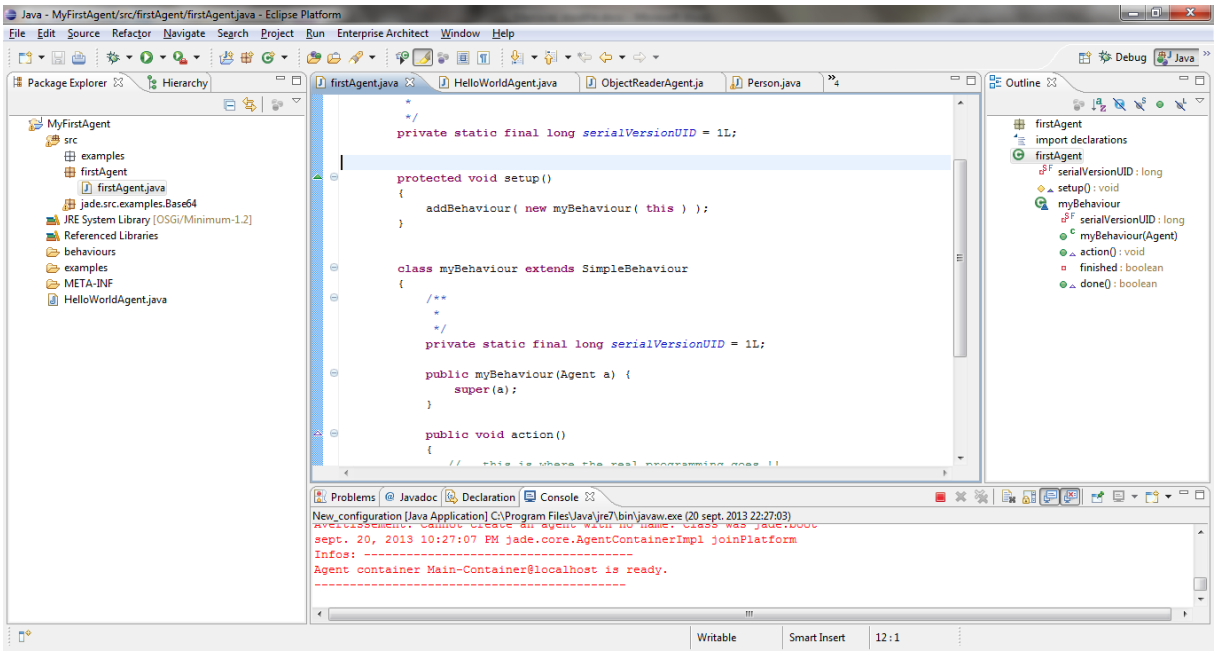


Figure 38: Interface ECLIPSE

```
package Composant;
import javax.ejb.Local;

@Local
public interface composantVoteLocal {
    public void authentification (String identifiant, String nom, String Prenom);
    public void vote (String choix);
    public void modifierVote (String identifiant);
    public void confirmerVote();
}
```

Figure 39: Interface d'un composant EJB

```
import Composant.compoantVoteRemote;


public class composantVoteClient {

    public static void main(String[] args) {
        Properties props = System.getProperties();
        props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
        props.put(Context.PROVIDER_URL, "jnp://localhost:1099");

        Context ctx;
        compoantVoteRemote interfacecompoantVote = null;

        try {
            ctx = new InitialContext(props);
            interfacecompoantVote = (compoantVoteRemote) ctx.lookup("composantVote/remote");

        } catch (NamingException e4) {
            // TODO Auto-generated catch block
            e4.printStackTrace();
```

Figure 40: Le composant client du composant de vote

Un composant de serveur de vote décrit par son interface *composantVoteLocal*, *composantVoteRemote* (*authentification, vote, modifierVote*, *confirmerVote*) permet un échange avec les clients, à travers cette interface qui est implémentée dans notre cas en EJB avec ECLIPSE et sous JBoss. On modélise cette interface par le formalisme (contrat de session) proposé plus haut, les contrats qui sont intégrés ainsi que les types de session (

Figure **32**).

Le bout de code de la (Figure **40**) comporte les le détail d'implémentation du client, le client fait appel aux procédures distantes qui sont offertes par le serveur de vote.

Les contrats *Ballot* et *Voter* (**Figure 33**) contraignent l'exécution des méthodes authentification (Login) et vote, confirmerVote et modifierVote (Wrong, OK). Ceci représente le moyen formel compréhensible par les différents intervenants dont les agents systèmes, les agents qui vont aider à vérifier l'assemblage des composants.

Les agents interviennent sur les interfaces des composants à travers les règles d'inférences des contrats de session (CS) et le sous typage, l'algorithme (**Figure 29**) de classement des assemblages.

