REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE LARBI BEN M'HIDI – OUM EL BOUAGHI
FACULTE DES SCIENCES EXACTES ET DES SCIENCES DE LA NATURE ET DE LA VIE
Département DE MATHEMATIQUES ET INFORMATIQUE

N° D'ordre : ……
Série : ……

Thèse

Pour l'obtention du diplôme de

DOCTORAT EN INFORMATIQUE

Option: ***Ingénierie des Systèmes Distribués***

Thème

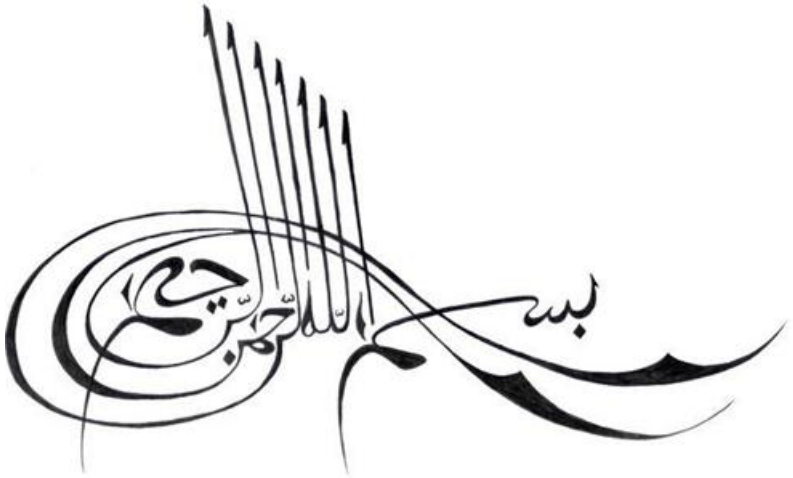
Test des Systèmes Multi-Agents Holoniques : Une Approche basée ASPECS

Présentée par **: DEHIMI Nour El Houda**

Devant le jury composé de :

Président :	**NINI Brahim**	**MCA**	**Univ. Oum El Bouaghi**
Directeur de thèse :	**MOKHATI Farid**	**Prof**	**Univ. Oum El Bouaghi**
Co-Directeur de thèse :	**BADRI Mourad**	**Prof**	**UQTR-Canada**
Examinateurs :	**LAFIFI Yacine**	**MCA**	**Univ. Guelma**
	MERAH ElKamel	**MCA**	**Univ. Khenchela**
	DERDOURI Lakhdar	**MCA**	**Univ. Oum El Bouaghi**

2015/2016


بسم الله الرحمن الرحيم

Remerciements

Mes remerciements, les plus vifs et les plus chaleureux, ainsi que toute ma reconnaissance, sont dédiés à mon encadreur Mr le Professeur, ***Farid MOKHATI*** *qui a dirigé et supervisé ce travail avec beaucoup d'expérience. Qu'il soit assuré de mon respect indéfectible et de toute ma reconnaissance pour l'aboutissement de ce travail dont la concrétisation a été possible grâce à son entière disponibilité et à son imperturbable patience.*

Mes remerciements vont aussi à mon co-encadreur, Mr le Professeur ***Mourad BADRI*** *pour les conseils qu'il m'a prodigués et la célérité avec laquelle il a toujours répondu à mes questions (malgré la distance). Puisse t-il trouver dans la concrétisation de ce travail, les résultats des efforts consentis et du temps précieux qu'il m'a accordé.*

Mes sincères remerciements au Dr ***Brahim NINI****, MCA à l'université d'Oum El Bouaghi, qui a accepté de présider le jury*

Mes remerciements vont aussi aux membres du jury composé de :

Dr. ***Yacine LAFIFI****, MCA à l'université de Guelma.*

Dr. ***Elkamel MERAH,*** *MCA à l'université de khenchela.*

Dr. ***Lakhdar DERDOURI****, MCA à l'université d'Oum El Bouaghi.*

Tout en les remerciant d'avoir accepté de juger notre travail. Leurs remarques et suggestions contribueront, très certainement à l'enrichissement et à l'amélioration de ce travail.

Je remercie également tous les membres du laboratoire ReLa(CS)² .

Je remercie tous ceux qui ont contribué de près ou de loin à l'élaboration de ce travail.

Dédicace

Je dédie ce travail à :

Ma mère et à mon père, puisse-t-ils y trouver, l'aboutissement de tous les investissements consentis

A tous les membres de ma famille

الملخص:

النظم المتعددة الوكلاء القائمة على المنظمة تتلقى بشكل متزايد المزيد من الاهتمام في مجال تطوير النظم المعقدة و توزيعها، تعتبر نظم متعددة الوكلاء الهلونيكية كمنظمة جد فعالة تسمح ببناء النظم المعقدة حيث تتسم بالكفاءة من حيث استخدام الموارد ، المرونة مع الاضطرابات و قابل يتي التكيف مع التغيرات في البيئة المحيطة بها .ومع ذلك، تطوير التطبيقات المستندة علي النظم المتعددة الوكلاء الهلونيكية ليست ناضجة بما فيه كفاية من حيث أنشطة الاختبار الضرورية جدا لضمان جودتها و هذا راجع لكونها ليست مغطاة بشكل كافي. في هذه الأطروحة نقترح طريقة جديدة للاختبار تقوم على النماذج خاصة بوكلاء الهولونيك ، تستخدم تقنية الخوارزميات الوراثية و تأخذ بعين الاعتبار التغيرات (النسخ المتعاقبة) لكل وكيل و تتم هذه الطريقة عبر مرحلتين أساسيتين تتمان بشكل متكرر و متعاقب. تركز المرحلة الأولى على إيجاد نسخة جديدة من الوكيل الذي تحت الاختبار لفحصها وتتناول المرحلة الثانية إجراء اختبار ع لى كل نسخة جديدة تم إيجادها. ويتم تحليل النسخة الجديدة من الوكيل لإيجاد النموذج السلوكي الذي نعتمد عليه في ايجاد حالات الاختبار. وتركز عملية ايجاد حالات الاختبار على الأجزاء الجديدة (و / أو المعدلة) لسلوك الوكيل. وبهذه الطريقة فإن التقنية تدعم التحديث في مرحلة إيجاد حالات الاختبار، و التي بدورها تمثل مشكلة. البرنامج الذي طورناه يدعم التقنية المقترحة من خلال تطبيقه على دراسة حالة ملموسة .

الكلمات الدالة :

نظم متعددة الوكلاء الهلونيكية ، الإختبار القائم على النماذج ، الخوارزميات الوراثية ،ضمان الجودة.

ABSTRACT

Organization-Based Multi-Agent Systems are receiving increasingly more attention in the field of complex and distributed systems development. Holonic Multi-Agent Systems (HMAS) are considered as well known organization allowing the construction of complex systems that are efficient in terms of use of resources, highly resilient to disturbances and adaptable to changes in their environment. However, HMAS-based applications development is not mature enough yet. Particularly, testing activities, which represent an important task in their quality assurance, are not well covered. This thesis aims at proposing a new model-based testing technique for holonic agents. The technique uses genetic algorithms and takes into account the evolution (successive versions) of an agent. The approach is organized in two main phases that are conducted iteratively. The first phase focuses on the detection of a new version of an agent under test. The second phase addresses the testing of each new detected version. The new version of the agent is analyzed in order to generate a behavioral model on which is based the generation of test cases. The test cases generation process focuses on the new (and/or changed) parts of the agent behavior. In this way, the technique supports an incremental update of the test cases, which is a crucial issue. A software tool that we developed supports the proposed technique. The approach and associated tool are illustrated using a concrete case study.

Keywords

Holonic Agent, Model-Based Testing, Genetic Algorithm, Quality Assurance.

RÉSUMÉ

Les systèmes multi-agents, basés organisation, s'imposent de plus en plus dans le domaine du développement de systèmes complexes et distribués. Les systèmes multi-agents Holoniques (SMAH) sont considérés comme étant une organisation bien établie permettant la construction des systèmes complexes d'une manière très efficace en termes d'utilisation des ressources, de la résistance aux perturbations et de l'adaptation aux changements de leur environnement. Le développement des applications basées sur les SMAH demeure peu solide. L'activité de test, qui représente une tâche importante dans l'assurance de leur qualité, n'est pas bien couverte. Dans cette thèse, nous proposons une nouvelle approche de test basée sur les modèles spécifiques aux agents holoniques. La technique utilise les algorithmes génétiques et prend en compte l'évolution (versions successives) d'un agent. Elle est organisée en deux principaux processus qui s'exécutent d'une façon itérative. Le premier processus porte sur la détection d'une nouvelle version de l'agent à tester. Le deuxième processus porte sur le test de chaque nouvelle version détectée. La nouvelle version de l'agent est analysée afin de générer un modèle comportemental sur lequel est basée la génération des cas de test. Le processus de génération des cas de test se concentre sur les nouvelles (et / ou modifiées) parties du comportement de l'agent. De cette manière, la technique prend en charge une mise à jour incrémentielle des cas de test, qui est un problème crucial. Un outil logiciel que nous avons développé prend en charge la technique proposée. L'approche et l'outil associé sont illustrés à l'aide d'une étude de cas concrète.

Mots-clés

Agent Holonique, Test Basé Modèle, Algorithme Génétique, Assurance Qualité.

Table des matières

Table des figures

Introduction générale

1. Contexte général de la Problématique

Les systèmes multi-agents sont aujourd'hui largement acceptés comme une métaphore efficiente et un paradigme à part entière de modélisation des systèmes complexes dans plusieurs secteurs. Ils sont très efficaces pour gérer la nature hétérogène des composantes d'un système, pour modéliser les interactions entre ses composantes et pour tenter de comprendre les phénomènes émergents qui en découlent. Toutefois, dans les systèmes multi-agents non holoniques, la vision hiérarchique et la structure pyramidale et récursive de la plupart des systèmes complexes est encore relativement peu prise en compte. Les systèmes multi-agents holoniques (SMAH) peuvent offrir une alternative intéressante à ce problème. L'idée sous-jacente à ce type de systèmes considère que les agents peuvent se regrouper pour créer un agent de niveau supérieur ou qu'un agent peut être décomposé en plusieurs agents de niveau inférieur. De ce fait, le paradigme des Systèmes Multi-Agents Holoniques est une approche adéquate pour la conception et la mise en œuvre des systèmes complexes, émergents et adaptatifs, où la flexibilité, la reconfigurabilité et l'adaptation au changement de l'environnement jouent un rôle crucial [Pau09]. Les SMAH sont, en fait, plus appropriés pour les environnements ayant de nombreux agents et une complexité élevée. Les SMAH permettent, en particulier, la réduction de la complexité des systèmes complexes.

Le modèle holonique a été utilisé, dans les années 90, dans le domaine des systèmes manufacturiers comme un nouveau paradigme de modélisation. Il a, depuis, fait l'objet de beaucoup d'attention. De nombreux modèles sont apparus, tels que PROSA (Product-Resource-Order-Staff Architecture) [Van98], HCBA (Holonic-Component-Based-

Architecture) [Jin00], ADACOR (ADAptive holonic COntrol aRchitecture for distributed manufacturing systems) [Lei02]. Toutefois, la vision holonique dans les systèmes manufacturiers diffère de celle utilisée dans les SMA [Adr04], [Sch04]. Fisher et al. [Fis03] identifient d'ailleurs des différences très claires entre ces deux visions. En effet, les systèmes holoniques manufacturiers ne font aucune pré-supposition sur les caractéristiques et les propriétés internes d'un holon. La seule exigence est que chaque holon agisse en tant qu'unité de contrôle contrairement aux SMAH, où un holon se doit de posséder les propriétés classiquement assignées aux agents telles que l'autonomie, l'habilité sociale, la réactivité et la proactivité [Mic95]. La décomposition hiérarchique dans les SMAH est faite en fonction de la complexité des tâches, ce qui n'est pas le cas dans les systèmes holoniques manufacturiers où un holon est généralement composé d'une entité physique (unité de production) et d'un contrôleur logiciel associé à cette entité. Malgré ces différences entre la vision des holons adoptée dans le SMA et celle adoptée dans les systèmes manufacturiers, de nombreux modèles proposés dans ce dernier domaine peuvent être adaptés aux systèmes multi-agents holoniques.

La référence actuelle de la notion de holon dans les systèmes multi-agents est incarnée par l'introduction de Gerber et al. [Ger99], où la vision des holons correspond davantage à la notion d'agent récursif ou composé. De nombreuses approches ont tenté de modéliser cette idée d'agent récursif, entre autres, MORISMA [Cor01], SWARM [Bur97], SOHTCO [Ada00b]. ANEMONA [Bot08] est la première méthodologie multi-agents, au sens entendu dans l'ingénierie logicielle, qui fut proposée pour les systèmes holoniques. Elle est spécialisée dans le domaine des systèmes holoniques manufacturiers. ANEMONA est inspirée de la méthodologie INGENIAS [Pav05] et adopte les mêmes perspectives sur le système. Cette méthodologie réutilise les différentes architectures de holon proposées dans le modèle PROSA. Bien qu'organisationnelle, car basée sur INGENIAS, ANEMONA adopte néanmoins, une approche centrée sur l'agent et son architecture, à la fin du processus de conception. De plus, elle introduit un ensemble de notations spécifiques pour assister le processus de modélisation et représenter les concepts de rôle, d'agent, de but, de groupe, etc., au lieu d'utiliser les standards existants.

Rodriguez [Rod05] a ensuite adapté cette vision des holons, avant tout centrée sur l'agent, vers une vision organisationnelle qui offre plusieurs avantages [Fer04], parmi lesquels la modularité et la réutilisation, en proposant un "framework" organisationnel générique pour décrire la notion de holon, qui constitue l'une des bases de la conception du méta-modèle

CRIO [Mas07]. Le méta-modèle CRIO fournit les abstractions nécessaires pour décrire la décomposition hiérarchique d'un système complexe en un nombre quelconque de niveaux d'abstraction. Il est doté d'un processus méthodologique nommé ASPECS [Mas09]. Le processus ASPECS peut également être considéré comme une extension des méthodologies PASSI [Cos04] et AgilePASSI [Che04]. Il fournit un guide complet, depuis l'analyse des besoins jusqu'à l'implantation et le déploiement, permettant la modélisation d'un système à différents niveaux de détails, en procédant par raffinements successifs. Ce processus offre plusieurs avantages. En effet, il supporte l'approche organisationnelle de l'analyse à l'implantation. Par ailleurs, il utilise le standard de notation UML et AUML, ce qui augmente la possibilité de pouvoir transférer les technologies multi-agents dans le milieu industriel. Il encourage la modularité et la réutilisation des modèles et des connaissances grâce à la définition du comportement des rôles, sur la base des capacités, ce qui lui permet d'augmenter la généricité des organisations et ainsi favoriser leur réutilisation. Il utilise les ontologies comme base des connaissances communes et transversales au processus de modélisation. Ceci lui permet de regrouper, réutiliser et classifier les connaissances du domaine du problème. Il est doté d'une plateforme nommée Janus [Nic08], qui fournit une implantation directe des quatre concepts à la base de CRIO (Capacité, Rôle, Interaction et Organisation), ce qui contribue à réduire l'écart entre la conception et l'implantation.

Les systèmes holoniques ont été appliqués dans de nombreux autres domaines tels que la santé [Uli02], les systèmes de transport [Bür98], les robots footballeurs [Bar01] ou encore le travail collaboratif [Ada00a]. Malgré l'intérêt de la recherche pour les systèmes multi agents holoniques, le développement des applications basées sur ces systèmes demeure peu solide. L'activité de test, qui représente une tâche importante dans l'assurance de leur qualité, n'est pas bien couverte. En effet, il n'existe, dans la littérature, que très peu de propositions concernant le test de ces systèmes. Ceci laisse en suspens plusieurs questions sans réponse, par exemple : *« comment peut-on tester l'évolution des systèmes multi-agents, afin d'assurer qu'elle réponde à leurs spécifications fonctionnelles ? »*, demeure, encore, un objectif de recherche. Le test unitaire des agent consiste à tester les fonctionnalités internes d'un agent et celles que ce dernier met à la disposition des autres agents, en prenant en considération ses propres objectifs. Le test unitaire peut assurer que les agents opèrent correctement s'ils sont exécutés séparément, mais il ne peut pas détecter des défauts qui pourraient être crées si les agents sont mis en commun. Le test du niveau système assure, quant à lui, que tous les agents, dans le système, opèrent selon les spécifications et

interagissent correctement. Compte tenu du fait que le comportement des agents se développe avec le temps dans le but de répondre à leurs objectifs et leurs besoins et de s'adapter aux changements de l'environnement, le test de niveau système doit aussi assurer que le comportement acquis avec le temps est correct et répond aux spécifications du système. Ceci représente le point focal de notre approche.

Parmi les techniques de test on trouve la technique de test basée modèle Model-Based Testing (MBT) [Apf97], [Elf01], qui consiste à générer, à partir du modèle comportemental du système, des cas de test avec lesquels on exécute le système ; puis on compare la réponse du système à celle issue du modèle (réponse attendue). Cette technique offre plusieurs avantages puisqu'elle permet de maitriser le niveau d'abstraction et de spécifier les comportements attendus du système sous test. Elle possède, par ailleurs, la capacité de produire une série de cas de test de manière automatique, et facilite la comparaison entre les résultats obtenus à l'exécution du système avec les résultats attendus. Les avantages de cette technique nous incitent à l'appliquer pour tester le comportement de l'agent holonique qui est défini essentiellement par les interactions de ses membres. Cependant, deux problèmes se posent : (i) l'évolution imprévisible du comportement et de la structure de l'agent holonique dans le temps suite à l'introduction (à tout moment) de nouvelles interactions entre les membres de l'agent holonique (qui seront déterminés par la suite). En effet, si on applique le test sur la première version de l'agent holonique (avant l'évolution de son comportement et de sa structure) seuls les comportements initiaux seront testés, et un manque de confiance vis-à-vis des nouveaux comportements acquis avec le temps subsistera et nous imposera l'obligation de les tester. Par conséquent, il est nécessaire et important de maintenir les cas de test générés initialement par l'intégration de nouveaux cas de test afin de couvrir les nouveaux comportements (et structures) de l'agent holonique. (ii) la nature hiérarchique de l'agent holonique sachant que le comportement d'un super holon résulte des interactions de ses membres du niveau inférieur. Donc, si un développement s'effectue à un niveau, il affecte automatiquement le niveau supérieur. Dans ce cas, le test sera basé sur un modèle comportemental hiérarchique, ce qui complique la phase de génération des cas de test.

2. Objectifs et Contributions

Dans la littérature, il existe quelques travaux sur le test des SMA. Nous citons, entre autres, ([Zha09], [Eki08]) pour le test du niveau unitaire, ([Lam05], [Nun05], [Coe06], [Ngu08], [Ngu12]) pour le test du niveau agent, [Ser08] pour le test du niveau d'intégration et [Ngu10] pour le test multi-niveau. Bien que ces approches aient apporté des éléments de réponse importants à différents problèmes relatifs aux tests des SMA, ils ne tiennent pas en compte des spécificités des SMA holoniques. Ces approches sont, principalement, conçues pour les agents qui sont définis comme des entités atomiques alors que les agents holoniques sont, en fait, définis comme agents composés d'agents. En outre, elles ne sont pas appropriées pour couvrir le problème lié à l'évolution du comportement et de la structure des agents holoniques qui émergent avec le temps, suite à l'introduction (à tout moment) de nouvelles interactions entre les membres de l'agent holonique, et celle de la nature hiérarchique de l'agent holonique qui complique la génération des cas de test. Dans cette thèse, nous proposons une nouvelle approche de test basée sur les modèles pour les agents holoniques développés en utilisant le processus ASPECS en raison de son importance et de ses avantages. L'approche considère l'évolution du comportement et de la structure des agents holoniques et prend en compte leur nature hiérarchique. L'approche utilise les algorithmes génétiques pour l'évaluation et le suivi du développement de l'agent holonique. Elle est supportée par un outil que nous avons développé. L'approche proposée et l'outil associé sont illustrés à l'aide d'une étude de cas concrète.

3. Plan de la thèse

La présente thèse est organisée en cinq chapitres présentés comme suit :

- **Systèmes multi-agent holoniques** : dans ce chapitre sont présentés le méta-modèle CRIO, le processus ASPECS, la plate-forme JANUS, ainsi que la définition et les caractéristiques des agents holoniques.

- **Test et systèmes multi agents** : dans ce chapitre sont décrites les définitions et les différentes méthodes de test ainsi que la différence entre le test des agents et le test classique. Une synthèse bibliographique sur les travaux de test des agents est donnée dans ce chapitre.

- **Développement des agents holoniques** : dans ce chapitre sont étudiés l'évolution du comportement et de la structure de l'agent holonique et l'effet de cette évolution sur le processus de test.

- **Une nouvelle approche de test pour les agents holoniques**: dans ce chapitre est décrite l'approche de test proposée qui prend en considération l'évolution du comportement et de la structure de l'agent holonique ainsi que sa nature hiérarchique.

- **Étude de cas : Système de Distribution de Marchandises**: dans ce chapitre, notre approche est appliquée sur un cas concret dans le but de montrer la faisabilité de l'approche et ses bénéfices.

Chapitre 1

Systèmes Multi-Agents Holoniques

1. Introduction

Les systèmes multi-agents revêtent de plus en plus d'importance pour leur capacité à aborder les systèmes complexes. Ils sont très efficaces pour gérer la nature hétérogène des composantes d'un système, pour modéliser les interactions entre ses composantes et pour tenter de comprendre les phénomènes émergents qui en découlent. Néanmoins, la plupart des modèles proposés considèrent les agents comme des entités atomiques [Adr03], ce qui rend la vision hiérarchique et la structure pyramidale et récursive de la plupart des systèmes complexes relativement peu prise en compte. Pour pallier à cette faiblesse, l'utilisation des systèmes multi-agents holoniques (SMAH) trouve toute son application puisque ces derniers considèrent que les agents peuvent se regrouper pour créer un agent de niveau supérieur ou qu'un agent peut être décomposé en plusieurs agents de niveau inférieur. Ce qui rend les SMAH intrinsèquement récursifs et capables de décrire naturellement des systèmes de nature hiérarchique.