6.4 La plateforme JADE

Le meilleur moyen pour construire un système multi-agent (SMA) est d'utiliser une plate-forme multi-agent. Une plate-forme multi-agent est un ensemble d'outils nécessaire à la construction et à la mise en service d'agents au sein d'un environnement spécifique. Ces outils peuvent servir également à l'analyse et au test du SMA ainsi créé. Ces outils peuvent être sous la forme d'environnement de programmation (API) et d'applications permettant d'aider le développeur JADE (Java Agent DEvelopement framework) est une plate-forme multi-agent créé par le laboratoire TILAB et décrite par Bellifemine et al. Dans (Bellifemine, Poggi, & Rimassa, 1999). JADE permet le développement de systèmes multi-agents et d'applications conformes aux normes FIPA . Elle est implémentée en JAVA et fourni des classes qui implémentent « JESS » pour la définition du comportement des agents. JADE possède trois modules principaux (nécessaire aux normes FIPA).

JADE encapsule les spécifications de la FIPA :

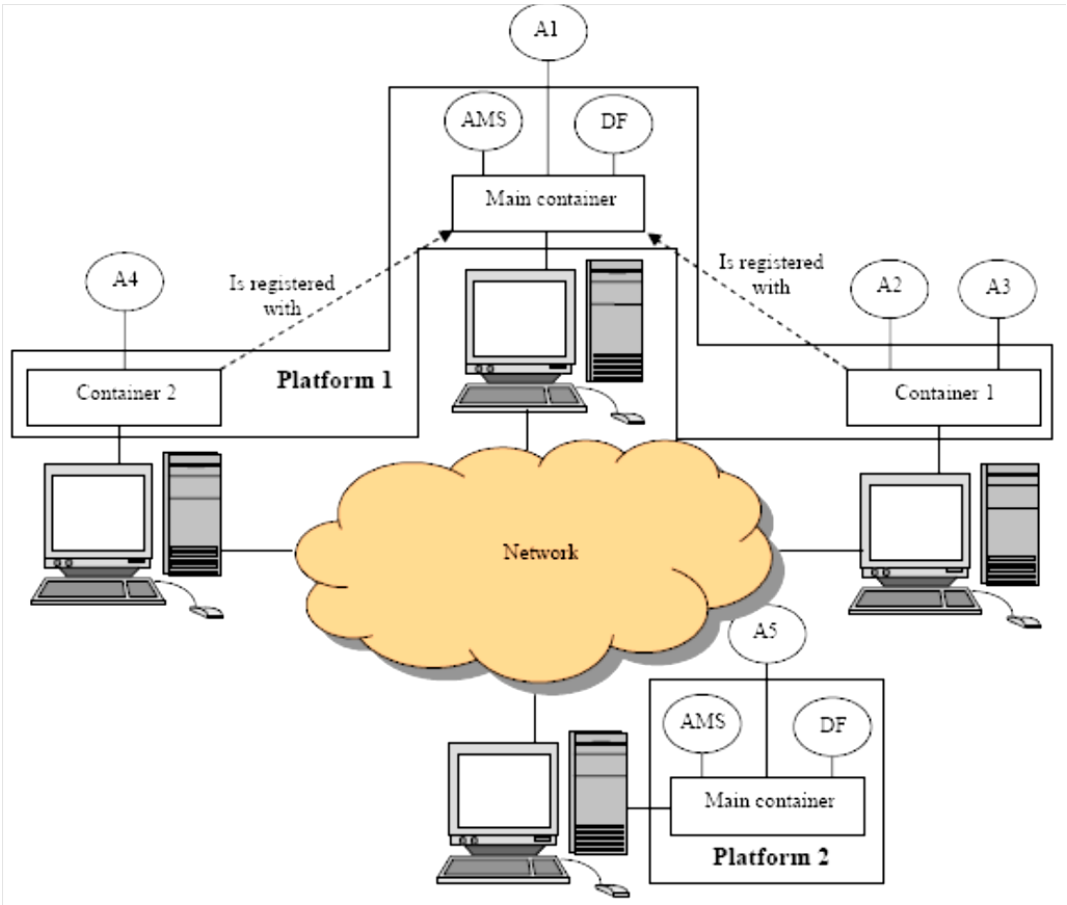


Figure 41: L'environnement conteneur de JADE **(Boissier)**

- La plateforme fournit : AMS, DF, MTS et ACC
- Agent-management-ontology, constructeur d'un agent l'enregistre dans la plateforme (nom, adresse), classe DFService permet un accès au DF
- Transport et traitement des messages
- Extension des protocoles standards d'interaction par les méthodes handle

Système de gestion d'événements dans le noyau de la plateforme

- Permet l'observation de la plateforme, des messages, du transport des messages, des agents

Outils de gestion basés sur des agents

- Agents spéciaux (RMA, Sniffer, Introspector) qui communiquent avec FIPA ACL

- Extensions à l'ontologie fipa-management-ontology pour y inclure des actions spécifiques
- Ontologie particulière pour l'observation jade-introspection

Agents utilitaires

- DummyAgent tool permet à des utilisateurs d'interagir avec les agents déployés sur la plateforme
- Sniffer Agent agent utilisé pour observer les messages

6.4.1 Installation et configuration

Installation : Sources disponibles à http://jade.tilab.com

Configuration : Ajouter dans le classpath les archives java (.jar) se trouvant dans le répertoire lib.