De nombreuses approches ont tenté de modéliser l'agent holonique, entre autres, MORISMA [Cor01], SWARM [Bur97], SOHTCO [Ada00b]. Mais, ces approches ont toujours été centrées agent ce qui a engendré leur faiblesse. Rodriguez [Rod05] a ensuite adapté cette vision des holons, qui était centrée agent, vers une vision organisationnelle qui offre plusieurs avantages [Fer04], dont, entre autres, la modularité et la réutilisation. Les

travaux de Rodriguez [Rod05] ont été à l'origine de la conception du méta-modèle CRIO [Mas07], qui fournit les abstractions nécessaires pour décrire la décomposition hiérarchique d'un système complexe en un nombre quelconque de niveaux d'abstraction. Ce méta-modèle a été doté d'un processus méthodologique nommé ASPECS [Mas09] qui fournit un guide complet, depuis l'analyse des besoins jusqu'à l'implantation et le déploiement, permettant la modélisation d'un système à différents niveaux de détail, en procédant par raffinements successifs tout en supportant l'approche organisationnelle de l'analyse à l'implantation. Ce processus est doté d'une plateforme nommée Janus [Nic08], qui fournit une implantation directe des quatre concepts à la base de CRIO (capacité, rôle, organisation et holon), ce qui contribue à réduire l'écart entre la conception et l'implantation.

Dans ce chapitre, nous définirons l'agent et les systèmes multi agent. Nous développerons, par la suite, les systèmes multi agents holoniques. Cependant, pour une meilleure compréhension des systèmes multi-agents holoniques, nous décrirons au préalable le méta-modèle CRIO, le processus méthodologique ASPECS ainsi que la plateforme Janus.

2. Systèmes multi-agents

2.1. Définition d'agent

Selon [Fer95] l'agent est une entité physique ou virtuelle :

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

2.2. Définition d'un système multi-agents (SMA)

Un SMA est une société organisée d'agents dans laquelle un certain nombre de phénomènes peuvent émerger comme la résultante des interactions entre les agents. Selon [Fer95], un Système Multi-Agents est un système composé des éléments suivants :

- Un environnement E, c'est-à-dire un espace disposant généralement d'une métrique.
- 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.
- Un ensemble A d'agents qui sont des objets particuliers ($A \subseteq O$), lesquels représentent les entités actives du système.
- Un ensemble de relations R qui unissent des objets (et donc des agents) entre eux.
- Un ensemble d'opérations Op permettant aux agents de A de percevoir, produire, consommer, transformer, et manipuler des objets de O.
- 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.

3. Système multi-agent holonique (SMAH)

3.1. Le méta-modèle CRIO

CRIO est un méta-modèle organisationnel destiné à la modélisation des systèmes complexes ouverts et de grande échelle. Le méta-modèle CRIO est issu de l'intégration et l'extension de deux méta-modèles existants. Le premier étant "RIO" dont le nom provient des trois principaux concepts sur lesquels il repose, à savoir : Rôle, Interaction, Organisation. Il a été proposé par [Hil00a] et a été introduit pour la modélisation organisationnelle des systèmes multi-agents non hiérarchiques. Le second étant le "framework" générique pour la modélisation de systèmes multi-agents holoniques, proposé par [Rod05]. De ce fait, CRIO précise et redéfinit certains des concepts précédemment introduits dans RIO et leur ajoute celui de Capacité. Il intègre ensuite les concepts holoniques et s'intègre dans le processus de développement logiciel ASPECS qui sera décrit par la suite. CRIO s'inspire de l'approche de

développement dirigée par modèles (ou MDD). Dans la logique de l'approche MDD, CRIO offre trois niveaux de modèles, chacun d'eux est qualifié de domaine:

- **Le domaine du problème (CIM)** : fournit la description organisationnelle du problème indépendamment d'une solution spécifique. Les concepts introduits dans ce domaine seront principalement utilisés durant la phase d'analyse et au début de la phase de conception du processus de développement.
- **Le domaine agent (PIM)** : introduit les concepts multi-agents et fournit une description d'une solution multi-agents, éventuellement holonique, basée sur les éléments du domaine du problème. Le domaine agent est davantage associé à la fin de la phase de conception.
- **Le domaine de la solution (PSM)** : il est relatif à l'implantation de la solution sur une plate- forme spécifique. Cet aspect est donc dépendant d'une plate-forme de déploiement particulière. Dans le cas présent, cette phase repose sur la plate-forme Janus qui fut spécifiquement développée pour faciliter l'implantation de modèles organisationnels et holoniques.

3.1.1. Le domaine du problème

Le domaine du problème fournit une description organisationnelle du problème indépendamment d'une solution spécifique. La figure 1.1 présente le diagramme UML de la partie du méta-modèle CRIO consacrée à la modélisation d'un problème.

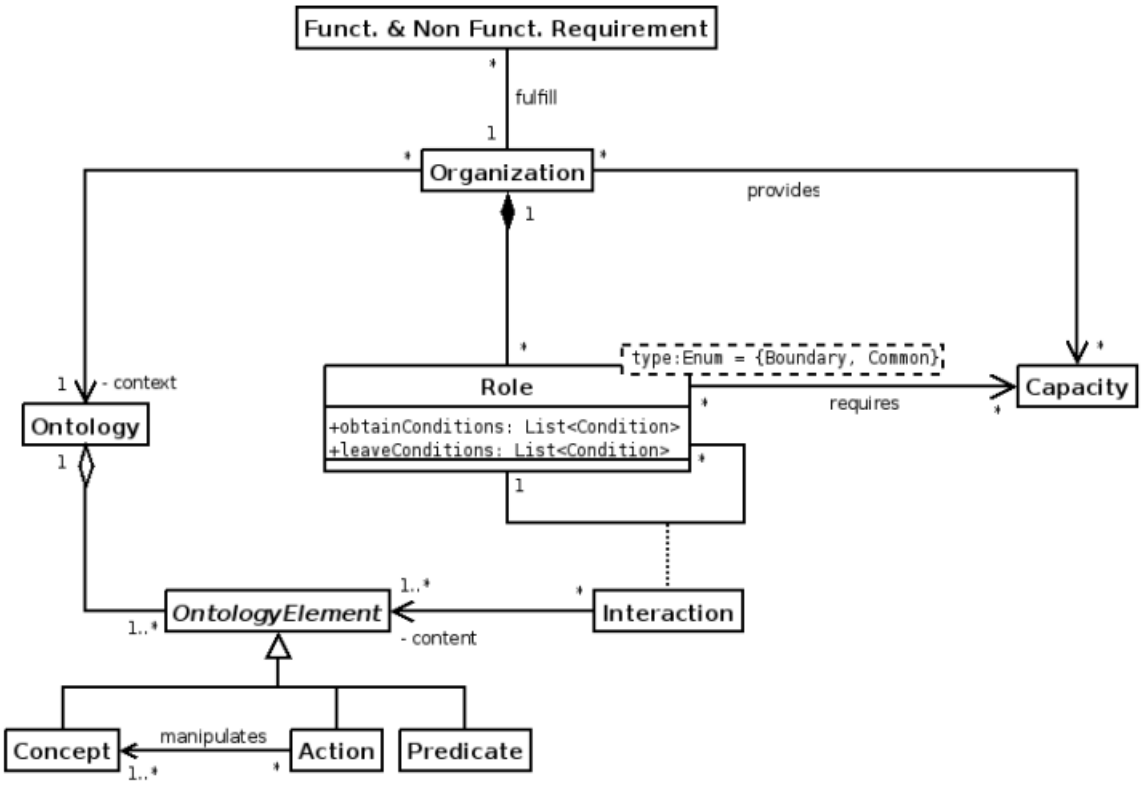


Figure 1.1 : Diagramme UML du domaine du problème du méta-modèle CRIO [Mas07].

Les concepts introduits dans le domaine du problème seront principalement utilisés durant la phase d'analyse et au début de la phase de conception du processus de développement. Ces concepts sont: ontologie, organisation, rôle, interaction et capacité.

- **Ontologie** : Une ontologie [Gru92] est un ensemble structuré de concepts visant à décrire un ou plusieurs domaines d'étude. Dans le méta-modèle CRIO, l'ontologie possède une double fonction : (i) l'ontologie permet, tout d'abord, de rassembler et d'organiser l'ensemble des connaissances disponibles sur le problème et sur son domaine, et de définir le contexte des organisations utilisées pour les modéliser.(ii) l'ontologie constitue, par la suite, dans le domaine agent, la base de connaissances nécessaire à la définition des communications entre les agents du fait qu'elle regroupe l'ensemble des connaissances qui seront échangées au cours des interactions entre les rôles qui composent les organisations du système.
- **Organisation** : Une organisation [Hil00b], [Rod05] est définie par un ensemble de rôles et des interactions entre ces rôles dans un contexte commun définissant ainsi un schéma d'interaction spécifique. Le contexte d'une organisation est défini par tout ou partie de l'ontologie du domaine associée aux besoins qu'elle est en charge de

satisfaire. Chaque organisation est associée à au moins un besoin fonctionnel qu'elle a pour objectif de satisfaire. Chaque besoin fonctionnel, associé à l'organisation devra être satisfait soit par le comportement individuel de l'un des rôles de l'organisation, soit par le comportement global émergent des interactions de tout ou partie des rôles. L'organisation étant à la fois le support et la manière de satisfaire un besoin. Elle peut être considérée comme la description d'un comportement global auquel devront se plier une ou plusieurs entités, afin de satisfaire les objectifs qui leurs sont attribués. L'organisation est un concept intrinsèquement récursif puisque son comportement peut être décomposé fonctionnellement en un ensemble de comportements de complexité inférieure qui interagissent, pour satisfaire les objectifs associés à l'organisation. Selon le niveau d'abstraction considéré, une organisation peut être vue soit comme un comportement unitaire soit comme un ensemble de comportements en interaction.

- **Rôle** : Un rôle [Hil00b], [Rod05] est l'abstraction d'un comportement dans le contexte d'une organisation. Il est défini dans une et une seule organisation. Il peut interagir avec les autres rôles définis par la même organisation. L'objectif d'un rôle est de contribuer pour toute ou partie aux besoins associés à l'organisation dans laquelle il est défini. Dans le contexte de son organisation, un rôle inclue un comportement et un statut. (i) Le comportement d'un rôle fixe les responsabilités qui sont associées au rôle et la méthode pour les satisfaire. Pour définir ce comportement, chaque rôle dispose d'attributs qui lui sont propres. Le comportement d'un rôle est spécifié par un plan comportemental (ou plan de comportement) qui peut être représenté par un diagramme d'activité UML ou un diagramme état-transition ("state-chart"). Ce plan décrit comment un ou plusieurs objectifs d'un rôle peuvent être satisfaits par son comportement. Il détaille comment combiner et ordonner les interactions du rôle avec les autres rôles, les événements extérieurs au rôle avec les tâches qui composent le comportement du rôle, (ii) Le statut définit la position du rôle au sein de son organisation ainsi qu'un ensemble de droits et d'obligations pour l'agent qui le joue. Le statut défini, également, l'interface au travers de laquelle l'agent jouant le rôle est perçu par les autres entités de la même organisation. Il fournit à cet agent le droit d'exercer ses capacités (compétences) dans le contexte de l'organisation et l'obligation de respecter le comportement décrit par le rôle. Le statut d'un rôle est

caractérisé par au moins un concept de l'ontologie définissant le contexte de son organisation.

- **Interaction**: Une interaction [Hil00b], [Rod05] entre rôles est composée de l'événement généré par un premier rôle et perçu par les autres rôles, ainsi que les réactions induites dans ces derniers. Les rôles qui interagissent partagent automatiquement un contexte d'interaction commun. La notion d'interaction est fondamentale puisqu'elle permet à un groupe de rôles d'accomplir une tâche plus complexe associée à leur organisation. Les interactions survenant entre les rôles au sein d'une organisation sont décrites dans un scénario d'interaction qui est généralement représenté par un diagramme de séquence UML. Un scénario d'interaction décrit comment un ensemble de rôles interagissent et se coordonnent pour satisfaire un objectif commun, lequel peut lui même être associé à un rôle de niveau d'abstraction supérieur. Cette association implique de pouvoir faire transiter de l'information entre deux niveaux d'abstraction adjacents.
- **Capacité** : Une capacité [Rod07] est la description abstraite d'un savoir-faire. Elle regroupe la description de moyens qui permettent l'accomplissement d'une tâche. Ceci incarne une fonctionnalité logique aux yeux des entités qui la fournissent (les propriétaires) et de celles qui l'utilisent ou la requièrent (les utilisateurs). Les propriétaires de capacités peuvent être des agents, des holons, des rôles ou des organisations. Les utilisateurs sont généralement des rôles ou des organisations. La capacité possède une double fonction dans le méta-modèle CRIO : (i) Elle constitue tout d'abord une interface entre l'agent et le rôle. Pour définir son comportement, le rôle requiert certaines compétences modélisées par des capacités et, pour qu'un agent obtienne un rôle, il doit acquérir toutes les capacités requises par ce rôle en fournissant une réalisation concrète (ou implantation) à chacune des capacités que le rôle requiert. (ii) La capacité permet également, dans le processus de modélisation, d'effectuer l'interface entre deux niveaux d'abstraction adjacents dans la hiérarchie organisationnelle du système.

3.1.2. Le domaine agent

Le domaine du problème de CRIO permet la modélisation du problème en termes d'organisations, de rôles, de capacités et d'interactions. Le résultat de cette modélisation doit aboutir à la définition d'une hiérarchie d'organisations combinant leurs comportements respectifs pour satisfaire les besoins identifiés. A partir du modèle du problème, l'objectif du

domaine d'agent est de concevoir un modèle de solution multi-agents. Il s'agit d'élaborer le modèle d'une société d'agents (et/ou de holons) capable d'offrir une solution au problème étudié, en décrivant les interactions entre agents ainsi que leurs éventuelles dépendances. La figure 1.2 présente le diagramme UML du domaine agent du méta-modèle CRIO.

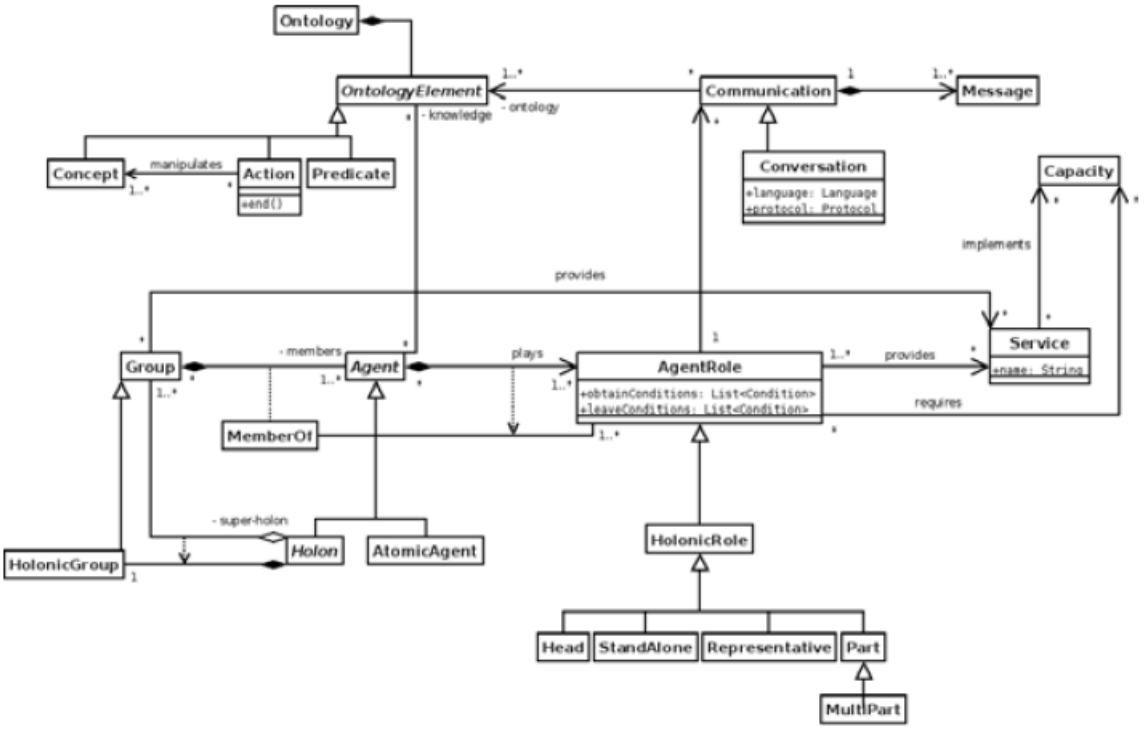


Figure 1.2 : Diagramme UML du domaine Agent du méta-modèle CRIO [Mas07].

Le domaine agent introduit les concepts suivants : Groupe, rôle d'agent, Communication, Agent, Holon.

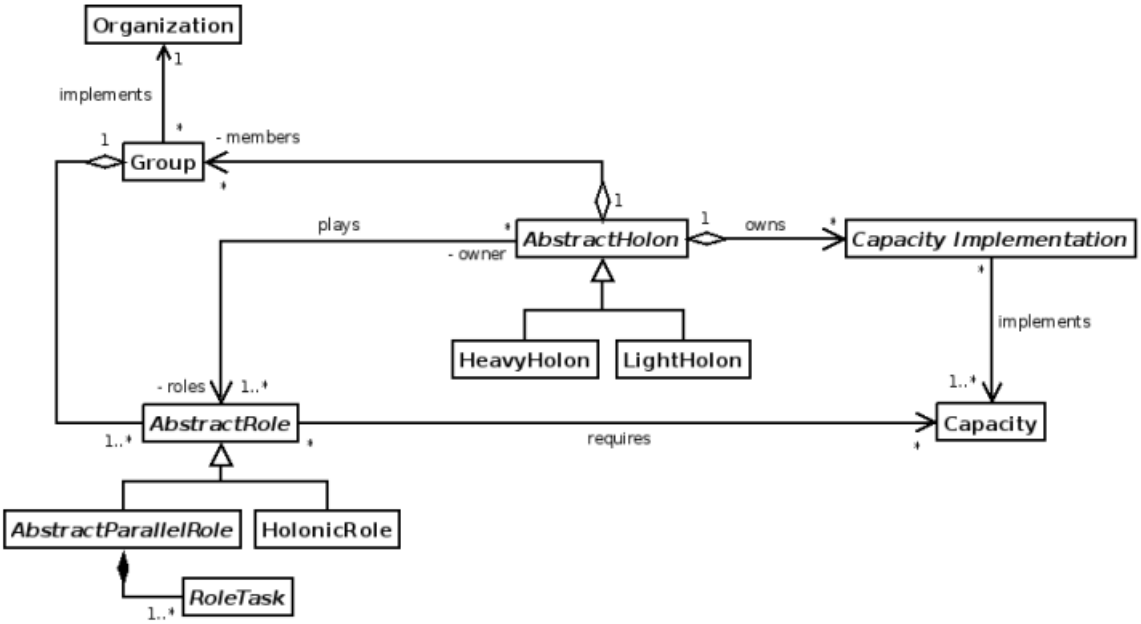
- **Groupe** : Au sein du domaine agent, les organisations issues du modèle du problème sont instanciées sous forme de groupes. Un groupe est une instance concrète d'une organisation. Les agents qui jouent des rôles dans le même groupe interagissent et coopèrent pour satisfaire un ou plusieurs objectifs associés à ce groupe tout en respectant le comportement du rôle qu'ils y jouent. Le comportement global du groupe doit suivre le schéma spécifique d'interaction décrit par l'organisation qu'il instancie.
- **Rôle d'agent** : Les rôles qui composaient ces organisations sont eux-aussi instanciés sous forme de rôles d'agent. Un rôle d'agent est une instance concrète d'un rôle. Il décrit un comportement dans le contexte défini par un groupe. Il confère à l'agent un statut dans ce groupe et les moyens d'interagir avec les autres agents jouant des rôles

au sein du même groupe. Pour qu'un agent obtienne un rôle, il doit acquérir toutes les capacités nécessaires requises par ce dernier.

- **Communication** : Les interactions sont spécialisées en communication dans laquelle les connaissances partagées par les participants sont représentées par un ensemble d'éléments de l'ontologie.
- **Agent** : Un agent est une entité jouant un ensemble de rôles qui peuvent être définis dans plusieurs groupes. L'agent est caractérisé par ses rôles, ses connaissances et ses capacités. Le comportement global d'un agent résulte de la combinaison des rôles, des connaissances et des capacités dont il dispose à un instant donné. Chaque agent doit satisfaire les objectifs qui ont été attribués aux rôles qu'il joue, et qui sont spécifiés dans leur comportement.
- **Holon** : Un holon est une entité auto-similaire composée de holons comme sous-structures. La structure hiérarchique composée de holons est appelée holarchie. Un holon peut être vu, en fonction du niveau d'observation, à la fois comme une entité atomique, et comme un groupe de holons en interaction. De la même manière, un ensemble constitué de différents holons peut être considéré comme un ensemble d'entités en interaction ou comme des parties d'un holon de niveau supérieur. À un niveau d'observation donné, le holon composé est qualifié de super-holon. Les holons qui composent un super-holon sont appelés sous-holons ou holons membres. Une description plus détaillée des agents holoniques est présentée dans la section1.5.

3.1.3. Le domaine de la solution

Le domaine de la solution représente une partie des concepts fournis par la plate-forme *Janus*. La plate-forme Janus est conçue pour faciliter la transition entre la conception et l'implantation logicielle d'un système. La figure 1.3 présente le diagramme UML simplifié de la plate-forme Janus et du domaine de la solution du méta-modèle CRIO.



1.3 : Diagramme UML simplifié de la plate-forme Janus et du domaine de la solution du méta-modèle CRIO [Nic08].

Janus fournit ainsi une implantation directe des cinq concepts à la base de la conception dans CRIO : organisation, groupe, rôle, holon et capacité.

- **Organisation, Groupe** : L'organisation est considérée comme une classe (au sens du méta-modèle objet) qui regroupe un ensemble de classes de rôles. Une organisation peut être instanciée sous forme de groupes, chaque groupe contenant un ensemble d'instances des différentes classes de rôles, associées à l'organisation qu'il implémente. Le nombre d'instances autorisées pour chaque rôle est spécifié dans l'organisation.
- **Rôle** : Un rôle est local à un groupe et fournit les moyens de communiquer au sein de ce groupe. L'un des aspects les plus intéressants de *Janus* concerne l'implantation du concept de rôle en tant qu'entité de première classe. Le rôle est considéré comme une classe à part entière, et les rôles sont implantés indépendamment des entités qui les jouent. Une telle implantation facilite la réutilisation des organisations dans d'autres solutions, mais autorise également une grande dynamique pour les rôles.
- **Holon** : Dans Janus, un agent est représenté par un holon atomique (ou non composé). Un holon peut jouer simultanément plusieurs rôles définis dans plusieurs groupes. Il peut accéder dynamiquement à de nouveaux rôles et libérer des rôles dont il n'a plus l'usage. Lorsqu'un holon accède à un rôle, il obtient une instance de la classe de ce

rôle qu'il stocke dans son conteneur de rôles et lorsqu'il libère un rôle, l'instance correspondante est supprimée. De ce fait, la taille d'un holon, en termes de code, dépend des rôles qu'il joue car il ne contient que le code des rôles qu'il joue à un instant donné. Pour accéder ou libérer un rôle, un holon doit satisfaire les conditions d'obtention ou de libération du rôle et du groupe correspondant (obtain Conditions et leave Conditions). Les conditions d'accès et de libération des rôles sont en revanche définies au niveau de l'organisation.