Archives nécessaires : jade.jar, jadeTools.jar, Archives optionnelles: http.jar, iiop.jar

Lancement de base : java jade.Boot [liste agents]

Lancement avec interface graphique : java jade.Boot **-gui** [liste agents]

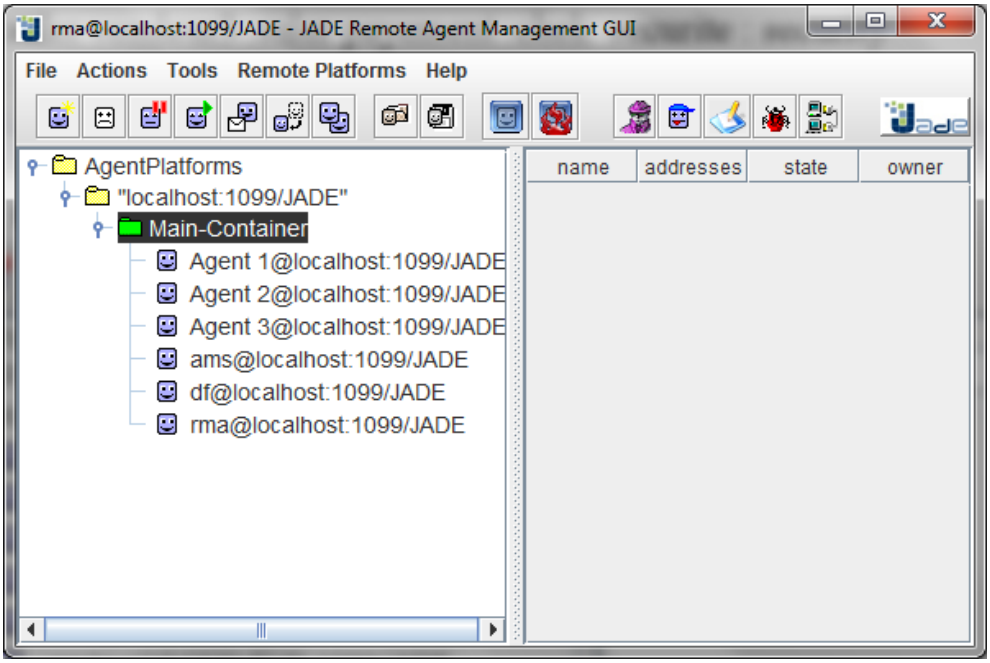


Figure 42: Simulation des agents de vérification de l'assemblage

Un message KQML se compose alors d'un performatif (action à faire) qui est l'en-tête du message et d'un certain nombre de couples attributs/valeurs.

Il a donc la forme suivante :

(KQML-performative

: sender <word> //l'émetteur du message

: receiver <word> //le destinataire du message

: language <word> //le langage d'expression du message

: ontology <word> //le vocabulaire du domaine

: content <expression>...) //le contenu du message

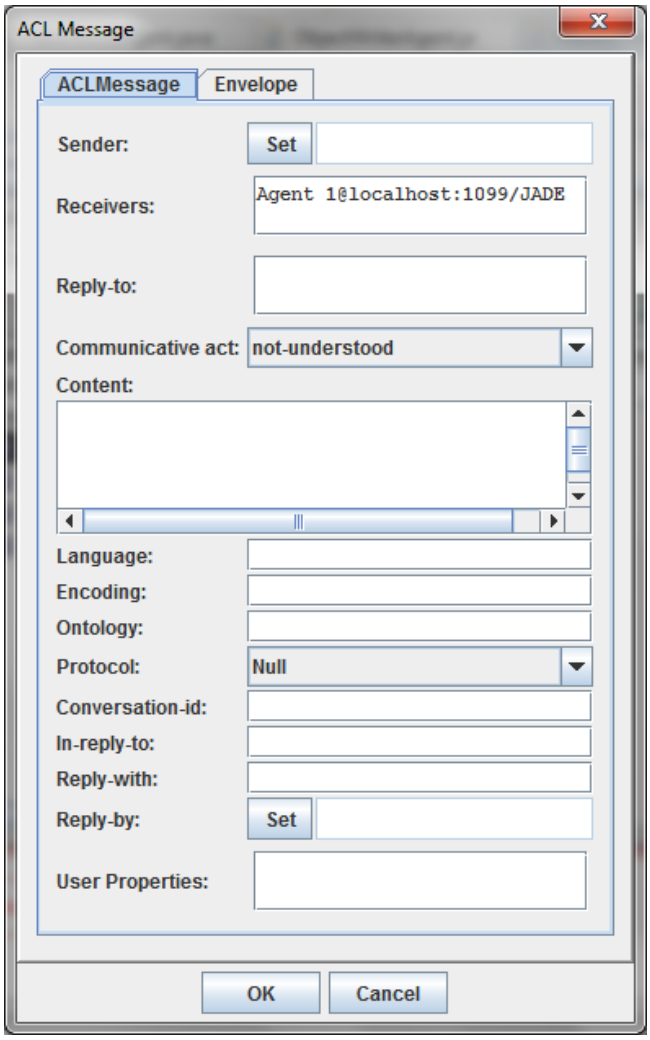


Figure 43: Message ACL pour agents

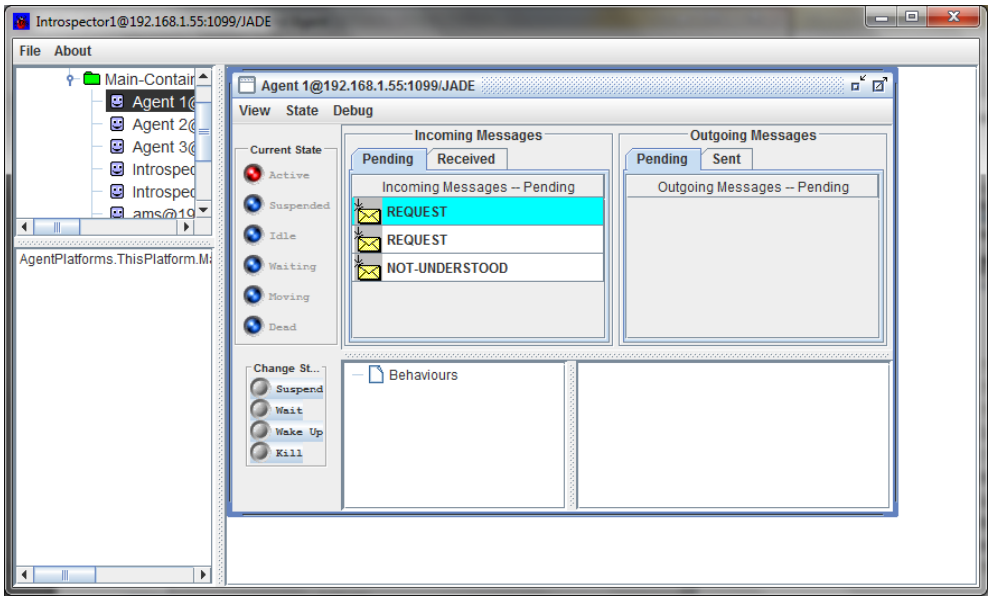


Figure 44: Outils d'Introspection d'Agents

Quand un utilisateur décide d'épier un agent ou un groupe d'agents, il utilise un agent sniffer. Chaque message partant ou allant vers ce groupe est capté et affiché sur l'interface du sniffer. L'utilisateur peut voir et enregistrer tous les messages, pour éventuellement les analyser plus tard. L'agent peut être lancé du menu du RMA ou de la ligne de commande suivante : Java jade.Boot sniffer :jade.tools.sniffer.sniffer.