- **Capacité** : La notion de capacité permet de représenter les compétences d'un holon. Tout holon dispose, dès sa création, d'un ensemble de compétences de bases, dont la possibilité de jouer des rôles (et donc de communiquer), d'obtenir des informations sur les organisations et les groupes existants au sein de la plate-forme, de créer des holons, et d'obtenir de nouvelles capacités. Tout comme dans le domaine agent du méta-modèle CRIO, la capacité permet de faire l'interface entre le holon et les rôles qu'il joue. Le rôle requiert certaines capacités pour définir son comportement, les quelles peuvent ensuite être invoquées dans l'une des tâches qui composent le comportement du rôle. L'ensemble des capacités requises par un rôle sont spécifiées dans les conditions d'obtention du rôle. Une capacité peut être réalisée de diverses manières. Elle peut être implémentée directement ou bien implémentée à travers un service publié par le niveau inférieur.

Le méta-modèle CRIO fournit les abstractions nécessaires pour décrire la décomposition hiérarchique d'un système complexe en un nombre quelconque de niveaux d'abstraction. Il est nécessaire, pour mener à bien cette activité de décomposition hiérarchique d'un système complexe, de disposer d'un guide méthodologique pour assister le travail de l'ingénieur. Ce guide est fourni par le processus d'ingénierie logicielle ASPECS [Mas09]. Le processus ASPECS.

3.2. Le processus ASPECS

ASPECS est un processus d'ingénierie logicielle qui décrit, pas à pas, les étapes à suivre pour le développement de logiciels. Il fournit un guide complet, depuis l'analyse des besoins jusqu'à l'implantation et au déploiement de celui-ci sur une plate-forme spécifique. Il est basé sur le méta-modèle CRIO qui définit les principaux concepts pour l'analyse, la conception et l'implantation des systèmes multi-agents holoniques. ASPECS peut être considéré comme

l'extension des méthodologies PASSI et AgilePASSI à partir desquelles il a obtenu sa nature itérative. Le processus ASPECS permet la modélisation d'un système à différents niveaux de détails, en procédant par raffinements successifs. En effet, il offre la possibilité au concepteur de modéliser un système avec des entités de granularités différentes. Il peut ainsi récursivement décomposer un système en sous-systèmes, jusqu'à atteindre un niveau où la complexité des tâches demandées est suffisamment faible pour être exécutée par des entités considérées comme atomiques et faciles à implanter. La description du processus ASPECS et de ses différentes composantes est basée sur le méta-modèle de conception de processus de développement logiciel SPEM [SPEM07] proposé par l'OMG. Le langage de modélisation adopté est UML et AUML afin depleinement satisfaire les objectifs et les besoins spécifiques à l'approche orientée-agent. Le cycle de développement d'ASPECS est composé de quatre phases [Mas09] :

- **L'analyse des besoins** : Cette phase vise à fournir une description organisationnelle du système (décomposition hiérarchique du système). Elle doit également collecter les connaissances disponibles sur le domaine du problème et les organiser au sein d'une ontologie.
- **La conception d'une société d'agents** : Cette phase consiste à élaborer un modèle de société d'agents dont le comportement global doit être en mesure de fournir une solution au problème décrit dans la phase précédente. Après avoir modélisé le problème en termes d'organisations, de rôles, d'interactions et de capacités, le modèle de la société d'agents est décrit en termes de communication et de dépendances entre entités (agents et holons). Cette description est essentiellement basée sur les concepts du domaine agent du méta-modèle CRIO. Les connaissances sur le système sont affinées et intègrent les éléments spécifiques à la solution proposée. Le comportement des rôles est affiné sur la base des concepts nouvellement introduits, et associé aux agents en charge de les exécuter. Les interactions sont également raffinées pour décrire précisément leur contenu et leur éventuel protocole.
- **L'implantation de la solution :** Cette phase décrit l'architecture des holons impliqués dans la solution et doit fournir le code source de l'application. La phase d'implantation vise à fournir un modèle implantatoire de la solution multi-agents conçue durant la phase précédente. Cette étape est dépendante de la plate-forme d'implantation choisie. Dans ASPECS, la phase d'implantation est basée sur le modèle de la plate-forme Janus.

- **Le déploiement de la solution** : Cette phase constitue la phase finale en charge du déploiement de l'application sur la plate-forme choisie.

Les différentes phases du processus ASPECS seront détaillées dans le chapitre 5, dans la partie "*Phases du développement de l'étude de cas avec le processus ASPECS*".Le processus ASPECS offre plusieurs avantages, entre autres [Mas09]:

- **La réutilisation et la modularité des modèles et des connaissances** : ASPECS supporte la réutilisation des modèles et l'utilisation des schémas de conception organisationnels et favorise leur réutilisation dans de futures applications. Ceci, est possible grâce à la définition du comportement des rôles, sur la base des capacités. La réutilisation est également encouragée dans la modélisation des connaissances grâce à l'ontologie, considérée comme la base de connaissances commune et transversale au processus de modélisation. Les connaissances du domaine du problème sont regroupées et classifiées.
- **Le support de l'approche organisationnelle** : ASPECS supporte et conserve les avantages de l'approche organisationnelle depuis l'analyse jusqu'à l'implantation. Ceci est dû au fait que ASPECS (utilisé avec la plate-forme Janus) implante les organisations et les rôles en tant qu'entités de premier ordre et indépendamment des entités qui les jouent.
- **La prise en compte de la modélisation distribuée de la relation environnement-système** : ASPECS prend en compte la modélisation distribuée de la relation environnement-système ; ceci grâce à l'utilisation des "*boundary roles*" qui permettent de délimiter le périmètre de l'application à développer. Des capacités spécifiques sont ensuite utilisées pour faire l'interface entre ces rôles et les ressources environnementales nécessaires à l'application. Ces capacités permettent de définir le comportement des "*boundary roles*" en faisant abstraction de la représentation spécifique des ressources environnementales. L'usage de ces capacités permet de modifier la représentation de l'environnement, sans avoir pour autant à modifier le comportement des rôles. Cette approche facilite la distribution des applications dans des environnements hétérogènes et distribués.
- **Possibilité d'une modélisation du système à plusieurs niveaux d'abstraction:** ASPECS permet la décomposition organisationnelle hiérarchique d'un système jusqu'au niveau où le concepteur considère que la complexité des comportements est suffisamment faible pour une implantation immédiate. Ce qui le rend compatible avec

l'approche holonique qui consiste à modéliser un système à différents niveaux d'abstraction.

- **Le respect des standards** : ASPECS utilise le standard de notation UML et AUML, ce qui augmente la possibilité de pouvoir transférer les technologies multi-agents dans le milieu industriel.

3.3. Les agents holoniques

3.3.1. Définition d'un agent holonique

L'agent holonique est une entité auto-similaire composée de holons comme sous-structures (Figure 1.4). À un niveau d'observation donné, le holon composé est qualifié de super-holon. Les holons qui composent un super-holon sont appelés sous-holons ou holons membres.

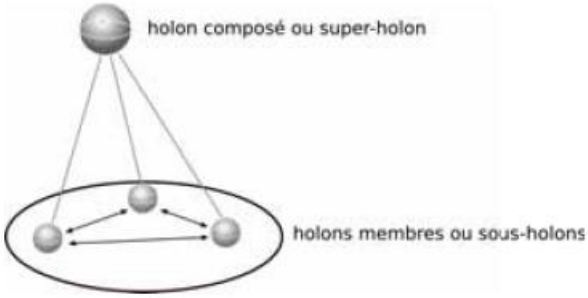


Figure 1.4: Un holon composé de trois holons membres.

Chaque holon composite de l'agent holonique est caractérisé par ses rôles, ses capacités et ses connaissances. La capacité est la description abstraite d'un savoir-faire. Elle regroupe la description de moyens qui permettent l'accomplissement d'une tâche. Elle est introduite pour permettre aux agents de raisonner sur leurs propres compétences et celles de leurs collaborateurs de sorte à pouvoir s'adapter et satisfaire des nouveaux objectifs. Elle possède une double fonction:(i) Elle constitue tout d'abord une interface entre l'agent et le rôle sachant que pour que l'agent obtienne un rôle, il doit disposer de toutes les capacités requises par ce rôle. (ii) Elle permet également, dans le processus de modélisation, d'effectuer l'interface entre deux niveaux d'abstraction adjacents dans la hiérarchie organisationnelle du système. Quant au rôle, il décrit un comportement défini dans un et un seul groupe. Un agent jouera simultanément et/ou successivement au cours de son exécution un grand nombre de rôles. Il tend à chaque fois à satisfaire les objectifs qui ont été attribués aux rôles qu'il joue et respecter le comportement de ces rôles. Un agent peut changer ses

rôles en accord avec ses besoins et ses objectifs. La communication entre les agents est liée aux rôles que ces derniers jouent à un instant donné car deux agents ne peuvent communiquer que s'ils jouent chacun, un rôle dans un groupe commun, qui leur précise comment ils sont organisés et comment ils interagissent en son sein pour satisfaire les objectifs (les besoins fonctionnels) qui leur sont assignés. Chaque groupe est associé à au moins un besoin fonctionnel. Chacun d'entre eux devra être satisfait par le comportement émergent des interactions de tous ou partie des rôles définis dans le groupe. Un holon peut donc être défini de la manière suivante [Nic07] :

$H\ n =< R_n, H_{n-1}, OP, \psi >$

avec :

R_n : l'ensemble des rôles joués par H_n , $R_n = 2^{roles}(O_n)$ avec O_n l'ensemble des groupes où H_n joue au moins un rôle.

H_{n-1} : l'ensemble des sous-holons membres du super-holon H_n.

OP : l'ensemble des groupes qui participe à la vie et au fonctionnement du super-holon H_n et qui contribue notamment à la satisfaction des objectifs liés aux rôles de R_n.

ψ: $H_{n-1} \rightarrow 2^{roles(OP)}$: fonction associant un sous-holon membre à l'ensemble des rôles qu'il joue dans les groupes définis au sein de H_n , tel que $\forall\ h_i \in H_{n-1}$, $\psi(h_i) \neq \emptyset$ et $card(\psi(h_i)) \geq 2$. La fonction roles fournit l'ensemble des rôles définis dans les groupes de OP.

3.3.2. D'une décomposition organisationnelle vers une holarchie

La notion de holon est très proche de celle d'organisation [Nic07] en tant que comportement à part entière ou composante d'un comportement de plus haut niveau. Ce parallèle entre ces deux concepts facilitera d'autant le processus d'agentification du modèle issu de la modélisation du problème. La figure 1.5 illustre cette association entre la décomposition hiérarchique organisationnelle d'un problème et la création d'une holarchie. En respectant l'approche décrite dans le domaine du problème, le système est décomposé en différents niveaux d'abstraction.

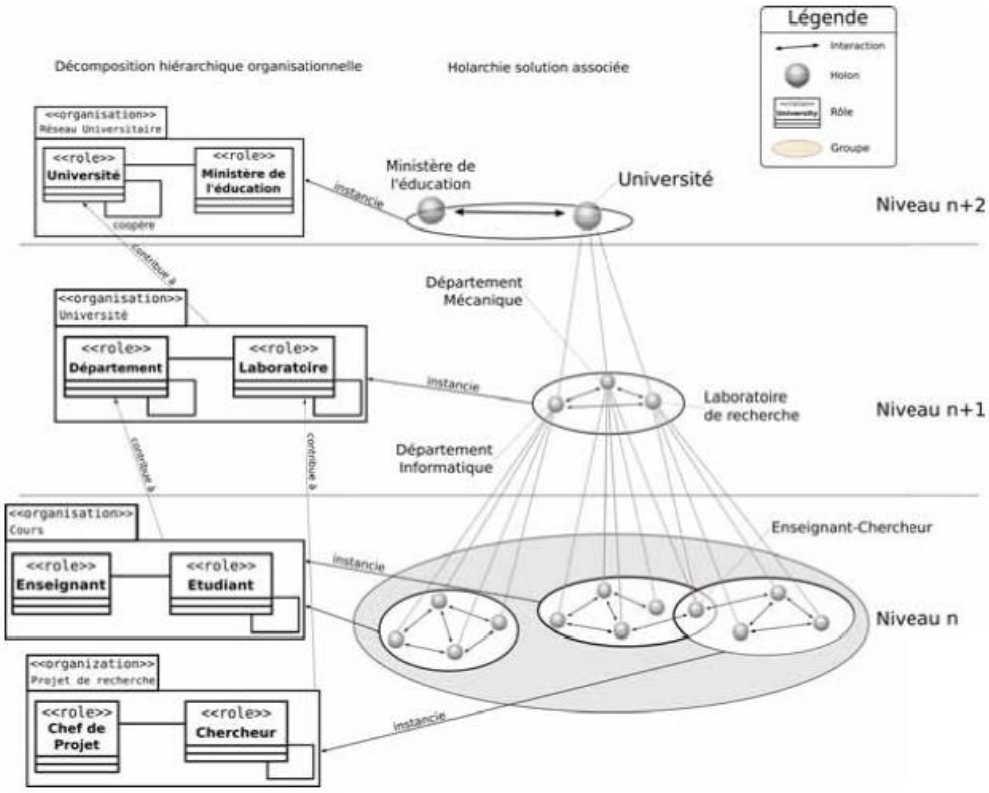


Figure 1.5 : Lien entre décomposition hiérarchique organisationnelle et holarchie d'exécution [Nic07].

La construction de la holarchie consiste à instancier les organisations sous forme de groupes. Un ensemble de holons est ensuite créé à chaque niveau, chacun d'eux jouant un ou plusieurs rôles dans un ou plusieurs groupes du niveau considéré. Les relations de composition entre super-holons et sous-holons sont ensuite définies en accord avec les contributions entre organisations dans la hiérarchie organisationnelle. Par exemple, les super-holons de niveau n+1 jouent des rôles dans les groupes de niveau n + 1. Les membres respectifs de ces super-holons jouent des rôles dans les différents groupes de niveau n, qui contribuent au comportement des rôles de niveau n + 1 joués par leur super-holon. Les aspects individuel et collectif d'un holon sont ainsi pleinement exploités.

Dans une holarchie on distingue 2 types de groupes [Rod05] :

- les groupes de production: sont des instances d'organisations qui décrivent comment les membres interagissent et se coordonnent pour satisfaire les objectifs et les tâches assignés à leur super-holon. La définition de ces organisations est dépendante du problème traité.

- le groupe holonique : est une instance de l'organisation holonique (cf : paragraphe 1.5.3) qui précise comment les membres sont organisés pour gérer le super-holon.

La figure 1.6 illustre cette distinction entre le groupe holonique et les groupes de production au sein d'un super holon. La holarchie présentée détaille la modélisation holonique (à l'aide de CRIO) d'une université. Le holon H1 est composé de deux groupes de production g1 et g2, et du groupe holonique gH au niveau n − 1.

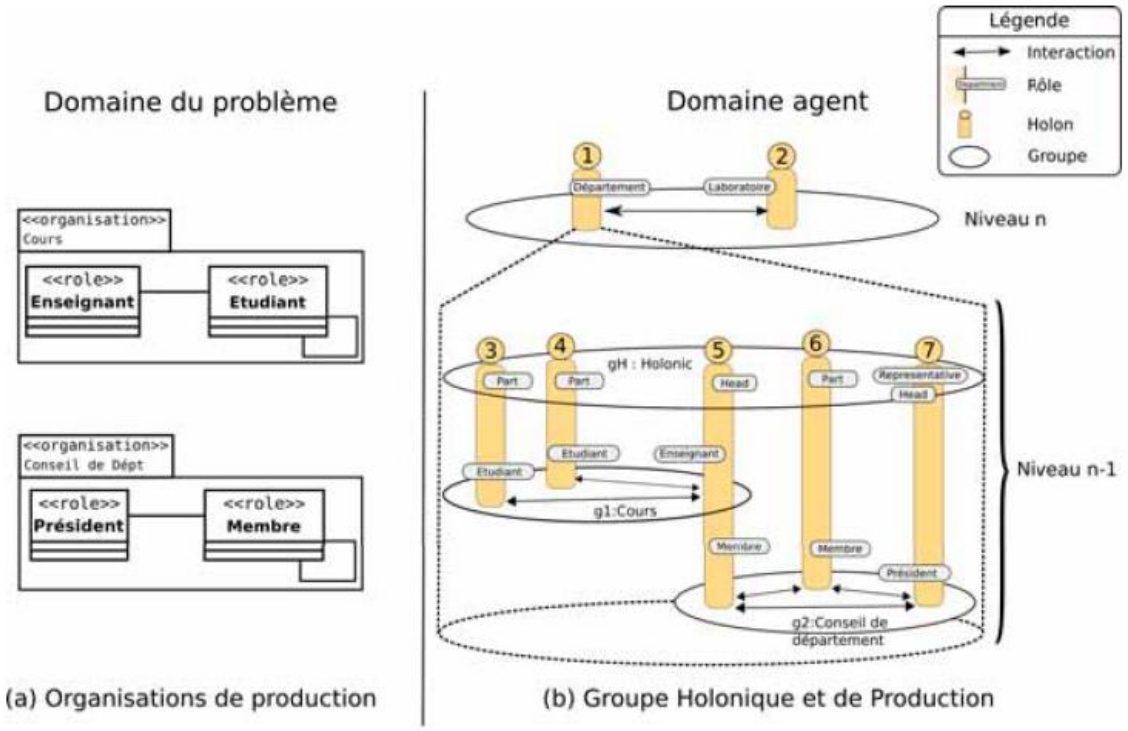


Figure 1.6 : Un exemple de la structure de la holarchie Université [Nic07].

3.3.3. L'organisation holonique

L'organisation holonique est conçue pour décrire le mode de gestion et la structure d'un super-holon en termes de répartition d'autorité et de pouvoirs. Elle est inspirée de la notion de Groupe Modéré [Ger99] et fut choisie pour sa souplesse et le large panel de configurations qu'elle offre, en modifiant simplement le degré d'engagement des holons membres envers leur(s) super-holon(s). Dans un groupe modéré, un sous-ensemble des membres représente les autres membres à l'extérieur du groupe. Pour représenter un groupe modéré avec une approche organisationnelle, cinq rôles, qualifiés de rôles holoniques, ont été identifiés :

- **Head** : représente un statut privilégié auquel les membres peuvent conférer un certain degré d'autorité et certains droits particuliers. Il prend part au processus de prise de décision et, en contrepartie, il assure une partie de la charge d'administration du super-holon. Cette charge dépend de la configuration choisie et peut varier au cours de la vie du super-holon.
- **Representative** : il fait partie de l'interface visible du super-holon. Il joue le rôle d'interface entre l'intérieur et l'extérieur du super-holon. Il assure la redistribution et l'éventuelle traduction de l'information arrivant de l'extérieur. Il représente les autres membres à l'extérieur du super-holon. Plusieurs Representatives peuvent être en charge de représenter les autres membres.
- **Part** : identifie les membres d'un unique super-holon. Ils sont normalement en charge de l'exécution des tâches qui leur sont affectées par les Heads. Ils peuvent éventuellement prendre part au processus de décision. Cette participation à la prise de décision dépend de la structure de gouvernement choisie pour le super-holon.
- **MultiPart** : extension du rôle Part, il identifie les membres partagés entre plusieurs super-holons. Le fait qu'un membre puisse être partagé entre plusieurs super-holons peut éventuellement générer des conflits d'intérêt ou d'autorité. Si au cours de la vie du super-holon, un de ses membres Part rejoint un autre super-holon, il devra changer de rôle pour devenir Multipart.
- **Stand-Alone** : Ce rôle représente le statut attribué aux holons non-membres. Tous les holons non-membres sont extérieurs au super-holon, et sont perçus par les membres à travers le statut Stand-Alone. Ce rôle a été ajouté pour gérer le recrutement de nouveaux membres au sein d'un super-holon.

Les quatre premiers rôles holoniques décrivent le statut d'un membre au sein d'un super-holon et participent à la définition de l'organisation holonique. Chacun de ces rôles peut être joué par un ou plusieurs membres, sachant que tout super-holon doit disposer d'au moins un Representative et un Head. Les rôles Head, Part et Multi-Part sont exclusifs entre eux, alors que Representative peut être joué simultanément avec l'un des trois autres. Les interactions entre ces rôles sont décrites par l'organisation holonique. Une instance de cette organisation est nommée groupe holonique. Chaque membre d'un super-holon doit jouer au moins un rôle dans le groupe holonique, et tout holon qui rejoint le super-holon après sa création, devra également jouer un rôle dans ce groupe.

L'organisation holonique offre un large panel de configurations possibles pour définir le gouvernement d'un super-holon [Rod05] parmi lesquelles on distingue :

- **Monarchie** : Le commandement est centralisé au niveau d'un Head unique. La notion de monarchie ne réfère pas ici au mode de nomination du Head, mais uniquement à la répartition des pouvoirs au sein de la communauté et au fait qu'un unique Head contrôle l'intégralité du processus de prise de décision. Le mode de nomination est une partie intégrante de la dynamique de création d'un super-holon et se doit d'être spécifié séparément.
- **Oligarchie** : Un petit groupe de Heads partage le commandement sans en référer aux autres membres de statut Part.
- **Polyarchie** : Un petit groupe de Heads partage le commandement, mais ils peuvent se référer aux Parts pour certaines décisions (via un vote par exemple).
- **Apanarchie** : Le commandement est entièrement partagé entre tous les membres du super-holon. Toute la communauté est impliquée dans le processus de prise de décision.

4. Conclusion

Les systèmes multi-agents holoniques (SMAH) peuvent offrir une alternative intéressante à la vision hiérarchique et la structure pyramidale et récursive de la plupart des systèmes complexes ce qui justifie l'intérêt que leur accorde la recherche. Néanmoins, le développement des applications basées sur ces systèmes demeure peu solide, eu égard à la faible importance accordée à l'activité de test qui, représente une tâche importante dans l'assurance de leur qualité. Dans le chapitre suivant nous aborderons, dans un premier temps, les différentes méthodes de test classiques et nous mettrons en relief leur insuffisance par rapport au test agent. Nous consacrerons, par la suite, un chapitre entier pour l'approche que nous avons proposée pour le test des systèmes multi agents holoniques.

Chapitre 2

Test et systèmes multi-agents

1. Introduction

Durant le développement et la maintenance d'un logiciel, l'activité de validation requiert une attention particulière de la part du développeur du logiciel. Elle exige, certes, un coût mais elle est nécessaire pour l'assurance de la qualité du logiciel. L'ignorance de cette activité de validation peut se traduire par des pertes financières, humaines, etc. Au-delà des pertes humaines et matérielles, la correction des défauts se ferait avec des couts élevés, d'où l'importance de la validation du logiciel avant sa mise sur le marché. Parmi les moyens de la validation et la vérification des logiciels, se trouve le "*test*" qui est apparu dans les années 50 et qui a depuis largement prouvé son intérêt économique et qualitatif. Depuis sa création, il s'est imposé comme étant le moyen principal pour la validation du fonctionnement d'un programme [Ham93]. Il a pour objectif d'examiner ou d'exécuter un programme dans le but d'y révéler des erreurs, ce qui augmente la confiance dans le logiciel. Il est souvent défini comme le moyen par lequel on s'assure qu'une implantation est conforme à ce qui a été spécifié.

Contrairement aux autres paradigmes de programmation antérieurs (procédural, orienté objet, etc.), la phase de test des systèmes multi-agents n'a reçu que très peu d'intérêt de la part des chercheurs, comparé à l'intérêt accordé aux premières phases des différents cycles de développement orienté agent. En effet, la plus grande majorité des

contributions dans ce domaine étaient plutôt orientées vers les architectures, les protocoles, les infrastructures de messagerie et les interactions inter et intra agents [Fer98]. Dans ce chapitre, nous aborderons les méthodes de test classiques et nous mettrons en relief leur insuffisance pour le test des agents et nous décrirons quelques travaux sur le test des agents.

2. Définition du test

Devant la pluralité des définitions du terme test de logiciel, nous avons opté pour les définitions suivantes :

Selon l'IEEE (Standard Glossary of Software Engineering Terminology) [IEEE90], Le test est l'exécution ou l'évaluation d'un système ou d'un composant, par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou pour identifier des différences entre les résultats attendus et les résultats obtenus.

Selon G. J. Myers (The Art of Software Testing) [Mye04], tester c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts.

Selon AFCIQ (Norme Expérimentale de Terminologie), le test est une technique de contrôle consistant à s'assurer, au moyen de son exécution, que le comportement d'un programme est conforme à des données préétablies.

D'après les définitions déjà vues, il est clair que le test logiciel a pour but de détecter des erreurs dans un programme. Ceci consiste à exécuter le programme sous test avec un ensemble de *données en entrée*, et à comparer les résultats obtenus avec les résultats attendus. Si les résultats diffèrent, cela implique l'existence d'une erreur qu'il faut localiser et corriger. Les *données en entrée* appartiennent à un domaine d'entré 'D' très vaste ou infini parfois. L'idéal est de tester le programme avec tout le domaine d'entrée (test *exhaustif*). Ce type de test assure la détection de toutes les erreurs avec un taux de confiance égal à 100% dans le programme testé. Mais ceci est presque impossible à cause de l'infinité du domaine d'entrée lors de l'étape d'exécution d'où l'apparition des techniques de génération des cas de test. Ces dernières sont des techniques qui exploitent les caractéristiques du programme, son comportement, ses spécifications, etc., pour générer des *données de test* (un sous ensemble du domaine d'entrée) pouvant assurer un taux de confiance proche à celui attendu par le test exhaustif.

On appelle cas de test le couple formé par une entrée, la donnée de test générée, et la sortie attendue pour cette entrée. Après l'exécution de chaque cas de test, il faut comparer le

résultat obtenu avec le résultat attendu. Ce prédicat qui permet de déterminer si le résultat est incorrect est appelé un oracle ou une fonction d'oracle qui peut être manuelle ou automatique. Lorsqu'un cas de test échoue, il faut localiser la source de la défaillance dans le programme pour pouvoir corriger la faute. Cette étape de localisation est appelée la phase de diagnostic. Lorsque le résultat obtenu est conforme à l'oracle, la dernière étape du test consiste à déterminer si les cas de test sont suffisants pour garantir la qualité du logiciel. Pour cela, il faut définir un critère de test ou critère d'arrêt et vérifier si les cas de test générés vérifient ou non ce critère. Les techniques de génération visent donc souvent à vérifier un critère d'arrêt particulier (plutôt qu'à chercher à détecter des erreurs). Pour générer des cas de test en fonction d'un critère de test, il est possible de définir des objectifs de test. Par exemple, si le critère exige l'exécution de toutes les instructions du programme au cours du test, "*couvrir l'instruction 11*" est un objectif de test possible. Il faut ensuite écrire un cas de test qui vérifie cet objectif.

3. Le test dans le cycle de développement

Parmi les avantages du test, il faut signaler le fait que ses activités se déroulent tout au long du cycle de développement et interagissent fortement avec les activités de développement. Le processus de test doit donc suivre plusieurs étapes, de manière incrémentale, et ce, tout au long de l'implémentation du système. On distingue classiquement trois phases successives de test dans le processus de vérification/validation d'un produit logiciel : test unitaire, test d'intégration et test de validation. Ces grandes phases de test sont présentes dans de nombreux cycles de vie. Le plus connu d'entre eux est certainement le cycle de vie en "V" de la figure 2.1 où la branche descendante du cycle correspond aux phases de développement et où la branche montante correspond aux tests effectués à chaque étape du développement.

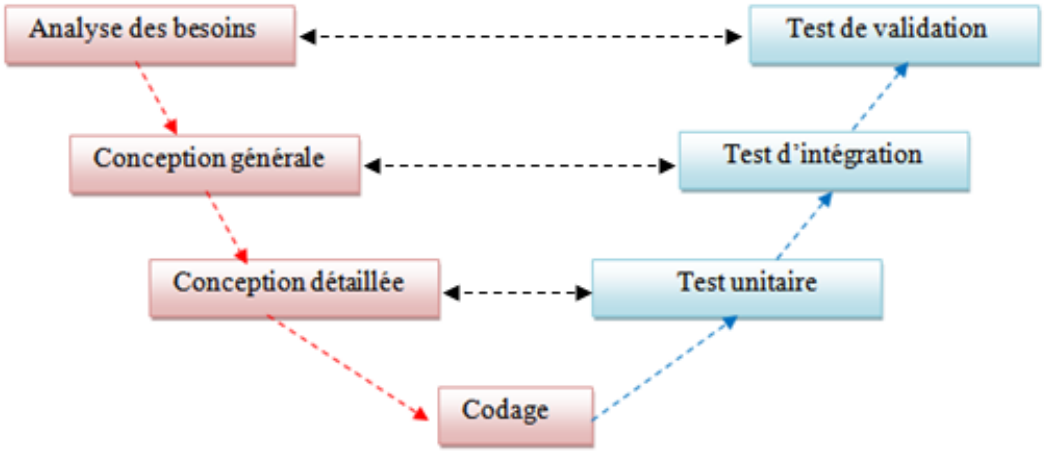


Figure 2.1 : Les niveaux de test.

3.1. Test unitaire

Le test unitaire [Xan00], [Bur02] est lié à chaque composant du logiciel à tester. Il peut être entamé dès que celui-ci est codé (dans certaines approches agiles avant). En pratique, la nature des composants testés dépend de la méthode de conception employée. Il s'agira des modules élémentaires dans le cas d'une conception dite classique ou structurée; des objets terminaux dans le cas de la programmation orientée objet ; ou encore des agents dans la programmation orientée agent. Le but principal des tests unitaires est de comparer les fonctions d'une unité par rapport à certaines spécifications. Les tests unitaires sont menés en se basant sur le plan de tests unitaires conçus à l'étape de «conception détaillée» du cycle de vie du logiciel. Ce plan met en évidence les spécifications à tester pour chaque unité. Le principal problème rencontré dans le test unitaire est le fait que chaque entité constituante du logiciel doit être testée seule indépendamment des autres, aucune interaction avec d'autres entités du logiciel n'est prise en compte. Dans ce cas, il faut que ces interactions soient court-circuitées par l'utilisation de bouchons (simulateurs).

3.2. Test d'intégration

Le test d'intégration [Xan00], [Mye04] se positionne au niveau des interfaces entre les unités testées au niveau précédent pour assurer la bonne communication entre elles. Cette phase de test ne consiste pas à répéter exactement le même type de test que la phase unitaire sur des groupes d'entités. Durant cette phase, on cherche à tester la manière dont chaque entité interagit dans son environnement d'utilisation. Le test d'intégration peut être effectué à chaque fois qu'une nouvelle entité est disponible. Il s'agit alors de tester sa bonne intégration

parmi celles déjà testées. Le principal problème rencontré dans le test d'intégration est le fait qu'il soit effectué sur un ensemble incomplet d'entités. Il arrive qu'il fasse appel à des entités qui ne sont pas encore disponibles ou qui ne puissent être sollicitées que par des entités qui ne sont pas encore intégrées. Dans ce cas, un travail de préparation de l'ensemble des entités doit être réalisé avant le test. On construit de petits programmes de remplacement (des bouchons). La construction des bouchons est relativement complexe et il est préférable de les éviter ou du moins limiter le recours à eux. Ceci est possible à travers le choix adéquat de la stratégie d'intégration et de l'ordre selon lequel les entités seront intégrées (l'ordonnancement). En effet, selon la stratégie d'intégration et l'ordonnancement choisis pour le développement du logiciel, l'effort de préparation nécessaire à chaque phase de test d'intégration et le nombre des bouchons qu'il faut développer sera diminué. Parmi les stratégies d'intégration on trouve:

- La stratégie "big-bang" où le test d'intégration ne commence que lorsque toutes les entités du logiciel sont disponibles. On les assemble alors toutes d'un coup et on teste le logiciel complet. Cette stratégie a l'avantage d'éliminer complètement le besoin de préparation au test. Malheureusement il s'en suit d'énormes difficultés pour identifier l'origine des erreurs mises au jour par le test.
- La stratégie par incrément où, après avoir défini l'ordre dans lequel on va intégrer les entités, on les ajoute une par une au logiciel à chaque étape du test. Ici, la localisation des erreurs est facilitée : les erreurs détectées sont majoritairement situées dans la dernière entité intégrée. En contrepartie, cette stratégie est fastidieuse. Elle demande un grand nombre d'étapes. De plus, elle induit un travail de préparation important pour bouchonner les composants manquant à chaque étape de test.
- La stratégie par agrégat propose de regrouper les entités par groupe, un agrégat, pour former une fonctionnalité de haut niveau du logiciel. On crée de multiples sous-ensembles d'entités pour ensuite assembler ces sous-ensembles entre eux et créer le logiciel complet. Les erreurs rencontrées au cours du test sont plus difficiles à corriger que dans la stratégie par incrément. Une erreur détectée peut être localisée dans n'importe quelle entité de l'agrégat. Par contre, elle limite le besoin de bouchons et de lanceurs en regroupant les entités s'appelant les unes les autres au sein du même agrégat. Elle diminue le nombre de cas de tests à produire pour solliciter le logiciel. Les entrées de tous les composants sont en rapport les unes avec les autres. Enfin, elle

est particulièrement adaptée aux architectures logicielles où les fonctionnalités sont indépendantes et facilement associées à un groupe d'entités restreint.

3.3. Test de validation

Le test de validation [Xan00] permet de vérifier que nous avons bien réalisé le logiciel et qu'il s'agit bien du bon logiciel. Il s'agit donc clairement de quitter le point de vue développeur pour adopter celui de l'utilisateur final et se fixer pour objectif de s'assurer que le logiciel réalise effectivement tout ce que le client est en droit d'attendre. Cet objectif est d'autant plus important qu'il est en général associé à un objectif contractuel. Lors des tests de validation, le produit logiciel est à présent entièrement assemblé et les tests de validation doivent se dérouler dans des conditions aussi proches que possible des conditions d'exploitation. L'idéal étant bien sûr de réaliser les tests sur la machine d'exploitation dans son environnement réel. Il n'est pas toujours possible de satisfaire ces deux conditions, notamment lorsqu'il s'agit d'un logiciel de contrôle-commande d'un satellite ou d'une usine, ou encore d'un logiciel gérant les transactions bancaires. Dans tous ces genres de cas, on devra alors simuler l'environnement réel en veillant à être le plus représentatif possible. Il faudra également veiller à reproduire fidèlement les autres contraintes d'exploitation.

3.4. Test de non-régression

Le test de non-régression consiste à re-tester le logiciel afin de vérifier soit la présence d'anomalies ayant échappées à tous les tests, soit de nouvelles anomalies engendrées par les corrections des erreurs rencontrées pendant les différentes étapes de tests, ou bien crées lors d'introduction de nouvelles fonctionnalités demandées par le client. C'est tout l'intérêt des tests de non régression [Xan00] à la suite de la modification d'un logiciel (ou d'un de ses constituants). Un test de non-régression a pour but de montrer que les autres parties du logiciel n'ont pas été affectées par cette modification.

4. Les méthodes de test

Il existe plusieurs techniques et stratégies de contrôle couvrant la majorité des phases du cycle de développement du logiciel. Parmi toutes les solutions de test existantes, on s'intéresse plus particulièrement, dans la section suivante, aux solutions de test basées sur les modèles (en anglais Model-based Testing (MBT)). Celles-ci s'appuient sur un modèle du

système afin de produire des cas de test fonctionnels. Une méthode MBT produit des cas de test à caractère fonctionnel, à partir d'exigences relatives au niveau de test souhaité. Tous les niveaux (d'unitaire jusqu'à système) peuvent être considérés par une telle solution. Les autres solutions seront citées pour un intérêt bibliographique.

4.1. Test basé sur les modèles

Les approches et les techniques de génération automatique de tests ont été développées depuis le début des années 90, et sont connues au niveau international sous le terme "*Model-Based Testing (MBT)*" [Apf97], [Elf01]. Elles ont donné naissance aujourd'hui à des solutions outillées de production et de maintenance du référentiel de tests, qui s'intègrent avec la gestion du référentiel et les environnements d'automatisation des tests, et supportent un processus de bout en bout: des exigences métier au référentiel de tests automatisés. Une solution de Model-based Testing consiste à produire des cas de test à partir d'un modèle du système à tester. Un processus MBT (Figure 2.2) se décompose couramment en cinq phases qui sont :

1. La modélisation d'un modèle abstrait dédié au test d'un système donné.
2. La production des cas de test abstraits à partir de ce modèle. Cette production de tests est la plupart du temps automatisée par des outils de génération automatique de tests.
3. La concrétisation des cas de test abstraits en tests exécutables sur le système sous test.
4. L'exécution des tests sur le système et la constitution de leur verdict.
5. L'analyse des résultats ainsi obtenus.

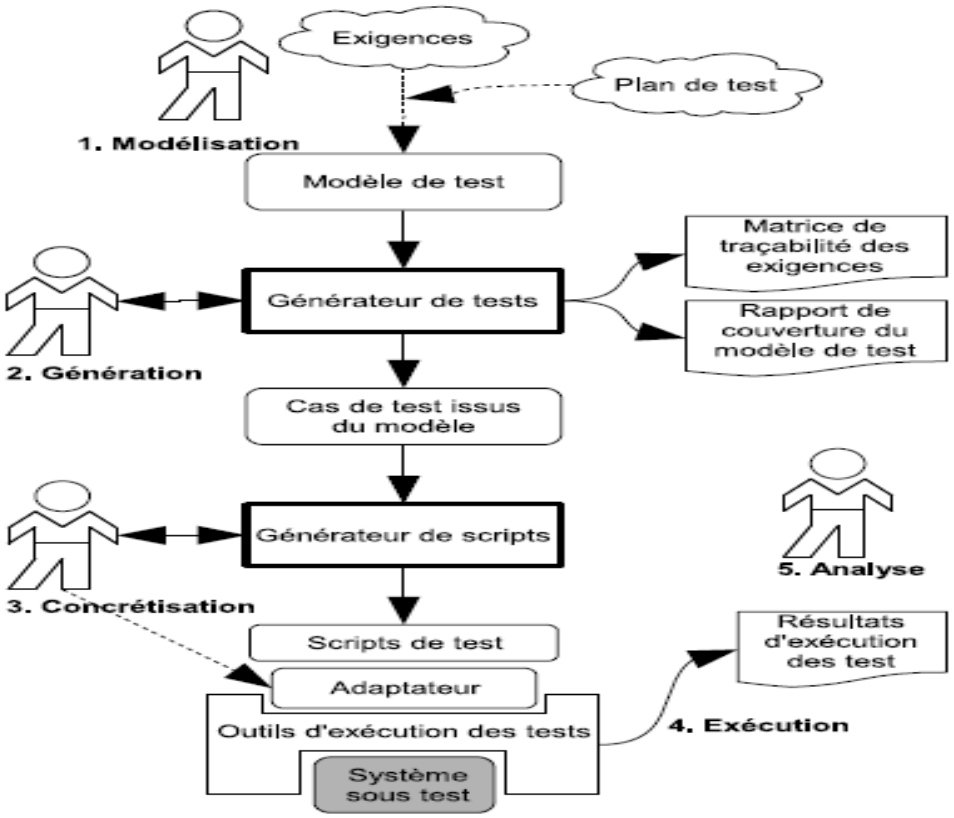


Figure 2.2 : Le processus de Model-based Testing [Elf01].

Les solutions MBT existantes sont nombreuses et se distinguent notamment par le type de modèle, la méthode de génération des cas de test, ou encore le type d'exécution de ces tests. Les sous-sections suivantes présentent un état de l'art caractérisant les différentes phases d'une solution MBT.

4.1.1. La modélisation

La production d'un modèle dédié au test est l'activité principale d'un utilisateur de processus Model-based Testing. Il s'agit de concevoir un modèle à partir duquel seront produits les cas de test à exécuter sur le système sous test. Ce modèle représente les comportements du système à un certain niveau d'abstraction. Les modèles d'une solution MBT doivent répondre à un ensemble de caractéristiques, parmi lesquelles on trouve :

- L'indéterminisme de comportement est une caractéristique des systèmes concurrents par exemple. L'adéquation entre le modèle et le système est respectée si cet indéterminisme est exprimé dans le modèle de test. Dans ce cas, les cas de test issus

de ce modèle ne sont pas des séquences de stimulus, mais des arbres ou graphes de stimulus, permettant de satisfaire cet indéterminisme.

- Certains systèmes ont une caractéristique temporelle forte. Les systèmes temps-réel se différencient des autres systèmes par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat. Dans la littérature, [Lap92] évoque la vérification et la validation d'applications temps-réel.
- Enfin, les modèles de test se distinguent par leur caractère dynamique discret ou continu (ou une combinaison des deux dans le cas de modèles hybrides). Ces différentes propriétés caractérisant le système sous test influent sur le choix du paradigme de modélisation, ainsi que sur les choix techniques de génération de tests.

4.1.2. La génération des cas de test

La production des cas de test au sein d'un processus MBT consiste en la génération de tests à partir du modèle de tests préalablement conçu. Cette génération s'appuie sur les comportements issus du modèle de test et sur des critères de sélection de tests choisis par l'ingénieur-validation. La figure 2.3 donne une classification liée à la génération des cas de test selon les deux dimensions.

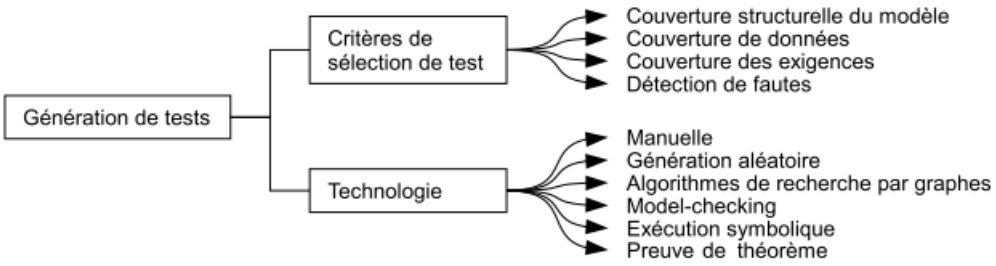


Figure 2.3 : Classification des problématiques liée à la génération de cas de test [Utt06].

a. Critères de sélection des tests

Un critère de sélection permet de générer un ensemble de cas de test partageant des propriétés communes. Le critère de sélection permet de mesurer la qualité d'une suite de tests, par son taux de couverture du modèle de test. La couverture nécessite l'exécution du logiciel ou d'une de ses représentations. Elle est effective si le composant ou le comportement considéré a été sollicité. Parmi les critères de sélection applicables sur un modèle de test, on trouve les grandes familles suivantes :

- **Critères de couverture structurelle du modèle :** Leur objectif est de garantir des types de couverture de la structure du modèle.
    - **Les critères de couverture orientés "flux de contrôle"** sont directement issus des critères de couverture de code. La couverture porte sur les instructions, les décisions ou les chemins exprimés par le graphe de flots de contrôle issu du modèle [Off99], [Hay01].
    - **Les critères de couverture orientés "flux de données**" se basent sur l'utilisation et la définition des variables du modèle. Parmi ces critères, ***all-définitions*** assure la couverture d'au moins une paire définition-utilisation pour chaque variable. Le critère ***all-uses*** assure la couverture de toutes ces paires, c-à-d de toutes les utilisations de toutes les définitions. Le critère ***all-def- use-paths*** garantit la couverture de toutes les paires définition-utilisation *(dv; uv*) par tous les chemins de *dv* à *uv* [Fra88], [Sal00].
    - **Les critères de couverture basés sur les transitions :** sont spécifiques aux diagrammes états-transitions (FSM, LTS, machines à états UML, etc.). Les éléments de base de ces types de couverture sont les états et les transitions [Sou00].
    - **Les critères de couverture basés sur UML** : permettent de satisfaire la couverture d'éléments propres à la notation UML (Association-end multiplicity coverage, Generalization coverage, Class attribute coverage) [And03].

- **Critères de couverture des données** : Leur objectif est de garantir une couverture des valeurs de chaque variable du modèle. Le critère ***One-value*** garantit que chaque variable du modèle obtient une valeur de son domaine. Le critère ***All-values*** garantit que chaque variable du modèle obtient toutes les valeurs de son domaine [Kos04], [Bac04].

- Parmi les autres critères de sélection de tests connus, on peut également citer les stratégies de test basées sur la ***détection de fautes***, dont le test par mutation qui permet de mesurer le pouvoir de détection de fautes d'une suite de tests. Ces approches se basent sur une mutation syntaxique ou sémantique du modèle.
- Enfin, les critères basés sur ***les exigences*** permettent de qualifier une suite de tests en fonction des exigences fonctionnelles qu'elle couvre.

b. Les technologies de génération

L'intérêt du Model-based Testing réside dans sa capacité à produire une série de cas de test de manière automatique. Cette génération peut s'effectuer de manière stochastique, par l'utilisation d'algorithmes de recherche par graphes, par model-checking, par exécution symbolique ou encore par preuve déductive.