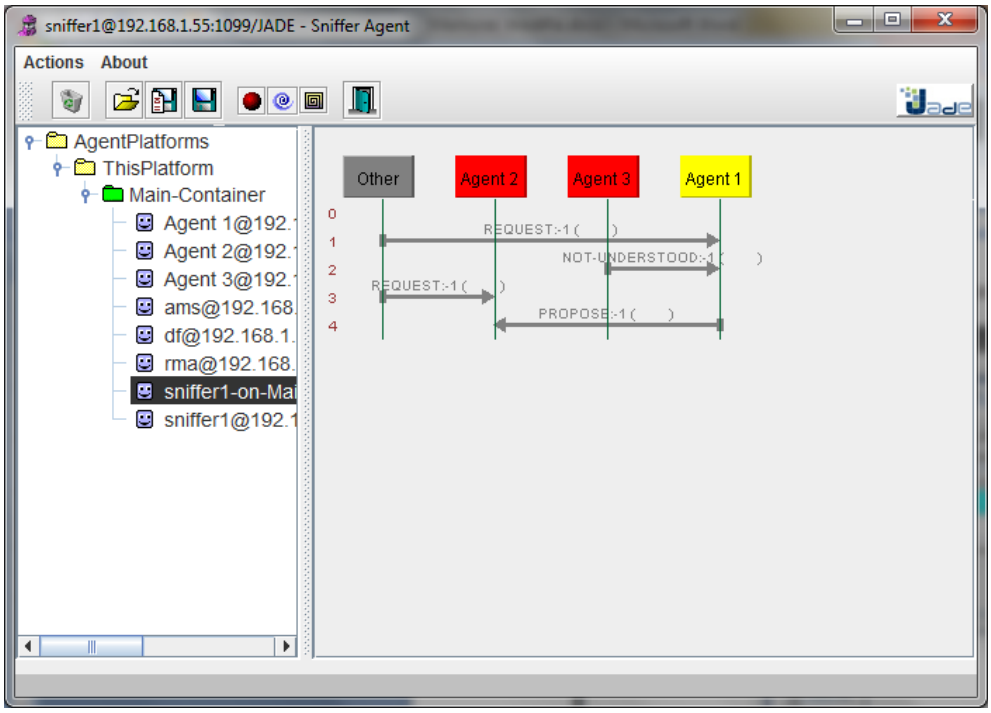


Figure 45: Outils « Sniffer » d'Agents

6.5 Conclusion

Nous avons présenté dans ce chapitre une étude de cas portant sur le processus de vote, un serveur et des clients avec des composants dispersés sur le réseau dédié, notre objectif était de donner une idée sur le fonctionnement du système par une un mécanisme de vote à travers ces composants de vote (un serveur et des clients). Dans cette perspective, cette étude de cas est réalisée pour permettre d'éluder les différents aspects de notre approche. Ensuite, nous avons utilisé les plateformes ECLIPSE et JADE pour comme plate forme de développement de composants du serveur de vote avec des composants légers qui font office de clients, dans le but d'effectuer des simulations concernant les interactions entre agents du système.

Conclusion générale

Dans ce mémoire nous avons essayé d'aboutir à un système qui permet la vérification de la validité fonctionnelle des assemblages de composants systèmes, il existe aussi la validité structurelle et celle de consistance qui ne sont pas traitées dans ce mémoire, à la validité fonctionnelle que nous avons tenter d'apporter une réponse à travers les formalismes : contrat et types de session, des définitions ont été proposées et des propositions ainsi qu'un algorithme pour le classement des assemblages proposés, par un système multi agents axé sur la mobilité favorisant ainsi une meilleure prise en charge du problème d'adaptabilité.

Dans ce contexte nous avons proposé l'utilisation des contrats de session qui représentent une combinaison des contrats et des types de session, utilisés initialement pour les systèmes client-serveur, nous avons adapté les contrats de session pour consolider une vérification comportementale d'assemblage de composants systèmes, sachant que les contrats donnent accès à une vérification comportementale. Les contrats sont moins contraignant que les types de session qui sont plus rigoureux, ceci laisse penser qu'ensemble les deux mécanismes permettront une fiabilité accrue, nous avons aussi proposé un système de calcul de distance entre deux systèmes de composants qui se base essentiellement sur les services offerts et services requis présenté par (les branchements et sélections) respectivement, ce système de distance peut classer plusieurs systèmes de compositions possibles offrant une aide précieuse aux développeurs dans le choix des assemblages à implémenter parmi plusieurs possibles éventuellement.

Cependant et pour des raisons d'indisponibilité d'une plateforme distribuée à base d'agents mobiles permettant de tester la validité de ces propositions de manière concrète, nous espérons pouvoir le faire dans un futur proche. Donc on a crée des composants qui servent de base pour intégrer le formalisme proposé (CS) avec une possibilité du calcul proposé de distance entre systèmes ainsi qu'une simulation de communication entre agents. Ceci nous a permet de voir comment ce système de calcul réagit avec des composants concrets.

En perspectives nous pensons qu'un raffinement de la méthode de classement est possible en utilisant plus de paramètres, par exemple les préférences du développeur et sa politique qui sera intégrée au système d'assemblage, à travers un package qui prendra en charge les configurations possibles. En plus nous pouvons affiner le choix des agents à utiliser et proposer un système plus personnalisé et personnalisable pour le développeur.

Bibliographie

(Abadi & Lamport, 1993) ABADI, M., & LAMPORT, L. (1993). *Composing Specifications.* ACM.

(ACCORD, 2002) ACCORD, P. (2002). *Assemblage de composants par contrats.* Paris: École nationale supérieure des télécommunications Paris.

(Alagar & Periyasamy, 1998) Alagar, V. S., & Periyasamy, K. (1998). *Specification of Software Systems* (Vol. 2). (S. L. York, Éd.) New York, Etats Unis: Springer.

(Alfaro & Henzinger, 2001) Alfaro, L. d., & Henzinger, T. A. (2001). Interface automata. *ACM*.

(Bahri, 2009) BAHRI, M. R. (2009). *Une approche intégrée Mobile-UML/Réseaux de Petri pour l'Analyse des systèmes distribués à base d'agents mobiles.* Constantine.

(Barbier, Cauvet, Rieu, Bennasri, & Souveyet, 2002) Barbier, F., Cauvet, C., Rieu, D., Bennasri, S., & Souveyet, C. (2002). Composants dans l'ingénierie des systèmes d'information : concepts clés et techniques de réutilisation. *Actes des deuxièmes assises nationales du GdR I3*.

(Bellifemine, Poggi, & Rimassa, 1999) Bellifemine, F., Poggi, A., & Rimassa, G. (1999). JADE – A FIPA-compliant agent framework.

(Bernardi & Hennessy, 2012) Bernardi, G., & Hennessy, M. (2012). Modelling session types using contracts. *ACM*, 1-6.

(Bernardi & Hennessy, Giovanni Tito Bernardi) Bernardi, G., & Hennessy, M. (s.d.). *Giovanni Tito Bernardi.* Consulté le avril 13, 2013, sur School of Computer Science & Statistics (SCSS): https://www.scss.tcd.ie/~bernargi/index.xhtml

(Bertrand Braunschweig, 2002) BERTRAND BRAUNSCHWEIG, B. (2002). *VERS LA SIMULATION NUMERIQUE PAR AGENTS APPRENANTS.* Paris.