- Les algorithmes de recherches à partir de graphes permettent de révéler des traces couvrant les arcs et/ou les nœuds d'un graphe.
- Le model-checking est une technique permettant de caractériser une propriété d'un système comme vraie ou fausse. Il permet de vérifier qu'une propriété, sur un modèle donné, est satisfaite. La technique model-checking peut fournir un contre-exemple permettant de démontrer qu'une propriété n'est pas satisfaite. Générer des cas de test respectant une propriété *p* par cette technique consiste donc à générer un contre-exemple montrant que *p* est satisfaisable (*p* est alors la négation d'un objectif de test) [DeM04].
- L'exécution symbolique de modèles consiste à animer un modèle de test exécutable à partir de valeurs contraintes. Le système est alors représenté par un ensemble de contraintes sur ses variables d'état. Des solveurs de contraintes booléens (type SAT), numériques ou ensemblistes sont alors utilisés pour générer des traces valides respectant ces contraintes. Chaque trace contrainte est ensuite évaluée, selon le domaine contrainte de chaque variable, pour constituer un cas de test.
- La preuve déductive de théorèmes permet de vérifier la satisfaisabilité de formules logiques. Dans le cas d'une approche MBT, le système sous test est modélisé par un ensemble d'expressions logiques (ou prédicats) spécifiant les comportements du système. La génération de tests se fait alors à partir d'un partitionnement en classes d'équivalence de l'ensemble des prédicats valides. Une classe d'équivalence représente un comportement modélisé du système. Classiquement, le partitionnement est réalisé par une transformation des expressions logiques en forme normale disjonctive. Parmi les travaux qui utilisent cette technique on trouve : [Hel97] utilise l'assistant de preuve de théorème Isabelle/HOL [Pau93] pour générer des cas de test à partir de spécifications Z.
- L'utilisation de la logique de programmation par contraintes (CLP) est très répandue dans le domaine de la génération des tests. Parmi les travaux qui utilisent cette logique on trouve : [Got98] propose une génération automatique de données de test

dédiées au test structurel. [Pre01] utilise cette logique pour le test de systèmes réactifs. [Col04] utilise un solveur de contraintes ensemblistes pour calculer des préambules de test à partir d'une machine B.

4.1.3. L'exécution des cas de test

On distingue la génération de tests on-line et la génération de test off-line. La génération de tests on-line exploite directement les données réelles du système suite à sa stimulation. Le générateur de tests est alors fortement couplé au système sous test; le test se construit de manière incrémentale en fonction des réactions du système après exécution. Ce type de test est particulièrement adapté aux systèmes non-déterministes. La génération de tests off-line crée une distinction entre la phase de génération et la phase d'exécution. Ainsi, les tests peuvent être générés puis exécutés et/ou stockés ultérieurement par des outils de gestion et d'exécution des tests. Les tests peuvent être générés et exécutés sur des machines différentes, dans des environnements différents. A partir d'un modèle inchangé, une campagne d'exécution des tests ne nécessite pas de génération, alors que le test on-line nécessite ces deux phases.

4.2. Le test structurel statique

Le test structurel statique regroupe toutes les méthodes structurelles qui ne nécessitent pas l'exécution du code source pour être appliquées et dans lesquelles, le code source est analysé et examiné en tant qu'entité indépendante et passive. Le principal attrait de cette approche concerne son caractère absolu et formel. Son inconvénient majeur réside dans le fait que l'aspect pratique, concernant l'exécution réelle du programme avec de vraies données de test, n'est pas considéré. Parmi les techniques du test structurel statique on trouve :

4.2.1. Revues de code et lectures croisées

Les revues de code [Wie01] peuvent être menées soit individuellement soit par un groupe. Dans cette dernière option des extraits de programme sont relus par des membres d'une équipe de programmeurs, dans une réunion organisée par les chefs de projet, à laquelle assiste souvent un responsable qualité. Malgré leur efficacité indiscutable, les revues de code mobilisent un grand nombre de ressources humaines et sont parfois difficiles à être supportées par de petites structures de développement. C'est pourquoi on leur préfère les lectures croisées [Wie01] impliquant deux personnes, l'une examinant et critiquant le code

source de l'autre et vice et versa. L'objectif principal de ces deux techniques concerne le fait de s'assurer du respect de certains standards de codage et de l'identification de certaines pratiques de programmation suspectes.

4.2.2. Exécution symbolique

L'exécution symbolique [Xan00] consiste à assigner des valeurs symboliques aux variables intervenant dans l'algorithme et non pas des valeurs numériques. Le programme est ensuite exécuté selon la sémantique de chacune des instructions. L'exécution symbolique produit, pour chaque variable de sortie du programme, une expression retraçant les calculs effectués pour l'obtenir (le long de ce chemin) ainsi que les conditions associées (sur le chemin). L'exécution symbolique est une méthode qui se situe entre la vérification des programmes et les approches de test classique. La principale différence avec la preuve formelle est que la validation s'effectue ici sur des résultats symboliques finaux et non sur des assertions intermédiaires.

4.3. Le test structurel dynamique (boîte blanche)

Les tests structurels dynamiques [Nta88] sont des tests qui s'intéressent au code du logiciel à tester (détermination des cas de test en fonction du code). Ils permettent de valider la structure de chaque module composant le logiciel et d'avoir l'assurance que toutes les parties d'un programme ont été exercées. On vérifie ici ce que le programme fait et non pas ce qu'il est censé faire. Les techniques dites de test structurel sont caractérisées par le fait que le jeu de test est sélectionné à partir d'une description de la structure du produit logiciel sous test et non pas de ses spécifications. Deux principaux types de description sont utilisés: le graphe de contrôle et le flot de données :

- **Le graphe de contrôle (Control Flow Graph ou CFG)**
  Il s'agit de représenter via un graphe la structure interne du code. Un graphe de contrôle est donc constitué de nœuds représentant soit des points de décision (if else, while, switch…) soit une instruction ou un bloc séquentiel d'instructions. Les arcs (ou branches) de ce graphe symbolisent la possibilité de transfert de l'exécution d'un nœud (ou d'un bloc) à un autre [Mye04].
- **Le flot de données**
  Le graphe de flot de données [Nta88] s'obtient en annotant le graphe de contrôle par des informations pertinentes quant à la manipulation des variables.

Plus précisément, on dira qu'une variable est définie lors d'une instruction si elle appartient au membre gauche d'une affectation ou si elle apparaît comme paramètre d'une instruction de lecture. Par contre, cette variable est dite référencée (utilisée) si elle appartient au membre droit d'une affectation, si elle apparaît comme indice d'un tableau ou comme paramètre d'une instruction d'écriture. Le flot de données est ainsi modélisé.

4.4. Le test fonctionnel (boîte noire)

Au contraire du test structurel (boite blanche), cette technique de test ne s'intéresse pas à l'aspect implémentation du code mais uniquement à son aspect extérieur. C'est-à-dire qu'on va tester le programme à travers ses interfaces et la sélection des jeux de test va s'effectuer à partir des documents de spécification et de la conception. On s'attache à vérifier le comportement réel du logiciel par rapport à son comportement spécifié. Les données de ce type de test basé sur l'interface sont générées à partir des domaines des données d'entrées et de sorties. On choisit comme données de test des données qui couvrent les extrémités du domaine (c'est le test aux limites) ainsi que des données comprises dans ce domaine. On suppose ici que le comportement du logiciel peut se déduire d'un ensemble restreint représentatif des entrées. Dans le cas d'entrées multiples, il faut choisir des combinaisons de tous les représentants de ces entrées. Cela peut générer de très nombreux cas de tests. Parmi les techniques du test fonctionnel on trouve les suivantes :

4.4.1. Analyse partitionnelle

Fondée sur les spécifications d'un programme, la méthode de l'analyse partitionnelle [Ost88] établit des classes équivalentes (partitions). Les données de test se baseront sur le choix d'un représentant quelconque de chaque classe. De manière générale, pour construire les différentes classes d'équivalence, nous utilisons les spécifications pour en extraire deux types de matières premières: Les domaines de valeurs d'entrée (par exemple : telle variable est une date, telle autre est une personne, un fichier, etc.) et l'ensemble de fonctions qui devront être réalisées par le programme en question (par exemple : on ouvre un fichier, on calcule une date, on estime une moyenne, etc.). On choisit un représentant de chaque classe issue du partage des domaines des définitions des données et/ou un représentant de chaque classe concernée par chacune des fonctions réalisées. On dit alors que les classes créées sont fonctionnellement équivalentes. L'ensemble des classes doit constituer un recouvrement de l'espace total des valeurs.

4.4.2. Test aux limites

Le test aux limites [Wei88] est considéré comme l'une des méthodes fonctionnelles les plus efficaces. En effet, la plupart des programmeurs ont probablement constaté que, souvent, les erreurs sont dues au fait qu'ils n'avaient pas prévu le comportement d'un programme pour des valeurs limites (valeurs très élevées, valeurs nulles d'un indice, valeurs négatives). Les tests aux limites couvrent une large gamme d'erreurs, car les limites forment l'hôte naturel préféré de la plupart des défauts. Leur principal incommodité est surtout la difficulté (et peut être l'impossibilité) de formalisation de la notion de limite et de marginalité, ce qui explique sans doute leur caractère souvent intuitif et heuristique.

4.4.3. Graphes cause-effet

Cette méthode consiste à élaborer un réseau qui relie les effets d'un programme (sorties) aux causes (entrées) qui sont à leur origine [Mye04]. Ce réseau est constitué par des nœuds et des segments qui en symbolisent les liaisons logiques. La construction du réseau se base sur quatre type de symboles usuels exprimant les interdépendances des effets aux causes (identité, négation, OU logique et ET logique). La méthode de test par graphes cause-effet peut s'avérer une méthode de test fonctionnel très complète et très précise. Le fait qu'elle utilise un langage de représentation très formalisé rend possible l'automatisation de certaines de ses phases. Cependant, bien que l'utilisation attentive de cette méthode permette la production de jeux de test efficaces, son application pratique est en général très difficile. En particulier, les graphes peuvent devenir très complexes quand une fonction fait intervenir un grand nombre de causes.

4.5. Test de mutation

La génération d'un ensemble de cas de test initiaux est simple, le problème se pose généralement lors de l'amélioration de la qualité de cet ensemble qui demande un effort beaucoup plus important. L'analyse de mutation proposée par [Woo90] est la technique principale utilisée pour la validation et l'amélioration de données de test. Elle est fondée sur l'injection de fautes (différents types d'erreurs de programmation). L'efficacité de techniques ou méthodes de test est mesurée par la connaissance du nombre et de l'emplacement précis des erreurs dans le logiciel. Le premier pas de cette technique consiste à créer un ensemble de versions erronées du programme sous test, appelées *mutants* (copie exacte du programme sous test dans lequel une seule erreur simple a été injectée, générés automatiquement à partir de la définition d'un ensemble d'opérateurs de mutation (différents types d'erreur, exemple :

changement de signe des constantes, changement de sens d'une inégalité, etc.) et à exécuter un ensemble de cas de test sur chacun de ces programmes erronés. Le résultat de l'exécution des cas de test avec les mutants permet, d'une part, d'évaluer l'efficacité de l'ensemble de cas de test par la proportion de mutants détectés, d'autre part, l'analyse des erreurs qui n'ont pas été détectées permet de guider la génération de nouveaux cas de test. Ceci consiste à couvrir les zones où se situent ces erreurs, et à générer des données de test spécifiques pour couvrir les cas particuliers. L'analyse de mutation a d'abord été conçue pour le test de logiciels procéduraux, mais depuis plusieurs années, des travaux se sont intéressés à cette technique dans le cadre de logiciels OO [Ale02], [Che01], [Gho00]. Certains proposent de nouveaux opérateurs de mutation fondés sur les mécanismes spécifiques aux langages orientés objet [Ale02], [Che01] alors que d'autres utilisent cette technique pour valider des cas de test d'intégration [Gho00]. Enfin, dans [Che01] les auteurs détaillent une technique de génération automatique pour des mutants de programmes JAVA fondée sur le mécanisme d'introspection du langage.

4.6. Test aléatoire

Dans toutes les techniques exposées précédemment, les critères de sélection des différents jeux de test étaient déterministes. Dans l'approche de test aléatoire [Goo75], la génération se fait d'une manière probabiliste. Dans l'utilisation la plus répandue, ce processus se base sur un échantillonnage uniforme des domaines de définition des paramètres d'entrées du programme. Les méthodes de sélection de jeux de tests aléatoires sont donc à priori très simples, sous réserve de disposer d'une caractérisation de l'ensemble des entrées acceptables du logiciel testé. Lorsque la loi de probabilité des entrées lors de l'utilisation effective du produit logiciel est connue, elle peut être utilisée pour guider la sélection. De plus, il est possible de définir un modèle de croissance de fiabilité reposant sur cette loi. Ce dernier permet alors de fournir un critère d'arrêt du test. L'avantage d'une telle technique est évidemment le fait que la procédure de génération de jeux de tests est simplifiée et peut être facilement automatisable. L'autre avantage, consiste en l'objectivité des cas de tests produits contrairement aux approches déterministes. Néanmoins, il est impossible de produire au hasard des combinaisons d'entrées qui sensibilisent des comportements très spécifiques pour des programmes de taille industrielle.

Dans ce qui suit, nous aborderons le test agent, nous étudierons les nouveaux défis introduits dans ce paradigme de programmation, et nous citerons quelques travaux réalisés au test agent.

5. Test des systèmes multi-agents

Les systèmes multi-agents ont envahi des domaines d'application de plus en plus critiques tels que les secteurs financiers, les transports, les services publics, l'aérospatiale et même le secteur militaire où les risques d'erreurs peuvent engendrer des défaillances catastrophiques. Ceci nous oblige à accorder plus d'importance au test des systèmes multi-agents qui jusque-là n'a reçu que très peu d'intérêt de la part des chercheurs, comparé à l'intérêt accordé aux premières phases des différents cycles de développement agent. En effet, comme déjà signalé, la plus grande majorité des contributions dans ce domaine était plutôt orientée sur les architectures, les protocoles, les infrastructures de messagerie et les interactions inter et intra agents [Fer98].

Lorsqu'un logiciel est à base d'agents, la difficulté est d'autant plus conséquente que pour un produit logiciel développé selon une approche conventionnelle ou orientée objet. Les raisons sont multiples:

- L'évolution imprédictible du comportement des SMA complique la phase de test car ceci nous oblige à tester, même, le comportement qui a évolué d'une façon imprédictible et émergente.
- Une complexité croissante liée au caractère distribué des applications constituées de plusieurs agents s'exécutant de manière autonome et concurrente [Hou11].
- Une grande quantité de données manipulées par ces agents chacun ayant ses propres objectifs et ses propres buts [Hou11].
- L'effet non reproductible, qui signifie qu'il n'est pas garanti que deux exécutions du système avec les mêmes données en entrée aboutissent à un état non identique. Comme conséquence directe, la localisation d'une erreur peut s'avérer très ardue, des fois impossible, du fait qu'il n'est pas probable de reproduire l'exécution ayant révélé une erreur [Hug04].
- Ils sont également non déterministes, puisqu'il n'est pas possible de déterminer a priori toutes les interactions d'un agent pendant son exécution [Hou11].

- Les agents sont autonomes et coopèrent avec d'autres agents, ainsi ils peuvent fonctionner correctement seuls mais inexactement dans une communauté [Hou11].
- Un agent mobile est une classe particulière d'agent avec la capacité pendant l'exécution d'émigrer d'une place à l'autre où il peut reprendre son exécution. Cette mobilité représente de nouveaux défis [Hou11].

Il est clair, à ce stade, qu'il n'est pas possible d'appliquer directement les techniques de test traditionnel pour un système multi-agents. En conséquence, le test des SMA demande de nouvelles méthodes de test traitant leur nature spécifique.

5.1. Les niveaux de test agent

Le test des SMA se compose de cinq niveaux [Ngu08a] et [Mor09] : unité, agent, intégration, système, et acceptation. Les objectifs de test et les activités de chaque niveau sont décrits comme suit :

- **Test unitaire**

  Consiste à tester toutes les unités qui composent un agent, y compris les blocs de code, l'implémentation des unités d'agent comme les buts, les plans, base de connaissance, moteur de raisonnement, spécification des règles, et ainsi de suite, et s'assurer qu'elles fonctionnent comme prévu.

- **Test d'agent**

  Consiste à tester l'intégration des différents modules à l'intérieur d'un agent ainsi que la possibilité des agents d'atteindre leurs buts dans leur environnement.

- **Test d'intégration ou de groupe**

  Après le test d'un agent d'une façon individuel, il est indispensable de tester son intégration avec les agents en phase de développement. La stratégie d'intégration dépend de l'architecture du système multi-agents où les dépendances entre agents sont exprimées en termes de communications avec parfois des interactions de médiations environnementales. Deux questions s'imposent lors de ce test: La première est de savoir comment pouvons-nous assurer que les agents au sein de cette société travaillent ensemble comme prévu. La seconde est de savoir comment garantir que le travail qui en résulte est bien celui attendu. Lors d'un test de la société d'agents,

la validation des résultats de l'ensemble des agents est exercée et la bonne intégration des différents agents est vérifiée. En d'autres termes, il s'agit de vérifier que chaque agent dans la société reçoit le bon message depuis le bon agent, fournit la bonne réponse, et interagit correctement avec l'environnement [Rou02]. En outre, un tel test peut aussi impliquer l'assurance que l'objectif de la société dans laquelle les agents sont en interaction est pleinement atteint. Les erreurs qui peuvent être observées au cours du test d'une société d'agents sont dues soit à une mauvaise communication, soit au contenu des messages agent, soit à la présence d'inter-blocages dans les échanges de messages. Une dernière question qui doit être considérée lors de ce test est celle ayant trait à l'évolutivité du système. Plus la société d'agents est large, plus il est difficile de tester adéquatement toutes ses fonctionnalités [Rou02].

- **Test de système**

  Le test du niveau système implique l'assurance que tous les agents dans le système opèrent selon les spécifications. Dans ce niveau de test on se concentre sur :

  - Le test du fonctionnement du système dans l'environnement.
  - Le test des propriétés émergentes et macroscopiques prévues du système dans son ensemble.
  - Le test des propriétés de qualité qui doivent être atteintes par le système, comme l'adaptation, la franchise, la tolérance aux pannes et la performance.
  - On peut également, au niveau du test système, vérifier les principaux types d'interactions agents (coopération, négociation et coordination) ou faire appel à d'autres types de tests tels les tests de performance, de conformité, les tests de chargement et/ou de stress, etc.

- **Test d'acceptation**

  Consiste à tester le SMA dans l'environnement de l'exécution du client et vérifier qu'il atteint lesbuts des dépositaires avec la participation de ces derniers.

5.2. Les travaux qui existent sur le test agent

Les premiers travaux sur le test des SMA sont axés sur la définition des outils de débogage pour améliorer les plates-formes de développement des SMA [Gut01], [Cai04], [Pou09]. Les approches de tests structurés ont été proposées plus récemment, pour compléter l'analyse et les méthodologies de conception. Dans ce qui suit, nous examinons les travaux récents et actifs sur le test des SMA, en respectant les catégories précédentes. Cette classification est prévue pour faciliter seulement la compréhension des travaux de recherche dans ce domaine. Nous récapitulons d'abord les contributions des travaux qui se concentrent principalement sur un niveau particulier de test, puis les travaux qui concernent plus d'un niveau sont présentés.

a. Niveau unitaire

- **Zhang et al [**Zha09a**]**

  Ce travail est divisé en trois travaux [Zha07] [Zha08] [Zha09b], il présente un cadre pour le test unitaire automatisé des agents.

  - Dans [Zha07], les auteurs ont proposé une approche de test basée sur les modèles issus de la phase de conception de la méthodologie de conception des agents Prometheus [Pad04]. Le résultat a été la création de l'outil PDT (Prometheus Design Tool) [Zha07].
  - Dans [Zha08], les unités de base à tester sont identifiées comme des événements, des plans et des croyances. Les événements sont testés pour la couverture de : Est-ce que cet évènement sera traité et y a-t-il des plans multiples qui peuvent gérer cet événement dans la même situation ? Les plans sont testés pour savoir s'ils se déclenchent au moins dans certaines situations (mais pas tous, s'il ya une condition spécifique à son applicabilité), est-ce que le plan est complet, est-ce que le plan répond à l'événement comme indiqué dans la spécification. Le test des croyances se limite à vérifier si les domaines de données, comme indiqué dans la conception, sont mis en œuvre.
  - Dans [Zha09b], les détails de la façon dont les variables doivent être spécifiées, extraites et les valeurs attribuées pour la création des cas de test sont présentés. En outre, afin d'exécuter les cas de test, l'environnement du système doit être installé. Ce travail porte sur l'introduction des procédures

d'initialisation et des agents de simulation (*Agent Mock)* pour simuler d'autres agents / systèmes.

- **Ekinci et al. [**Eki09**]**

  Ce travail considère les buts d'agent comme les plus petites unités testables dans le SMA. Il propose pour tester ces unités un moyen de *buts de test*. Chaque but de test est conceptuellement décomposé en trois sous-buts : *setup, goal under test, et assert*. Les premiers et les derniers buts préparent des conditions préalables et vérifient les post conditions respectivement, tout en testant le but sous test.

b. Niveau d'agent

- **Lam et Barber [**Lam05**]**

  Ce travail propose un processus semi-automatisé pour comprendre le comportement d'agent logiciel. L'approche imite ce qu'un utilisateur humain, peut être un testeur, fait dans la compréhension de logiciel : construction et raffinage d'une base de connaissance au sujet des comportements des agents, et de l'employer pour vérifier et expliquer les comportements des agents au temps d'exécution.

- **Nunez et al. [**Nun05**]**

  Ce travail présente un cadre formel pour spécifier le comportement des agents de *commerce électronique* autonomes. Les comportements désirés des agents sous test sont présentés au moyen d'un nouveau formalisme, appelé la machine d'états de service qui incarne les préférences des utilisateurs dans ses états. Deux méthodologies de test, l'une active, l'autre passive, sont proposées pour vérifier si une exécution d'agent spécifique se comporte comme prévu (c.-à-d., test de conformité). Dans l'approche de test actif, les auteurs emploient, pour chaque agent sous test, un testeur (un agent spécial) qui prend les spécifications formelles de l'agent pour faciliter l'arrivée à un état spécifique. La trace opérationnelle de l'agent est alors comparée aux spécifications afin de détecter des défauts. D'autre part, les auteurs proposent également d'employer le test passif, dans lequel les agents sous test sont observés seulement, non simulés comme dans le test actif. Les traces invalides, si elles existent, sont alors identifiées grâce aux spécifications formelles des agents.

- **Coelho et al. [**Coe06**]**

  Ce travail propose un cadre pour le test unitaire des SMA basé sur l'utilisation des agents Mock. Leur travail se concentre sur le test des rôles des agents. Les agents

Mock qui simulent des vrais agents dans la communication avec l'agent sous test ont été implémentés manuellement; chacun correspond à un rôle d'agent.

- **Nguyen et al [**Ngu08b**]**

  Ce travail est basé sur les ontologies d'interaction d'agent qui définissent la sémantique des interactions d'agent, les auteurs : (i) produisent des entrées de test ; (ii) guident l'exploration de l'espace d'entrée pendant la génération ; (iii) vérifient les messages échangés entre les agents avec respect des interactions définies dans l'ontologie. Un ensemble de règles de génération a été défini et automatisé pour tester un agent particulier intensivement par un grand et divers nombre de cas de test.

- **Nguyen et al [**Ngu12**]**

  L'autonomie d'agent rend le test plus dur : les agents autonomes peuvent réagir de différentes manières aux mêmes entrées dans le temps, parce qu'ils ont des buts et des connaissances qui changent. Le test des agents autonomes exige un procédé qui couvre un large éventail de contextes de cas de test, et cela peut rechercher le plus exigeant de ces cas de test. Nguyen et autres [Ngu12] présentent et évaluent une approche pour tester les agents autonomes qui emploient l'optimisation évolutionnaire pour produire des cas de test exigeants. Dans ce travail, les auteurs ont proposé une manière systématique d'évaluer la qualité des agents autonomes. D'abord, des exigences de dépositaire sont représentées comme qualité de mesures, et les seuils correspondants sont employés comme critères de test. Les agents autonomes doivent répondre à ces critères afin d'être fiables. Des fonctions de forme physique qui représentent des objectifs de test sont définies en conséquence, et guident cette technique de génération de test évolutionnaire pour produire des cas de test automatiquement.

c. Niveau d'intégration

- **Serrano et Botia [**Ser08**], Serrano et al. [**Ser09**]**

  Une des issues quand on teste les SMA est leur évolutivité, en raison du nombre d'agents et plus spécifiquement le nombre important d'interactions qui peuvent surgir. Ceci le rend très difficile pour appliquer des méthodes de test classiques. ACLAnalyser aborde ces issues en appliquant des techniques d'exploitation de données (Data mining) sur les notations des exécutions de SMA.

d. Multi niveaux de test

- **Tiryaki et al. [**Tir07**]**

  Ce travail propose une approche test-conduite le développement des SMA qui soutient la construction itérative et par accroissement des SMA. Un cadre de test appelé SUnit, construit sur JUnit [Gam00] et Seagent [Dik05], a été développé pour soutenir l'approche. Le cadre permet des écritures de tests pour des comportements d'agent et des interactions entre les agents.

- **Nguyen et al. [**Ngu10**]**

  Ce travail propose une méthodologie de test complète, appelée GOST (Goal-Oriented Software Testing). GOST permet la génération des suites de test à tous les niveaux de test. Elle fournit un modèle de processus de test qui apporte les raccordements entre les buts et les cas de test explicites et une manière systématique de dériver des cas de test de l'analyse des buts. Ceci peut aider à découvrir des problèmes tôt, évitant d'implémenter des spécifications incorrectes. En outre, GOST est supporté par un outil qui soutient la génération des cas de test, la spécification, et l'exécution.

5.3. Discussion

Malgré l'intérêt de la recherche pour les systèmes multi agents holoniques, le développement des applications basées sur ces systèmes demeure peu solide. L'activité de test, qui représente une tâche importante dans leur assurance de la qualité, n'est pas bien couverte. En effet, il n'existe, dans la littérature, que très peu de propositions concernant le test de ces systèmes. Ceci laisse en suspens plusieurs questions sans réponse, par exemple : "*comment peut-on tester l'évolution des systèmes multi-agents, afin d'assurer qu'elle réponde à leurs spécifications fonctionnelles ?* ", demeure encore un objectif de recherche.

Les approches proposées jusqu'à maintenant pour le test des systèmes multi-agent ont certes apporté des éléments de réponse importants à différents problèmes pour les SMA. Cependant, ces approches ne tiennent pas en compte des spécificités des SMA holoniques, parce qu'elles sont, principalement, conçues pour les agents qui sont définis comme des entités atomiques alors que les agents holoniques sont, en fait, définis comme agents composés d'agents. En outre, ces approches ne sont pas appropriées pour couvrir le problème lié à l'évolution du comportement et de la structure des agents holoniques qui émergent avec le temps, suite à l'introduction (à tout moment) de nouvelles interactions entre les membres de

l'agent holonique, et celle de la nature hiérarchique de l'agent holonique qui complique la génération des cas de test.

Une approche nouvelle qui prendrait en compte les caractéristiques de l'agent holonique, et les défis relevés dans le cadre du test de cet agent, s'impose d'elle-même. Nous, nous proposons, dans le cadre de notre travail, de définir une nouvelle approche qui s'accorde mieux aux caractéristiques et aux défis du test de l'agent holonique.

6. Conclusion

Ce chapitre montre que le test des SMA demande de nouvelles méthodes capables de prendre en compte leur nature spécifique. Les méthodes proposées, à ce jour, dans la littérature pour le test des SMA ne prennent pas en compte les spécificités des SMA holoniques, notamment leur comportement et leur structure qui évoluent avec le temps. Dans le chapitre suivant, nous étudierons l'évolution du comportement et de la structure de l'agent holonique ainsi que l'effet de cette évolution sur le processus de test et ce, dans le but de développer une approche plus adéquate, capable de mieux s'adapter aux spécificités des SMA holoniques.

Chapitre 3

Évolution de l'agent holonique

1. Introduction

L'agent holonique se caractérise par un comportement et une structure qui se développent avec le temps. Dans ce chapitre, nous étudierons le comportement et la structure de l'agent holonique, les différentes techniques de leur développement, ainsi que l'effet de ce développement sur le processus de test. Le but étant d'arriver à une approche de test adaptable avec le comportement spécifique de l'agent holonique.

2. Etude de la structure et du comportement des agents holoniques

Le comportement global de l'agent holonique résulte de la combinaison des rôles, des connaissances et des capacités dont il dispose à un instant donné. L'aptitude de l'agent holonique pour obtenir de nouveaux rôles, de nouvelles capacités pendant sa vie, fait que le comportement et la structure globale de ce dernier se développent avec le temps et lui confèrent la possibilité de satisfaire de nouveaux objectifs et de s'adapter aux changements de l'environnement.

Ce développement peut être de deux manières :

- **Montante (bottom-up)** où un ensemble de holons s'associent pour créer un nouveau super-holon en charge de satisfaire un objectif donné.

- **Descendante (top-down)** où un holon dont les tâches deviennent trop complexes décide de créer un ensemble de groupes de production pour exécuter ses tâches et distribuer les coûts de calcul. Pour peupler ces différents groupes, le super-holon pourra ensuite recruter des membres disposant des capacités requises ou créer de nouveaux holons.

Cette section décrit l'ensemble des mécanismes de développement du comportement et de la structure d'un super-holon.

2.1. La publication des services

Un super holon, en plus de ses capacités de base, est apte à posséder de nouvelles capacités, réalisées par un service fourni par ses membres du niveau inférieur [Rod06]. La clef d'un tel mécanisme repose sur le fait qu'un groupe d'agent est en mesure de fournir un service résultant de la collaboration entre les différents membres du groupe. Le fait pour un super holon de posséder une nouvelle capacité, implique pour ce dernier la possibilité de jouer un nouveau rôle qui lui était inaccessible au départ. Ceci se traduit par l'introduction de nouvelles interactions entre le super holon et d'autres agents qui jouent des rôles dans le même groupe où ce dernier a obtenu un rôle. Cela implique une évolution dans le comportement et la structure de l'agent holonique (car le comportement des membres de nivaux *i* sont définis par les interactions de leur membres de nivaux *i-1*).

L'exemple présenté dans la figure 3.1 illustre le fait qu'un groupe peut réaliser une capacité via un service.

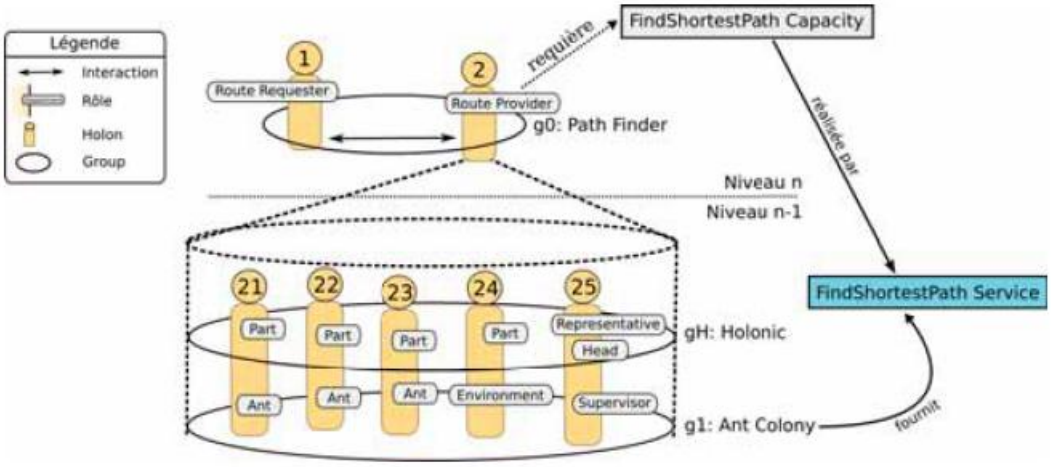


Figure 3.1 : Structure d'un super-holon exploitant la capacité collective [Rod06].

Dans cet exemple, le holon H2 a tenté de jouer le rôle *TrouverPlusCourtChemin*mais il lui manque la capacité nécessaire pour cela. Néanmoins, la collaboration entre ses membres du groupe g1 a permis de publier un service qui lui a donné la possibilité d'acquérir la capacité qui lui manquait pour jouer le rôle.

Un super-holon peut donc réaliser une capacité, soit en obtenant directement une réalisation sous forme d'algorithmes, soit en intégrant un groupe (de production) capable de fournir un service qui réalise la capacité dont il a besoin. Un groupe peut fournir un service de différentes manières:

- **Atomique** : le service est fourni par l'un des rôles de l'organisation. Le rôle possède une capacité disposant d'une description compatible avec celle du service.
- **Composé** : le service est obtenu depuis les interactions d'un sous-ensemble des rôles de l'organisation en suivant un protocole connu. Ceci est un scénario de composition de service où le plan d'obtention est a priori connu et où les performances peuvent être connues et assurées.
- **Émergent** : le service est obtenu grâce aux interactions d'un sous-ensemble de rôles de l'organisation, mais le plan d'obtention n'est pas connu a priori. Le résultat du service est émergent. Dans ce dernier cas, la gestion du service fourni par le comportement global de l'organisation est généralement assurée par l'un des rôles de l'organisation, comme c'est le cas dans la colonie de fourmis avec le rôle Supervisor.

2.2. Le recrutement

Les objectifs et les tâches d'un super-holon peuvent évoluer au cours de sa vie. Il peut par conséquent avoir besoin de recruter de nouveaux membres pour satisfaire de nouveaux objectifs. À un niveau d'abstraction donné, le holon est considéré comme un ensemble de groupes et donc de membres en interaction. Cependant, depuis l'extérieur du super-holon, ces membres sont invisibles par le reste du système. Aucun holon non membre ne peut directement interagir avec les membres, excepté avec le ou les représentants de ceux-ci. Un holon peut demeurer seul, et dans ce cas, ces décisions ne sont pas restreintes et ne dépendent que de ses objectifs propres. Un holon demeure généralement dans cet état tant qu'il est satisfait. En accord avec ses besoins, un holon peut décider de rejoindre [Rod06] un super-holon existant afin de satisfaire des objectifs communs. Pour ce faire, il doit demander son admission à l'un des représentants des membres, lequel sert d'interface entre la communauté

du super-holon et le candidat à l'intégration. Les Heads du super-holon décident ensuite, en fonction des capacités du candidat. Si son admission est acceptée, le holon devient un membre à part entière. Pour gérer ce processus d'intégration de nouveaux membres, une organisation spécifique, nommée *Merging*, a été créée. Cette organisation définit deux rôles : *Representative* joué par l'un des représentants des membres du super-holon, et *Stand-Alone* joué par le candidat au recrutement. Lorsqu'un candidat demande son intégration à un super-holon existant, cette organisation est instanciée sous forme de groupe. Le candidat obtient le rôle *Stand-Alone* et l'un des représentants du super-holon intègre le groupe, lequel constitue le support nécessaire aux négociations de recrutement.

La figure 3.2 présente un exemple d'instanciation de l'organisation *Merging* au sein d'un super-holon.

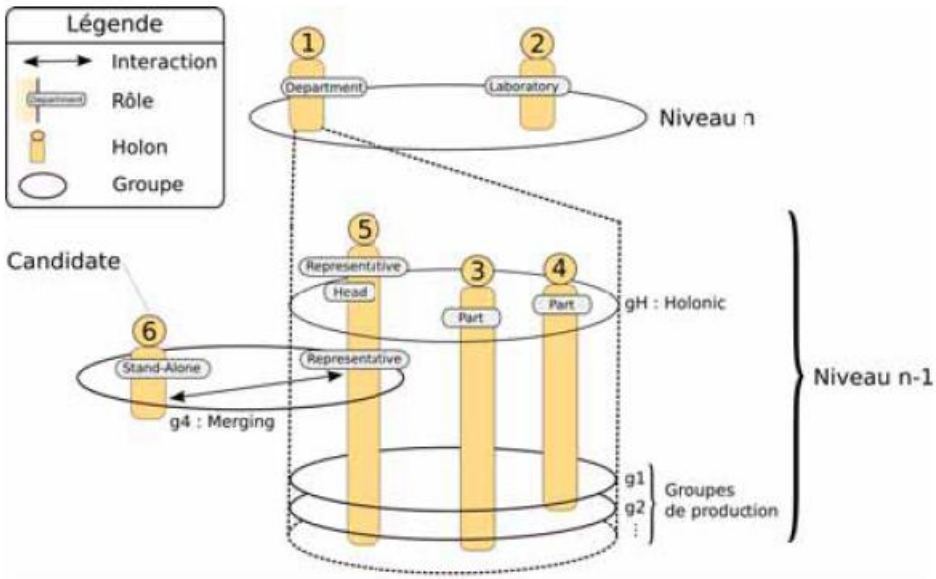


Figure 3.2 : L'intégration de nouveaux membres au sein d'un super-holon [Rod06].

Le nouveau membre recruté intègre le groupe holonique du super-holon ainsi que les groupes de production en lui conférant des rôles. Une fois que le membre recruté obtienne des rôles du groupe de production du super holon, il devient capable *d'interagir* directement avec les membres de celui-ci. Ceci se traduit par un effet sur le comportement et la structure du super holon et par voie de conséquence sur le comportement et la structure de l'agent holonique. Le membre recruté met tout ou partie de ses compétences à la disposition du super holon qui l'a recruté, ce qui augmente la possibilité de la publication d'un service fourni grâce à la collaboration entre le membre recruté et les membres du super holon qui existaient déjà.

Grâce à ce service, le super holon acquiert la capacité de jouer un nouveau rôle, ce qui se traduit par l'introduction de nouvelles interactions entre ce super holon et les agents qui jouent des rôles dans le même groupe où il a obtenu ce rôle.

2.3. L'acquisition dynamique de capacité

Un agent possède à l'origine un ensemble de capacités basiques qu'il peut faire évoluer dynamiquement. Pour acquérir une nouvelle capacité, un holon peut intégrer un nouveau groupe fournissant un service capable de la réaliser. La procédure d'acquisition de nouvelles capacités [Rod06] par l'intégration de nouveaux groupes peut être résumée par les étapes suivantes :

- Le holon compare les capacités fournies par les différentes organisations connues dans le système avec celles requises par le rôle qu'il désire jouer. Pour effectuer cet appariement ("matchmaking process") dans un système ouvert, un langage commun de description des capacités et des services est requis.
- Le holon doit choisir, parmi les organisations identifiées, celle qui lui correspond le mieux. Ce choix est en fait essentiellement guidé par la comparaison des capacités requises par les rôles de l'organisation et les capacités possédées par les membres actuels du holon.
- L'organisation choisie sera instanciée sous forme de groupe interne au holon. Les rôles de ce groupe sont distribués aux membres du holon. Si l'un des rôles ne peut être joué (par manque de membres, ou de capacités requises), une procédure de recrutement de nouveaux membres est lancée (ou éventuellement une procédure d'acquisition de capacité pour l'un des membres). De nouvelles interactions (définies par le groupe intégré) sont ainsi introduites entre les membres qui ont obtenu des rôles du groupe intégré.
- Lorsque chacun des rôles, définis dans le groupe nouvellement créé, est associé à un agent (en accord avec le nombre d'instances requis pour chaque rôle), le super-holon est alors en mesure d'obtenir la nouvelle capacité et ainsi acquérir le rôle qu'il convoitait. Ceci se traduit par l'introduction de nouvelles interactions entre le super holon et d'autres agents qui jouent des rôles dans le même groupe dans lequel il a obtenu le rôle.

La figure 3.3 illustre ce scénario d'acquisition sur l'exemple de la capacité Trouver-PlusCourtChemin.

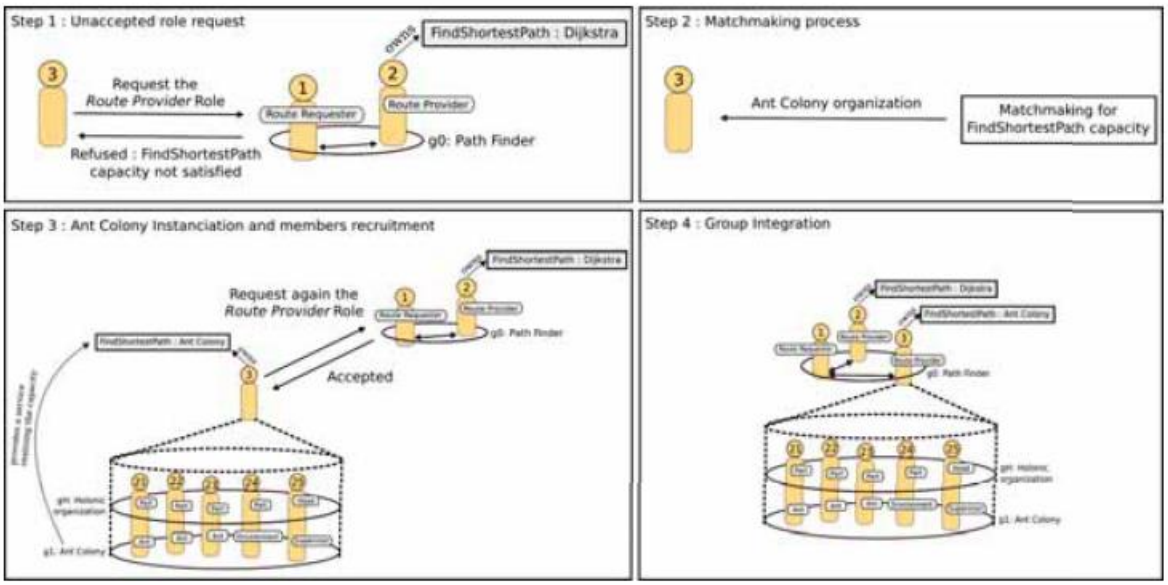


Figure 3.3 : Processus d'acquisition dynamique de capacité [Rod06].

Le groupe g0, instance de l'organisation *Détermination de route dans un graphe*, contient déjà deux holons H1 et H2 ; le premier jouant le rôle *Route Requester* et le second *Route Provider*. Le holon H2 possède la réalisation de la capacité *TrouverPlusCourtChemin* basée sur l'algorithme de Dijkstra. Le holon H3 souhaite intégrer le groupe et jouer le rôle *Route Provider* ; or il ne possède pas la capacité (étape 1). Il dispose alors de trois possibilités, soit acquérir une réalisation de type Dijkstra, soit recruter un nouveau membre qui possède déjà cette capacité, ou enfin intégrer une organisation capable de réaliser la capacité requise. Il est supposé ici que l'agent opte pour cette dernière alternative. L'organisation *Colonie de fourmis* est identifiée comme candidate via le processus d'appariement (étape 2). Elle est instanciée sous forme de groupe et celui-ci est intégré au holon H3 (étape 3). Disposant finalement de la capacité requise, H3 peut alors intégrer le groupe g0 et jouer le rôle désiré (étape4).

3. L'effet du développement du comportement et de la structure de l'agent holonique sur le processus de test

Le comportement et la structure de l'agent holonique évoluent avec le temps suite à l'introduction des nouvelles interactions entre les membres de l'agent holonique dès qu'il y a une obtention de rôle (cf: paragraphes 2.1, 2.2 et 2.3). Cette réalité rend l'application du test basé modèle limitée. En effet, pour tester l'agent holonique en nous basant sur son modèle comportemental, nous avons besoin d'un langage de modélisation pouvant supporter l'évolution du comportement et de la structure de l'agent holonique qui est généralement

imprévisible. Ce type de langage étant difficile à trouver, la première solution serait de laisser l'agent holonique se développer, de prendre la dernière version (la plus évoluée) et d'appliquer le test en nous basant sur le modèle associé à cette dernière version. Cependant, si on prend en considération que les membres de l'agent holonique peuvent obtenir et libérer des rôles pendant le développement et selon les besoins, cette solution sera rejetée car avec la libération de rôles, certaines interactions entre les membres disparaissent et ne seront donc pas prises en compte si le test est appliqué uniquement à la dernière version. Ceci rend le test peu fiable, puisqu'il sera difficile de connaitre les interactions disparues ainsi que le stade de développement dans lequel elles ont été créées. Il sera, en outre, difficile de savoir si les interactions ayant disparues sont à l'origine d'erreurs à même de biaiser le développement du comportement et de la structure de l'agent holonique. Ceci nous oblige à appliquer le test basé modèle à chaque apparition des interactions qui est justement l'idée principale de notre approche.

4. Conclusion

L'introduction de nouvelles interactions entre les membres de l'agent holonique, à tout moment, entraine l'acquisition de nouveaux comportements qu'il faut tester. De ce fait, si on se limite au test de la première version, avant le développement de l'agent holonique, un manque de confiance sera instauré vis-à-vis de tous les comportements acquis avec le temps. Si on attend jusqu'à la dernière version pour appliquer le test, nous risquons d'occulter des erreurs, crées pendant le développement et qui sont à l'origine de biais du développement du comportement de l'agent holonique, d'où la nécessité d'appliquer le test à chaque évolution. Dans le chapitre suivant, nous décrirons l'approche que nous proposons pour le test des agents holoniques en prenant en considérations ces spécificités.