(Beugnard, Jézéquel, Plouzeau, & Watkins, 1999) Beugnard, A., Jézéquel, J.-M., Plouzeau, N., & Watkins, D. (1999). Making Components Contract Aware. *IEEE*, 1.

(Boissier) Boissier, O. (s.d.). *JADE Environnement pour la programmation multi-agent.* Consulté le 2013, sur http://jade.tilab.com

(Booch, Rumbaugh, & Jacobson, 1998) Booch, G., Rumbaugh, J., & Jacobson, I. (1998). *The Unified Modeling Language - User Guide.* Massachusetts: Addison Wesley.

(Brada, 2001) Brada, P. (2001). Towards automated component compatibility assessment. *ECOOP* , 2.

(Briot & Seghrouchni, 2009) Briot, J.-P., & Seghrouchni, A. E. (2009). *Technologies des systèmes multi-agents et applications industrielles.*

(Brown, 2000) Brown, A. W. (2000). *Large-Scale, Component-Based Development.* Prentice-Hall.

(Bruneton, Coupaye, & Stefani, 2004) Bruneton, E., Coupaye, T., & Stefani, J. (2004). *The Fractal Component Model.* Consulté le fevrier 14, 2013, sur http://fractal.ow2.org/specification/: fractal.ow2.org/specification/fractal-specification.pdf

(Carbone, Honda, & Yoshida, 2007) Carbone, M., Honda, K., & Yoshida, N. (2007). Structured Communication-Centred Programming for Web Services.

(Carbone, Honda, & Yoshida, 2008) Carbone, M., Honda, K., & Yoshida, N. (2008). Structured Interactional Exceptions in Session Types. *ACM*.

(Comer, 1996) Comer, D. (1996). *TCP/IP Architecture, protocoles, applications.* Paris: InterÉditions.

(Cubat dit Cros, 2005) CUBAT DIT CROS, C. (2005). *Agents Mobiles Cooperants pour les Environnements Dynamiques.* Toulouse: l'Institut National Polytechnique de Toulouse.

(Cyril, 2003) Cyril, C. (2003). *Contrats Comportementaux pour Composants.* paris: École Nationale Supérieure des Télécommunications.

(Dezani-Ciancaglini, Yoshida, Ahern, & Drossopoulou, 2005) Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., & Drossopoulou, S. (2005). A distributed object-oriented language with session types. *Springer LNCS*, 1-3.

(Eriksson & Penker, 2000) Eriksson, H.-E., & Penker, M. (2000). *Business Modeling with UML: Business Patterns at Work.* Hoboken New_Jersey: John Wiley & Sons.

(Fabresse, 2007) Fabresse, L. (2007). *Du découplage à l'assemblage non-anticipé de composants Conception etmise en oeuvre du langage à composants SCL.* Memoire, Montpellier.

(Ferber, 1995) Ferber, J. (1995). *Les Systemes Multi Agents: vers une intelligence collective.* InterEditions.

(Fuggetta, Picco, & Vigna, 1998) FUGGETTA, A., PICCO, G. P., & VIGNA, G. (1998). Understanding Code Mobility. *IEEE*.

(Gao, Jacob Tsao, & Wu, 2003) Gao, J. Z., Jacob Tsao, H.-S., & Wu, Y. (2003). *Testing and Quality Assurance for Component-Based Software.* Boston/London: Artech House.

(Gay S. J., Vasconcelos, Ravara, Gesbert, & Caldeira, 2010) Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N., & Caldeira, A. Z. (2010). modular session types for distributed object-oriented programming. *ACM*, 1-2.

(Gay & Hole, 1999) Gay, S., & Hole, M. (1999). Types and Subtypes for Client-Server interactions. *citeseerx*, 2.

(Gay & Hole, 2003) Gay, S., & Hole, M. (2003). *Types and Subtypes for Correct Communication in Client-Server Systems.* Glasgow: University of Glasgow.

(Gay, Vasconcelos, & Ravara, 2003) Gay, S., Vasconcelos, V., & Ravara, A. (2003). Session Types for Inter-Process Communication. *citeseerx*, 1-2.

(Giachino, 2009) Giachino, E. (2009). *Session Types: Semantic Foundations and Object-Oriented Applications.* Paris.

(Grondin, Bouraqadi, & Vercouter, 2006) Grondin, G., Bouraqadi, N., & Vercouter, L. (2006). Assemblage Automatique de Composants pour la Construction d'Agents avec MADCAR. *Actes de la 2eme journée Multi-Agents et Composants*, (pp. 39-48). Nimes.

(Guttag, Horning, Garland, Jones, Modet, & Wing, 1993) Guttag, J., Horning, J., Garland, S., Jones, K., Modet, A., & Wing, J. (1993). *Larch: Languages and Tools for Formal Specification.* Pennsylvanie: Springer-Verlag.

(Hall, 1990) Hall, A. (1990). Seven Myths of Formal Methods. *IEEE Software*, 11-20.

(Harrison, Chess, & Kershenbaum, 1995) Harrison, C. G., Chess, D. M., & Kershenbaum, A. (1995). *Mobile Agents: Are they a good idea?* New York: IBM.

(Herzum & Sims, 1999) Herzum, P., & Sims, O. (1999). *Business Component Factory : A Comprehensive Overview of Component-Based Development for the Enterprise.* Wiley Publishing.

(Hoare, 1969) HOARE, C. (1969). An axiomatic basis for computer programming. *ACM*, 1-6.

(Hofmeister, 1993) Hofmeister, C. R. (1993). *Dynamic reconfiguration of distributed applications.* Maryland: University of Maryland.

HONDA, K., VASCONCELOS, V. T., & KUBO, M. (1998). language primitives and type discipline for structured communication-based programming. *ESOP'98*, 19.

Hu, R., Yoshida, N., & Honda, K. (2008). Session-based distributed programming in Java. *Springer LNCS*, 1-4.

(Hu, Yoshida, & Honda, 2008) Hu, R., Yoshida, N., & Honda, K. (2008). Session-Based Distributed Programming in Java. *ACM*.

(Huot, 2004) Huot, C. (2004). *Construction de collaborations entre composants.* Montpellier: Universié Montpellier II.

(Khayati, 2005) KHAYATI, O. (2005). *Modèles formels et outils génériques pour la gestion et la recherche de composants.* GRENOBLE: INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE.

(Laneve & Padovani, 2008) Laneve, C., & Padovani, L. (2008). The pairing of contracts and session types. *ACM*, 1-4.

(Legond-Aubry, 2005) LEGOND-AUBRY, F. (2005). *Un modèle d'assemblage de composants par Contrat et Programmation Orientée Aspect.* Paris: CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS.

(McIlroy D. , 1968) McIlroy, D. (1968). MASS PRODUCED SOFTWARE COMPONENTS. *In Proceedings of the 1st International Conference on Software Engineering*, (p. 79). Pattenkirchen.

(Meyer, 1992) Meyer, B. (1992). design by contract. *IEEE Computer*, 7.

(Meyer, 2000) Meyer, B. (2000, 03). *http://se.ethz.ch/~meyer/publications/.* Consulté le 12 2012, sur http://se.inf.ethz.ch/.

(Michel , 2004) Michel, F. (2004). *Formalisme, outils et ´el´ements m´ethodologiques pour la mod´elisation et la simulation multi-agents.* Montpellier: Université Montpellier II.

(OMG, 2002) OMG. (2002). *CORBA Components.* Lille: omg.

(Padovani, 2009) Padovani, L. (2009). Session Types at the Mirror. Dans D. G. Filippo Bonchi (Éd.), *Proceedings 2nd Interaction and Concurrency Experience*, (pp. 71-86). Bologna.

(Padovani, 2010) Padovani, L. (2010). Session Types = Intersection Types + Union Types. Dans B. V. Elaine Pimentel (Éd.), *Intersection Types and Related Systems*, (pp. 71-89). Edinburgh.

(Pessemier, 2007) Pessemier, N. (2007). *Unification des approches par aspects et a composants.* memoire, Lille.

(Projet ACCORD, 2003) Projet ACCORD. (2003). *Modèle abstrait d'assemblage de composants par contrats.* PARIS: Réseau National en Technologies Logicielles FRANCE.

(Ramadour, 2000) Ramadour, P. (2000). ReCoDe : un modèle pour la conception et l'utilisation de composants réutilisables en ingénierie des systèmes d'information. *INFORSID*.

(Ramadour, 2001) Ramadour, P. (2001). *Modèles et langage pour la conception et la manipulation de composants.* Marseille.

(Russell & Norvig, 2003) Russell, S., & Norvig, P. (2003). *Artificial Intelligence:A Modern Approach* (Vol. 2). New Jersey: Prentice Hall.

(Santhanam, Basu, & Honavar, 2011) Santhanam, G. R., Basu, S., & Honavar, V. (2011). Representing and Reasoning with Qualitative Preferences for Compositional Systems. *ACM*, 1-62.

(Saudrais, 2010) Saudrais, S. (2010). *Qualité de Service Temporelle pour Composants Logiciels.* RENNES: UNIVERSITÉ DE RENNES.

(Sommerville, 2007) Sommerville, I. (2007). *Software Engineering* (Vol. Eighth Edition). (Addison-Wesley, Éd.) Harlow, Essex, Angleterre: Addison-Wesley.

(Sycara, Widoff, Klusch, & Lu, 2002) Sycara, k., Widoff, s., Klusch, m., & Lu, J. (2002). LARKS: Dynamic Matchmaking among Heterogeneous Agents in Cyberspace. *Kluwer Academic Publishers*, 11-13.