Chapitre 4

Une nouvelle approche de test pour les agents holoniques

1. Introduction

L'activité de test représente une tâche importante dans le processus d'assurance qualité des SMA. Malgré l'évolution rapide des SMA, le test des SMA comme systèmes autonomes est encore un domaine clé ouvert. En fait, seules quelques propositions portant sur le test des SMA ont été proposées dans la littérature. Bien qu'ils aient permis de réels progrès dans le domaine du test des SMA, ils ne prennent pas en compte l'évolution des SMA. En outre, ils ne tiennent pas compte des spécificités des agents holoniques (par exemple, l'évolution de leur comportement, leur structure et leur nature hiérarchique).

Dans ce chapitre, nous proposons une nouvelle approche de test pour les agents holoniques pouvant s'adapter avec leurs comportements et leurs structures qui se développent avec le temps et avec leur nature hiérarchique.

2. Explication générale de l'approche proposée

Notre approche [Deh15] consiste à appliquer le test basé sur le modèle comportemental pour chaque nouvelle version développée de l'agent holonique et détecter à temps toute erreur, éventuelle, créée dans chaque nouvelle version (i.e. test par version). La question qui se pose et la suivante : Comment peut-on trouver les nouvelles versions à tester (détecter le

développement du comportement et de la structure)? Le comportement et la structure de l'agent holonique évoluent dans le temps en accord avec ses objectifs et ses besoins à travers l'obtention des rôles. Cette réalité, a fait germer, en nous, l'idée de le mettre dans des situations, pour la résolution desquelles, il se trouve contraint de se développer et d'obtenir de nouveaux rôles.

L'idée serait d'exécuter l'agent holonique sur des entrées générées automatiquement par des algorithmes génétiques et évaluées par une fonction de fitness. L'exécution de l'agent holonique avec ces entrées va, graduellement, mettre ce dernier dans des situations où il rencontre de nouvelles entrées (générées par l'algorithme génétique) pour la résolution desquelles il se trouve contraint d'obtenir de nouveaux rôles. Ceci entraine un développement dans le comportement et la structure de l'agent holonique. La fonction de fitness proposée à une relation directe avec le nombre de rôles obtenus par les membres de l'agent holonique au cours d'exécution, où l'incrémentation observée dans la fonction de fitness reflète une évolution dans le comportement et la structure de l'agent holonique. La génération des populations des entrées avec lesquels l'agent holonique est exécuté est basée, à chaque fois, sur les populations qui ont causé un développement dans la génération précédente. Ceci permet la poursuite et l'approfondissement de la situation dans laquelle se trouve l'agent holonique. Ceci l'incite à obtenir tous les rôles nécessaires pour résoudre les entrées générés et dépasser la situation dans laquelle il se trouve.

La phase pendant laquelle l'agent holonique obtient de nouveaux rôles est appelée la phase de développement. Cette phase commence dès que l'agent holonique s'avère incapable de résoudre les entrées générées par l'algorithme génétique. Durant cette phase, on constate une incrémentation dans la fonction de fitness qui indique que l'agent holonique est entrain d'obtenir de nouveaux rôles. Cette incrémentation se poursuit jusqu'à atteindre un état de maximisation caractérisée par le fait que l'agent holonique devient capable de résoudre les entrées générées par l'algorithme génétique, ce qui indique que ce dernier a obtenu l'ensemble des rôles nécessaires qui lui permettent de résoudre ces entrées. Ceci implique la fin de cette phase et la détection d'une nouvelle version plus développée. Dans ce cas, le processus de recherche de nouvelle version s'arrête et un modèle comportemental associé à cette nouvelle version est généré (le modèle comportemental généré concernera uniquement la version à tester). On applique le test sur cette nouvelle version en se basant sur le modèle comportemental associé, on corrige les erreurs éventuelles et on revient au processus de recherche des nouvelles versions pour trouver une autre nouvelle version plus développée

(Figure 4.1). Cette méthode est appliquée de façon itérative jusqu'à l'obtention de la dernière version de l'agent holonique.

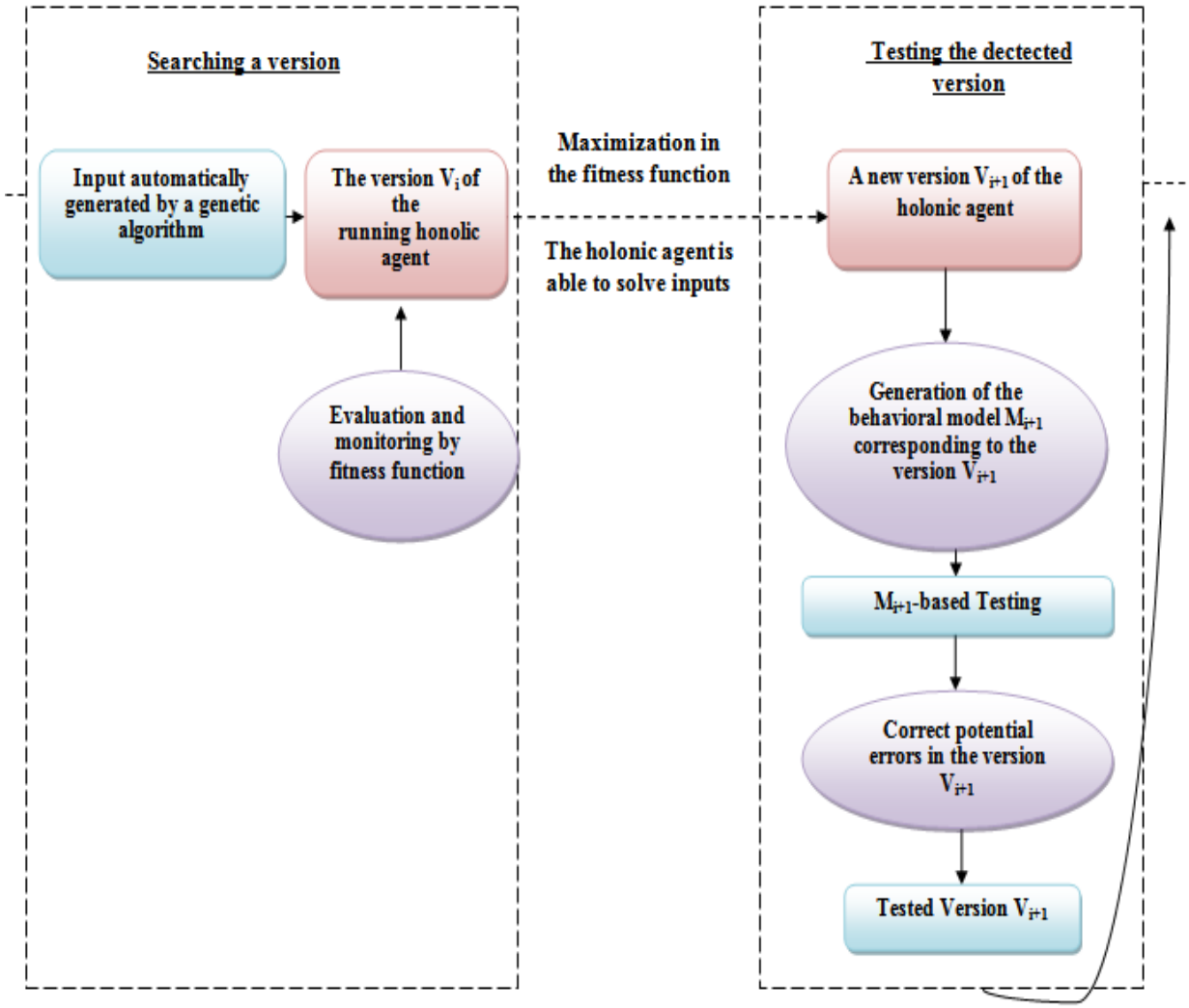


Figure 4.1: Méthodologie de notre approche.

L'approche débute par une version V_0, qui est la version primitive. On applique sur cette version un test basé sur son modèle comportemental (M_0) et on corrige les erreurs éventuelles rencontrées. Pour passer à la version V_1, la version V_0 sera exécutée avec des données générées par un algorithme génétique en prenant comme données initiales des données générées au départ de manière aléatoire.

Une fois une maximisation est détectée dans la fonction fitness, caractérisée par le fait que l'agent holonique devient capable de résoudre les entrées générées par l'algorithme génétique, on considère que nous sommes arrivés à la version V_1. Pour passer à la version V_2, la version V_1 sera exécutée avec des données générées par un algorithme génétique, en prenant comme données initiales les dernières entrées qui étaient la cause de l'apparition de la

version V_1. Ceci est admis si nous cherchons un développement dans la même direction sinon les entrées initiales seront modifiées dans le but d'avoir d'autres directions de développement, désirées. Pour cela nous fixons les entrées pour lesquelles l'agent holonique s'est déjà développé, ce qui permettra à l'algorithme génétique de se concentrer uniquement sur les autres entrées. De cette façon, l'arrivée aux autres directions et la détection de toutes les versions sera plus rapide avec une minimisation des calculs. La fixation des entrées pour lesquelles l'agent holonique s'est déjà développé n'est pas obligatoire dans le cas où la dépendance entre les entrées et les directions de développement n'est pas claire (ex : systèmes biologiques…). L'arrivée aux différentes directions sera, un peu plus longue, mais réalisable.

A partir de la version V_2, le même processus sera appliqué, jusqu'à l'obtention de la dernière version de l'agent holonique qui est caractérisée par le fait que la fonction de fitness ne change plus (Figure 4.2).

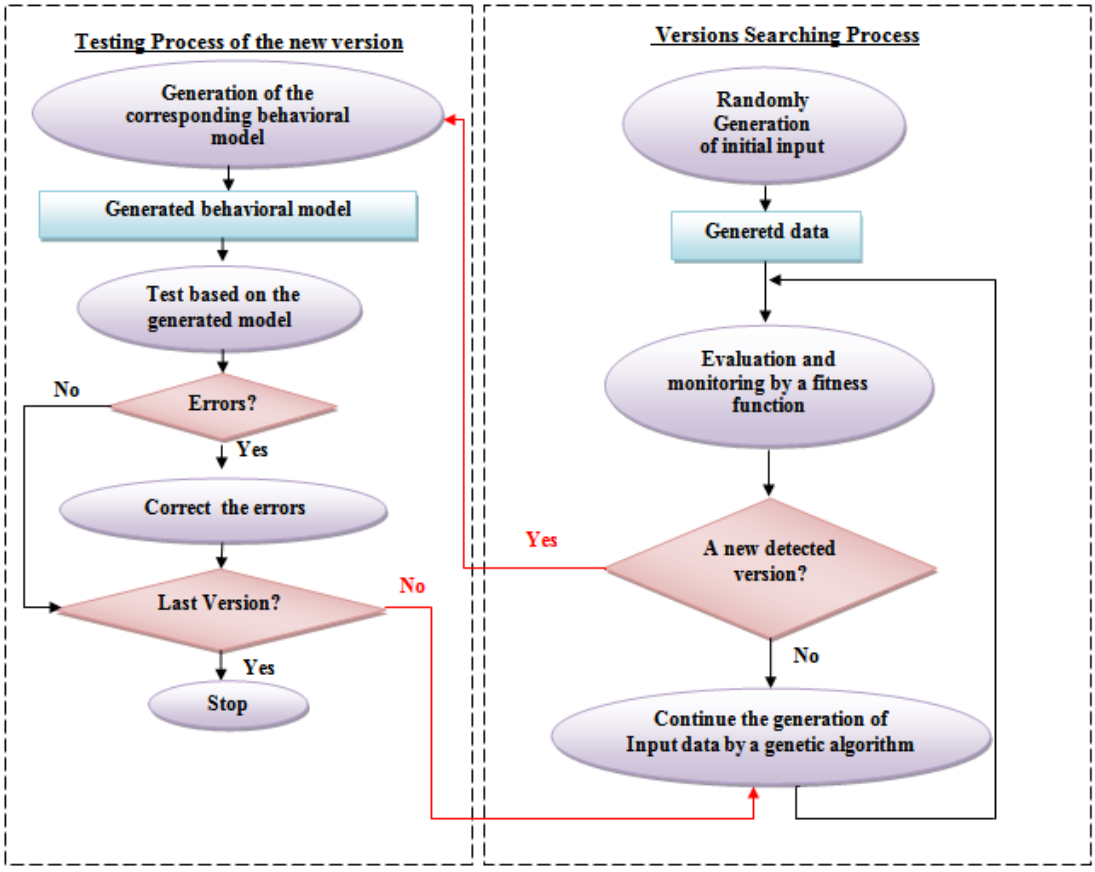


Figure 4.2: Les deux principaux processus de l'approche.

A ce stade, nous avons expliqué notre approche de façon globale. Elle contient deux processus qui s'exécutent de manière itérative à savoir : (i) La détection des nouvelles versions, (ii) Le test basé modèle de chaque nouvelle version détectée.

3. La détection d'une nouvelle version

Ce processus consiste à exécuter le programme par des données générées automatiquement par un algorithme génétique et évaluées par une fonction de fitness.

3.1. La fonction de fitness

La fonction de fitness est définie de façon récursive par un vecteur comme suit :

$$H = (f_1, f_2, \ldots f_i, \ldots . f_n) \,/\, 1 \leq i \leq n$$

- f_i représente la fonction de niveau *i*, qui permet de détecter tout changement survenant au niveau *i*.
- *n-1* représente le nombre des niveaux existant dans l'agent holonique H

$$f_i = (f_{i1}, f_{i2}, \ldots f_{ij}, \ldots, f_{im}) \,/\, 1 \leq j \leq m$$

- f_{ij} représente la fonction du membre *j* dans le niveau *i*; elle permet de détecter tout changement survenant au membre *j* de niveau *i*.
- *m-1*représente le nombre des membres dans le niveau *i*.

  $f_{ij} = nbr\_role$
- *nbr_role* représente le nombre des rôles obtenus par le membre *j* du niveau *i* à un moment donné.
- Lorsque l'agent *j* du niveau *i* obtient un nouveau rôle, le *nbr_role* de cet agent devient *nbr_rôle+1*et l'identifiant du rôle obtenu sera sauvegardé dans une table T_{ij} avec les autres identifiants des rôles de l'agent *j* de niveau *i*.
- Lorsque un agent libère un rôle, le *nbr_role* devient *nbr_rôle-1* et l'identifiant de ce rôle sera éliminé de la table.

La fonction de fitness est définie sous forme de vecteur dans lequel chaque membre de l'agent holonique est représenté par son nombre de rôles, parce que notre objectif est d'avoir un développement dans le comportement de l'agent holonique dans sa globalité quelque soit le membre à l'origine de ce développement. Nous avons évité de développer la fonction de fitness égale au nombre de rôles de chaque membre d'une façon individuelle qui nous aurait contraints d'attendre un développement de chaque membre. La définition de la fonction de fitness comme un vecteur nous permet d'avoir un double objectif. Il permet, d'une part, la détection du développement du comportement et de la structure de l'agent holonique et, d'autre part, la détermination exacte du niveau et du membre qui ont causé le développement.

Ceci nous permet de déterminer la différence entre deux versions successive: le delta entre $V_i$etV_{i+1} qui facilite désormais, la modélisation de la nouvelle version V_{i+1}.

3.2. Le delta

Lorsqu'on constate une maximisation dans la fonction de fitness caractérisée par le fait que l'agent holonique devient capable de résoudre les entrées générés par l'algorithme génétique, on analyse les résultats de la fonction de fitness pour déterminer les membres qui ont causé le développement ce qui nous permet de trouver le delta entre les deux versions. La modélisation de la nouvelle version se fera sur la base de la valeur du delta trouvé entre les deux versions. Delta (V_i, V_{i+1}) représente la différence entre deux versions successives V_i et V_{i+1} d'un point de vue interaction introduite et interaction disparue.

Les interactions introduites entre un agent *j* de niveau *i*, qui a obtenu un nouveau rôle dans la version V_{i+1} et les autres agents qui jouent des rôles dans le même groupe G_k où ce dernier a obtenu son nouveau rôle, sont représentés par des digrammes de séquences associés au groupe G_k où le rôle obtenu y est invoqué (les diagrammes de séquences sont définis dans la phase de description des scenarios d'interactions du processus ASPECS). Concernant les interactions disparues, entre un agent *j* de niveau *i*, qui a libéré un rôle dans la version V_{i+1}et les autres agents qui jouent des rôles dans le même groupe G_k où ce dernier a libéré un rôle, sont représentés par des digrammes de séquences associés au groupe G_k où le rôle libéré y est invoqué. Comme résultat, le delta (V_i, V_{i+1}) sera un ensemble de diagrammes de séquences représentant des interactions introduites et un ensemble de diagrammes de séquences représentant les interactions disparues.

4. Test de la nouvelle version

Une fois une nouvelle version est détectée, nous passons directement à la phase de génération des cas de test qui sera appliquée sur chaque nouvelle version détectée. Cette phase est divisée en deux étapes:

4.1. La modélisation de chaque version détectée

Le modèle comportemental sur lequel se base la génération des cas de test doit décrire toutes les interactions entre les membres de l'agent holonique, aux différents niveaux, à l'instant *t* dans lequel on veut réaliser un test. Il doit aussi décrire toutes les dépendances comportementales et hiérarchiques entre ces interactions.

- Les interactions qui existent entre les membres de l'agent holonique à l'instant *t* dépendent de l'ensemble des rôles *R* joués par ces membres à cet instant. Elles sont décrites par des diagrammes de séquences, générés dans la phase " *Description de Scenarios d'Interaction du processus ASPECS* ", dans lesquels les rôles de l'ensemble *R* sont invoqués.
- Quant aux relations comportementales et hiérarchiques entre ces interactions, elles dépendent des capacités qui seront exécutées en réponse à ces interactions. En effet, en réponse à une interaction, l'agent récepteur exécute des capacités qui peuvent être implémentées à travers un service publié grâce à un ensemble d'interactions entre ses membres de niveau inférieur. Dans ce cas, la réponse à l'interaction est remplacée par l'ensemble des interactions entre les membres de niveau inférieur de l'agent récepteur.

La génération des modèles comportementaux est réalisée d'une manière incrémentale où:

- La génération du modèle M_0 de la version V_0, est réalisée en fonction des étapes suivantes:
  - Récupérer tous les diagrammes de séquences des groupes contenant des rôles qui sont affectés à des agents à l'instant t_0,
  - Chaque interaction de ces diagrammes de séquences sera spécialisée en communication ou en conversation, ceci à partir de la phase : description des communications du processus ASPEC,
  - Relier les diagrammes de séquences selon leur dépendance hiérarchique et comportementale. Pour cela: chaque interaction obligeant l'agent récepteur à exécuter une capacité implantée par un service publié par ces membres du niveau inférieur, sera remplacée par une référence vers le diagramme de séquences qui représente les interactions entre ces membres depuis lesquels le service est publié (la phase de description des dépendances entre organisation du processus ASPECS sera utilisée),
  - Le résultat sera un diagramme de séquences AUML hiérarchique, sur lequel on se base pour générer les cas de test.
  - A partir de M_1, la génération du modèle M_{i+1} de la version V_{i+1}sera basée sur le modèle $M_i$de la version V_i en prenant en considération la différence entre $V_i$et V_{i+1}décrite par le

delta (V_i, V_{i+1}): M_{i+1}= M_i+ delta (V_i, V_{i+1}). Pour lier le M_i avec delta nous utilisons le même principe utilisé pour la génération de M_0 où chaque interaction, nouvellement introduite, oblige l'agent récepteur à exécuter une capacité implantée par un service publié par ces membres du niveau inférieur, sera remplacée par une référence vers le diagramme de séquences qui représente les interactions entre ces membres depuis lesquels le service est publié (la phase de description des dépendances entre organisation du processus ASPECS sera utilisée).

4.2. La génération des cas de test

Cette étape consiste à générer un ensemble de cas de test, à partir d'un diagramme de séquences AUML hiérarchique, capable de couvrir le comportement de l'agent holonique afin de détecter les potentiels erreurs d'interaction et de scénario où:

- Les erreurs d'interaction: parmi les erreurs qui peuvent survenir dans une interaction, nous pouvons trouver : mauvaise réponse à un message, message correct est passé à un agent incorrect ou message incorrect transmis à l'agent correct, un message invocation avec des arguments inappropriés ou incorrectes, sortie erronée ou manquante, etc.
- Les erreurs du scénario: un diagramme de séquences représente plusieurs scénarios de fonctionnement. Chaque scénario correspond à une séquence différente de messages sur le diagramme de séquences. Pour un scénario d'opération donnée, la séquence de messages peut ne pas suivre la trajectoire souhaitée en raison de l'incorrecte évaluation de la condition, arrêt anormal, etc.

Pour la génération des cas de test, nous avons adapté et étendue l'approche proposée par [Jug13].

Ce travail consiste à :

- Transformer un diagramme de séquences UML vers un graphe SDG (Sequence Diagram Graph) où chaque nœud représente une interaction et contient les informations nécessaires pour la génération des cas de test.
- Générer tous les chemins possibles à partir du graphe.
- Générer une suite de cas de test T sachant que T doit couvrir tous les chemins générés afin de détecter les erreurs d'interaction et de scénario potentiels.

Dans notre cas, on dispose d'un diagramme de séquences AUML hiérarchique qu'on ne peut transformer en un SDG simple car ce dernier ne supporte pas la hiérarchie et les

interactions AUML. Notre proposition consiste à faire une extension de SDG vers HSDG (Hierarchical Sequence Diagram Graph) en apportant les modifications suivantes :

- On crée un nouveau type de nœuds (complexe nœud) qui représente un graphe par rapport au niveau inférieur dans le but de supporter la hiérarchie,
- On divise les nœuds qui représentent les interactions en 2 types (nœud de simple interaction, nœud d'interaction AUML) dans le but de supporter les interactions AUML.
- Les chemins générés à partir de $HSDG_{i+1}$ de la version V_{i+1} vont être filtrés. On obtient deux ensembles de chemins P_α et P_β ; P_α représente les chemins qui existaient déjà dans le $HSDG_i$ de la version précédente V_i, $P_\beta$représente les chemins nouvellement crées.
- La génération des cas de test sera appliquée seulement pour les nouveaux chemins P_β.Les cas de test qui couvrent les chemins $P_\alpha$existent dans la version précédente restent valables pour cette version.
- L'algorithme de génération des cas de test sera appliqué de façon récursive, il se répète à chaque fois qu'il rencontre un nœud complexe dans un chemin.
- La suite de test sera l'union des suites de test qui couvre les chemins P_α et P_β.