(Szyperski, 1998) Szyperski, C. (1998). Component Software : Beyond Object-Oriented Programming. *ACM Press and Addison-Wesley, New York*(37).

(Szyperski, 2002) Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming.* Boston: Addison-Wesley.

(Tanenbaum, 1990) Tanenbaum, A. (1990). *RÉSEAUX : Architecture, protocoles, applications.* Paris: InterÉditions.

(Vallecillo, Vasconcelos, & Ravara, 2003) Vallecillo, A., Vasconcelos, V. T., & Ravara, A. (2003). Typing the Behavior of Objects andComponents using Session Types. *Elsevier Science*, 18.

(Vallecillo, Vasconcelos, & Ravara, 2005) Vallecillo, A., Vasconcelos, V. T., & Ravara, A. (2005). Typing the Behavior of Software Components using Session Types. *ACM*.

(Vasconcelos, Gay, & Ravara, 2006) Vasconcelos, V. T., Gay, S. J., & Ravara, A. (2006). Typechecking a Multithreaded Functional Language with Session Types. *citeseerx*, 2-6.

(Wegner & Zdonik, 1988) Wegner, P., & Zdonik, S. B. (1988). Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. (pringer-Verlag, Éd.) *ECOOP'88*, 55-77.

(Wooldridge & Jennings, 1995) Wooldridge, M., & Jennings, N. R. (1995). Intelligent agents : Theory and practice. *The Knowledge Engineering Review*, 115-152.

(Xu & Wims, 2000) Xu, C.-Z., & Wims, B. (2000). A Mobile Agent Based Push Methodology for Global Parallel Computing. 5-6.

(Zitouni, 2008) Zitouni, A. (2008). *Utilisation des design Patterns et des méthodes formelles dans le développement des systèmes d'information.* Constantine.

ANNEXE A

ADL : (Architecture Description Language) langue et / ou un modèle conceptuel pour décrire et représenter des architectures de système

B : Une méthode formelle de développement logiciel qui permet de modéliser de façon abstraite dans le langage de B le comportement d'un programme, puis par raffinements successifs.

CSP: (Communicating Sequential Processes) un langage informatique définit par Hoare en 1978

IDL : (Interface Description Language) appelé aussi (Interface Definition Language), abrégé en IDL, est un langage voué à la définition de l'interface de composants logiciels,

Larch: Langage de spécification algébrique des types de données abstraits, destinés à la spécification précise des systèmes informatiques, développé aux États-Unis dans les années 1980

LOTOS : (Language Of Temporal Ordering Specification) Un langage algébrique composé de: une partie pour la description des données et des opérations et une partie de la description de processus

NP –Complet : Un problème Non-déterministe Polynomial

Π-Calcul : Un langage de programmation théorique inventé par Robin Milner

RPC : (Remote Procedure Call) un protocole réseau permettant de faire des appels de procédures sur un ordinateur distant à l'aide d'un serveur d'applications

VDM: (Vienna Development Method) Un ensemble d'outils de développement informatique

Z: Un langage de spécification utilisé pour décrire et modéliser les systèmes informatiques, créé par J.R. Abrial

ANNEXE B

Publications personnelles

Use of agents for reasoning about component behavior

Bakhouche Abderraouf : Université de Khenchela

Siam Abderrahim : Université de Constantine

ICIST'2012 N° 142 Sousse Tunisie (24-26 Mars 2012)

Abstract In this paper we present an approach based on software components[1] for developing distributed, open and adaptable applications, Composition and its validity begins with the verification and validity of the behavior of individual components, then the system composed is validated recursively, Software components and multi-agent systems are design approaches and software development that currently has a large impact, it will involve agents to play roles of connectors between components as a support for the execution of the composition (assembly realization by exploiting the capabilities of dynamic cooperation, negotiation and interoperability of agents.

Use of mobile agents to ensure proper functioning of component-based applications

Siam Abderrahim : Université de Constantine

Maamri Ramdane : Labo LIRE of Constantine

Sahnoun Zaidi : Labo LIRE of Constantine

Bakhouche Abderraouf : Université de Khenchela

ICIST'2012 N° 143 Sousse Tunisie (24-26 Mars 2012)

Abstract In this paper we present an approach based on software components for developing distributed, open and adaptable applications, it will involve mobile agents to play roles of connectors between components as a support for the execution of the composition (assembly realization) on one side, and on the other side to verify the validity of compositions to achieve the functionality provided by the application.

The validity of the components' compositions is treated in three views. First a functional point of view of ensuring that the behavior results from a composition of software components will be apt with the expected functionality for a given service. Moreover, it will create an order of preference between several compositions that can achieve the same service according to their degree of satisfaction of the service for this we propose a fuzzy axiomatic semantics of a concurrent programming language to measure to what degree the behavior resulting from the expected composition will satisfy the expected functionality from the application. Second a structural point of view related to the dynamics of the network environment where the application runs, in this point we use mobile agents that we call Positioners to ensure structural validity criteria, finally a point of view of consistency of compositions to ensure that the signatures parts of the various components are consistent. This last point is not to be detailed in this paper.