La figure 4.3 explique le processus de test de chaque nouvelle version dans sa globalité.

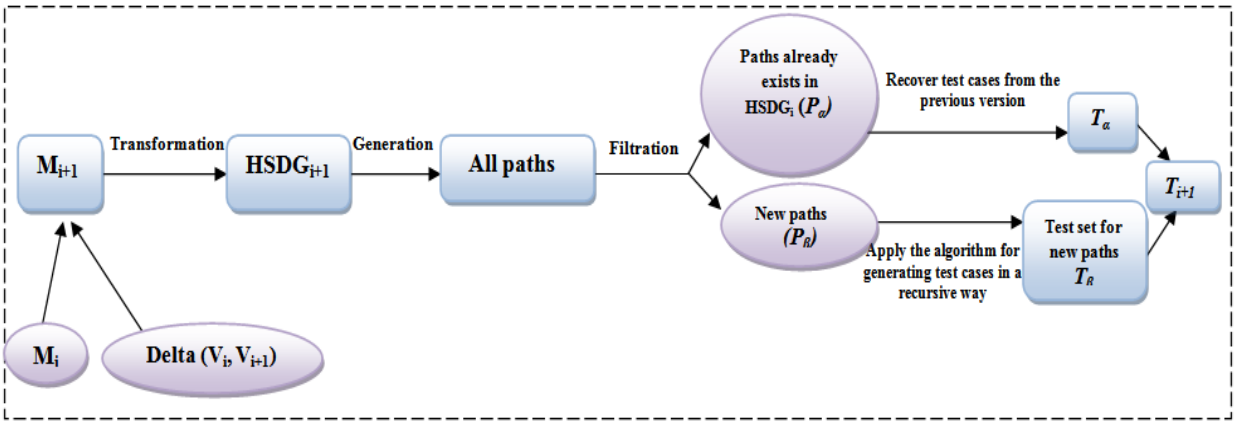


Figure 4.3: Processus de test de chaque nouvelle version détectée.

4.2.1. La définition du HSDG

Le graphe que nous avons proposé est défini comme suit: $HSDG_i = \{N, T, b, F\}$ où,

- **T** est l'ensemble des arêtes représentant les transitions d'un état à un autre.
- **b** est le nœud initial représentant un état à partir duquel une opération commence.
- **F** est l'ensemble des nœuds finaux représentant les états où une opération se termine.
- **N** est l'ensemble de tous les nœuds constitués de (Comp_N ∪Inter_N∪ AUML_N) où Comp_N est un ensemble de noeuds complexes qui représente un graphe par rapport au niveau inférieur, Inter_N est l'ensemble des nœuds d'interaction simple, et AUML_N est l'ensemble de nœuds d'interaction AUML. La structure de chaque nœud $\mathbf{S_i} \in$**N**est définie comme suit: $S_i = <$ Id, Type_N, Details > où:
  - **Id**est un label unique attaché à chaque noeud.
  - **Type_N** = {Simple_int} si $S_i \in$Inter_N,
    Type_N = {Complex} si $S_i \in$ Comp_N, et
    Type_N = {AUML_int} si $S_i \in$ AUML_N.
  - **Details =** {HSDG, [/Guard]} if $S_i \in$ Comp_N, où **HSDG** est un graphe par rapport au niveau inférieur, **guard** est la condition de garde nécessaire pour que l'interaction aura lieu, c'est une partie optionnelle (certaines interactions ne nécessitent pas une condition de garde).
    Details = {m, from_Agent, to_Agent, [/Guard]} if $S_i \in$ Intre_N, où **m** est le nom du message avec sa signature, **from_Agent** est l'émetteur du message et **to_Agent** est le récepteur du message, **Guard** est la condition de garde nécessaire pour que l'événement aura lieu, elle est une partie optionnelle.
    Details = {Set_N, Relationship, [/Guard]} if $S_i \in$ AUML_N, où Set_N représente un ensemble de noeuds où chaque noeud $S_j \in$Set_N a Type_N=Simple_int, Type_N = AUML_int ou Type_N = Complex**,** **Relationship** ∈ {Inclusive, Exclusive, Parallel} représente la relation entre les noeuds $S_j \in$ Set_N. La condition de garde de chaque noeud $S_j \in S_i$ où S_i.Type_N = AUML_int seront redéfinis en fonction de la **Relationship.**

4.2.2. Les informations stockées dans chaque nœud du HSDG

Chaque nœud S_i de type *Simple_int* dans le HSDG doit contenir un ensemble d'informations relatif à l'interaction représentée par ce nœud. Ces informations sont nécessaires lors de la génération des cas de test. Elles sont définies comme suit:

- les attributs (inputs et outputs) associés à la tâche qui sera exécutée suite à l'interaction représentée par le nœud S_i, l'intervalle des valeurs des attributs, et le prédicat de la condition de garde, si elle existe, impliquée dans l'interaction. Ces informations sont collectées à partir de diagramme de classe. Elles peuvent être définies soit dans une classe de rôle si la tâche à exécuter est incluse directement dans le comportement du rôle, soit dans une classe de capacité si la tâche à exécuter est extraite sous forme de capacité.
- En outre, un nœud stocke les contraintes des entrées, les contraintes des entrées-sorties et les résultats attendus. Ces données sont collectées à partir de contraintes, exprimées en OCL, spécifiée dans la tâche correspondante dans le diagramme de classe (les classes de rôle ou de capacité) et à partir du diagramme de cas d'utilisation.
- Enfin, le nœud initial stocke l'ensemble des données (pré-conditions) qui déclenchent tous les scénarios possibles dans le HSDG, et les nœuds finaux stockent les sorties attendues correspondantes (post-conditions) associées aux différent scénarios possibles. Cette information est obtenue à partir du diagramme de cas d'utilisation.

4.2.3. L'algorithme de génération des cas de test

L'algorithme de génération des cas de test **" TestCaseGeneration "** (Figure 4.4) prend comme entrée initiale un $HSDG_i$ de la version V_i. Il a comme sortie un ensemble T de cas de tests capables de couvrir tous les chemins possibles dans le $HSDG_i$. Pour calculer l'ensemble T, l'algorithme suit les étapes suivantes :

- l'algorithme commence par la génération de tous les chemins possibles à partir du niveau supérieur de $HSDG_i$ du premier au dernier nœud en utilisant la fonction *GenerationAllPaths*($HSDG_i$).Les autres chemins des niveaux inférieurs seront générés au fur et à mesure dès qu'un nœud complexe est rencontré. Pendant la génération des cas de test on prend en considération la compatibilité entre les chemins des différents niveaux.
- Ensuite, les chemins générés seront divisés en deux ensembles ; un ensemble de chemins représente des chemins qui existaient déjà dans le $HSDG_{i-1}$ (P_α) et l'autre

ensemble représente les chemins nouvellement crées dans $HSDG_i$ (P_β) en utilisant la fonction*FiltrationPaths (P)*. Cette fonction ne sera pas appliquée pour la version V_0 car dans ce cas tous les chemins générés sont tous considérés comme de nouveaux chemins.

```
Algorithm TestCaseGeneration

Input: Hierarchical Sequence diagram graph HSDG
Output: Test suite T
Begin
    GenerationAllPaths (HSDGi) // the output will be P = {p1,p2……pn} where P is a set of all paths from the start node to
                                  the final node in the HSDGi of Vi.
    Guard_condition_path (P)   // the output will be G= {G1, G2……Gn} where Gi is a set of Guard condition associated
                                  the path Pi
    if i=0 then
        Cover _path (P)        // the output will be Test suite Tthat cover all paths P
    else
        FiltrationPaths (P)    // the output will be Pβ a set of the paths newly created in Vi
        Cover_path (Pβ)        // the output will be Test suite Tβ that cover Pβ
    End if
End.
```

Figure 4.4: L'algorithme *TestCaseGeneration.*

- Pour chaque chemin P_i nouvellement créé, l'algorithme calcule l'ensemble T_i(cas de test de P_i) en utilisant la fonction *Cover_path (P)* (Figure 4.5). Pour calculer T_i, l'algorithme doit parcourir tout le chemin P_i, nœud par nœud où il récupère la pré-condition du scénario correspondant à P_i à partir du nœud initial, il récupère la post-condition du scénario correspondant à P_i à partir du nœud final, alors qu'à partir des nœuds S_j qui se trouvent entre le nœud initial et le nœud final, l'algorithme calcul t_j (cas de test d'une interaction représentée par S_j).
- La couverture des nœuds se diffère d'un nœud à un autre selon leurs types (Figure 4.5):
    - Si le nœud S_j est un nœud de type simple dans ce cas le calcul de t_j revient à récupérer les informations stockées dans ce nœud, sachant que les entrées récupérées doivent être compatibles avec la garde condition, si elle existe. Pour cela, on utilise la fonction *Cover_node (S_j)* (Figure 4.6).
    - Si le nœud S_j appartient à un nœud S_m de type *AUML_int*, dans ce cas, avant de récupérer les informations il faut recalculer sa garde condition, sachant que les entrées qui couvrent ce nœud sont capables de couvrir un autre nœud appartenant au nœud S_m ce qui risque de dévier l'exécution vers un autre chemin pendant le test de P_i. La redéfinition de la garde condition minimise l'intervalle des entrées de telle sorte qu'on soit certain que les entrées récupérées ne couvrent que le nœud désiré (sorte de bouchon). Après la redéfinition de la garde condition, S_j sera considéré comme un simple nœud. Il

sera couvert selon son type. De cette manière, l'algorithme est capable de générer des cas de test capables de couvrir chaque scénario individuellement, ce qui implique que chaque erreur détectée dans un scénario donné n’est pas associée à un autre scénario qui s’exécute en parallèle avec.

```
Function Cover_path (Pβ)
Input: Pβ                          // the set of the paths newly created in Vi (Pβ = P if we are in the version V0)
Output: Tβ                         // Test suite Tβ that cover Pβ
Begin
    For each path Pi∈ Pβ  do
        Sj= Sx                     // Sj is the current node; start with Sx the start node
        preCi                      // the precondition of the scenario corresponding to Pi  stored in Sx
        ti ← Φ                     // The test case for the scenario corresponding is initially empty
        Sj = Sj+1                  // Move to the first node of the path Pi
        While (Sj ≠ Sz)   do       // Sz being a final node
            If  Sj.Type_N = Simple_int then
             Cover_node (Sj)       // the output will be Test suite tj that cover Sj
             ti= tiUtj
            End if
            If Sj∈ Sm / Sm.Type_N = AUML_int then
                Initial_guard = Sj.Guard
                S_AUML = Sm
             (1):
              if  S_AUML.Relationship = Inclusive then
                  Sj.Guard = (Sj.Guard ∪ S_AUML.Guard) ∪ (! S1.Guard ∪! S2.Guard ∪…….∪! Sm.Guard )
                  // where: if it is the first execution of (1) then {S1, S2, ….. Sm} is the Set of nodes in Sm with the exception of Sj, else
                  {S1, S2, ….. Sm} is the Set of nodes in Sm with the exception of Sm-1
                 End if
                 if  S_AUML.Relationship = Exclusive then
                   Sj.Guard = (Sj.Guard ∪ S_AUML.Guard) ∩ (S1.Guard ∪ S2.Guard ∪…….∪Sm.Guard)
                  // where if it is the first execution of (1) then {S1, S2, ….. Sm} is the Set of nodes in  Sm with the exception of Sj, else
                  {S1, S2, ….. Sm} is the Set of nodes in  Sm with the exception of Sm-1
                 End if
                 if  S_AUML.Relationship = Parallel then
                   Sj.Guard = S_AUML.Guard∪ (! S1.next.Guard ∪! S2.next.Guard ∪…….∪! Sm.next.Guard )
                   // where if it is the first execution of (1) then {S1, S2, ….. Sm} is the Set of nodes in  Sm with the exception of Sj, else
                   {S1, S2, ….. Sm} is the Set of nodes in Sm with the exception of Sm-1. // S.next is the next node of S with have a guard
                 End if
              End (1)
              If  Sm∈ Sm+1/ Sm+1.Type_N = AUML_int then // Sm also belongs in AUML node
                 S_AUML = Sm+1
                 m=m+1
                 GOTO (1)
                End If
                If Sj.Type_N = Simple_int then
                   Cover_node (Sj)          // the output will be Test suite tj that cover Sj
                    ti= tiUtj
                End if
                If Sj.Type_N = Complexthen
                   GOTO (2)
                End if
             Sj.Guard = Initial_guard
            End if
          (2):
            If Sj.Type_N= Complex  then
                GenerationAllPaths (Sj)      // the output will be P_Sj = {P1_Sj, P2_Sj……Pn_Sj} where P_Sj is a set of all
                                               Path from  the start node to a final node in the Sj
              Guard_condition_path (P_Sj)    // the output will be G_Sj= {G1_Sj, G2_Sj……Gn_Sj } where Gk_Sj is a set of
                                               Guard Condition associated the path Pk in the graph Sj
              P_Sj_Compatible ← Φ
             For each Pk_Sj ∈ P_Sj do
                 If (∀ C ∈ Gk_Sj ) /  !C ∉ Gi then
                  P_Sj_Compatible= P_Sj_Compatible ∪ Pk_Sj
                 End if
             End for
                Cover_path (P_Sj_Compatible)  // the output will be Test suite tj_compatible that cover the compatible paths
                                                 in the graph Sj
                ti= tiUtj_compatible
            End if
          End (2)
            Sj = Sj+1                        // Move to the next node on the path Pi
            Tβ ← TβUti
        End while
        t = { preCi,  Ii , Oi, postC }       // Determine the final output Oi and postCi  for the  Pi  stored in Sz
      Tβ ← TβU t
    End for
  Return (Tβ)
End.
```

Figure 4.5: La fonction *Cover_path (Pβ)*.

- Si le nœud est un nœud complexe, c'est-à-dire un graphe par rapport au niveau inférieur, la récupération des informations nécessaires pour couvrir ce nœud revient à chercher les informations nécessaires pour couvrir les chemins inclus dans ce nœud. Pour cela, l'algorithme sera réappliqué d'une façon récursive, sauf que parmi les chemins générés l'algorithme sélectionne, seulement, ceux qui sont compatibles avec P_i. On dit qu'un chemin existe dans S_j est compatible avec P_i si l'exécution de P_i implique l'exécution de celui-là. Ceci ne se réalise que s'il n'y a aucune garde condition dans les nœuds de ce chemin qui soit antagoniste avec les gardes conditions qui existent dans les nœuds du chemin P_i. Pour cela, on utilise la fonction *Guard_condition_path* (Figure 4.7). De cette manière, l'algorithme soit capable de prendre en considération la compatibilité entre les scénarios des différents niveaux, ce qui permet à ce dernier d'outrepasser la difficulté de la nature hiérarchique du modèle comportemental de l'agent holonique.

```
Function Cover_node (Sj)
Input: a node SjwhereSj. Type_N = Simple_int
Output: Test suite tj:                    //a test case that cover Sj
Begin
    if Sj.Guard = Φ then                  //  there is no guard condition
        Select test case
        Tj= {preC, I (a1, a2, …..al), O (b1,b2,….bm), postC }
                                          //where preC = precondition of the task that will be executed, I (a1,  a2,…..,al) =set of input
                                            values for the task from from_Agent,  O(b1,b2,…..,bm) is a set of resulting  values in  the ,
        to_Agent  when the task is  executed,  postC = the postcondition of the task.
    end if
    if  Sj.Guard ≠ Φ then                 //the task  is under guard condition
        C (v) =(c1, c2, c3,…., cl )       //The set of values of clauses on the path Pi
        Select test case
        tj= {preC, I (a1, a2, …..al), O (b1,b2,….,bm),C(v),postC}
    End if
End.
```

Figure 4.6: La fonction*Cover_node (s_j).*

```
Function Guard_condition_path
Input: P = {p1,p2……pn}          //  where P is a set of all paths from the start node to  the final node in the HSDGiof Vi.
Output: G= {G1, G2……Gn}          // where Gi is a set of Guard condition associated to the path Pi
Begin
  G ← Φ
  For each path Pi∈ P do
    Gi ← Φ
    For each nodeSj∈Pido
      Gi = GiUSj.Guard
    End for
  G = G UGi
  End for
End.
```

Figure 4.7: la fonction *Guard_condition_path*.

L'application de l'algorithme de génération des cas de test sur un HSDG nous permet d'avoir un ensemble de cas de test capable de couvrir toute la version (s'il s'agit de la version V_0) et de couvrir les chemins nouvellement créés (ou ceux transformés) à partir de la version V_1. Chaque cas de test généré contient un ensemble d'entrées capables de couvrir un chemin désiré (P_i) et un ensemble de sorties prévues lors de l'exécution de l'agent holonique avec ces entrées. Il contient aussi la pré-condition nécessaire pour l'exécution du chemin (P_i) et la post-condition prévue lors de l'exécution de l'agent holonique avec les entrées présentées dans le cas de test. Pour détecter les erreurs potentielles, il suffit d'exécuter l'agent holonique avec les entrées décrites dans les cas de test et de comparer les résultats obtenus, suite à cette exécution, avec les sorties prévues (résultats attendus) décrites dans les cas de test. La non-conformité entre les résultats obtenus et les résultats attendus signifie qu'il ya une erreur d'interaction et de la non-conformité entre la post-condition obtenue et celle attendue signifie qu'il ya une erreur de scénario.

Cet algorithme a été implémenté à l'aide de l'environnement Eclipse Java 4.4. Nous avons utilisé le plug-in ECLIPSE UML2 pour éditer les diagrammes UML, dont le format XML (i.e. généré à partir des diagrammes UML) a été utilisé lors de la génération des $HSDG_i$. Pour les nœuds des $HSDG_i$, nous avons utilisé la structure des arbres pour faciliter la génération de cas de test.

5. Conclusion

Dans ce chapitre, nous avons proposé une nouvelle approche de test pour les agents holoniques pouvant s'adapter avec leurs comportements et leurs structures qui se développent avec le temps et avec leur nature hiérarchique. Cette approche consiste à tester l'agent holonique par version successives; ceci nous permet de couvrir tous les comportements acquis avec le temps et de corriger, à temps, toute erreur éventuelle, qui pourrait biaiser le comportement de l'agent holonique. Cette approche se décompose en deux sous processus. Le premier processus porte sur la détection des nouvelles versions à tester. Dans ce processus, la fonction de fitness proposée, nous permet de suivre le développement de l'agent holonique et de déterminer avec exactitude le niveau et le membre ayant causé le développement. Ce qui facilite la détermination du delta entre deux versions successives et nous fait gagner du temps en nous évitant la comparaison entre les versions. Le deuxième processus porte sur le test de chaque nouvelle version détectée. La nouvelle version de l'agent est analysée afin de générer un modèle comportemental sur lequel est basée la génération des

cas de test. Le processus de génération des cas de test se concentre sur les nouvelles (et / ou modifiés) parties du comportement de l'agent. De cette manière, la technique prend en charge une mise à jour incrémentielle des cas de test, qui est un problème crucial.

Chapitre 5

Étude de cas: Système de Distribution de Marchandises

1. Introduction

Pour valider notre approche, nous l'appliquons sur une étude de cas concrète: "***Distribution of Goods System***" (un Système de Distribution de Marchandises). Ce système représente, à notre avis, un bon exemple à tester en utilisant notre approche pour deux raisons: (i) il contient un grand nombre d'entités en interaction, et (ii) le comportement du système se développe suite à l'introduction de nouvelles interactions entre ses entités. L'objectif de ce système est de distribuer la marchandise à un ensemble de points définis par leurs coordonnées (x_i, y_i). Le système reçoit les points de distribution sous forme d'un vecteur de nombres réels (x_1, y_1,…x_i,y_i,… x_n, y_n) (x_i, y_i représentes les coordonnées d'un point de distribution). Selon les points de distribution, la nature de la requête est déterminée, où la requête peut être locale, auquel cas les *n* points sont dans la zone « A » ($0 < x_i < width_A$, $0 < y_i < lengthe_A$). La requête peut être à distance, auquel cas les *n* points ne sont dans la zone « A » ($x_i > width_A$, $y_i > lengthe_A$). Ils peuvent être aussi un mélange entre une requête locale et à distance, dans ce cas, nous avons *m* points dans la zone « A » ($0<x_i < width_A$, $0<y_i < lengthe_A$) et *n-m* points qui ne sont pas dans la zone « A » ($x_i > width_A$, $y_i > lengthe_A$). Une fois que le système assure qu'il dispose de la quantité demandée (quantité demandée < max quantité), il analyse le

vecteur de points de distribution pour déterminer la nature de la requête (locale, à distance ou un mélange entre eux) pour envoyer la requête au service approprié. Le choix de l'étude de cas a été fondée sur le fait qu'elle a permis l'illustration de différentes étapes de l'approche proposée d'une manière simple et claire. Dans ce qui suit, nous allons d'abord décrire les différentes phases du développement de notre étude de cas en utilisant le processus ASPECS. Ensuite, nous allons appliquer notre approche de test sur elle.

2. Phases du développement de l'étude de cas avec le processus ASPECS

2.1. Phase d'analyse des besoins

La phase d'analyse doit fournir une description complète du problème sur la base des abstractions définies dans le domaine de problème du méta-modèle CRIO (Capacité, Rôle, Interaction et Organisation). Dans ce qui suit, nous présentons l'objectif, la description et les diagrammes de chaque étape utilisée dans la phase d'analyse des besoins.

a. Description des besoins du domaine

L'objectif de cette phase consiste à dresser une première description du contexte de l'application et de ses fonctionnalités. Cette activité vise à identifier, classifier et hiérarchiser l'ensemble des besoins fonctionnels et non-fonctionnels des différentes parties prenantes du projet. Elle doit également fournir une première estimation du périmètre de l'application ainsi que de sa taille et de sa complexité. Une approche d'analyse des besoins, basée sur les diagrammes de cas d'utilisation UML, est adoptée. La figure 5.1 présente les différents cas d'utilisation associés à notre étude de cas. Quatorze cas d'utilisation et un acteur ont été identifiés. L'acteur représente l'utilisateur qui peut envoyer des requêtes au système. Parmi les cas d'utilisation que nous avons:

- *Distribution Orientation*: responsable de recevoir et de diriger les requêtes.
- *Request verification*: responsable de vérifier la requête et d'étudier la possibilité de sa réalisation.
- *Request analysis*: responsable d'analyser la requête et de la classer comme une requête locale ou à distance.
- *Local distribution request*: responsable du traitement des requêtes locales.
- *Distance distribution request*: responsable du traitement des requêtes à distance.

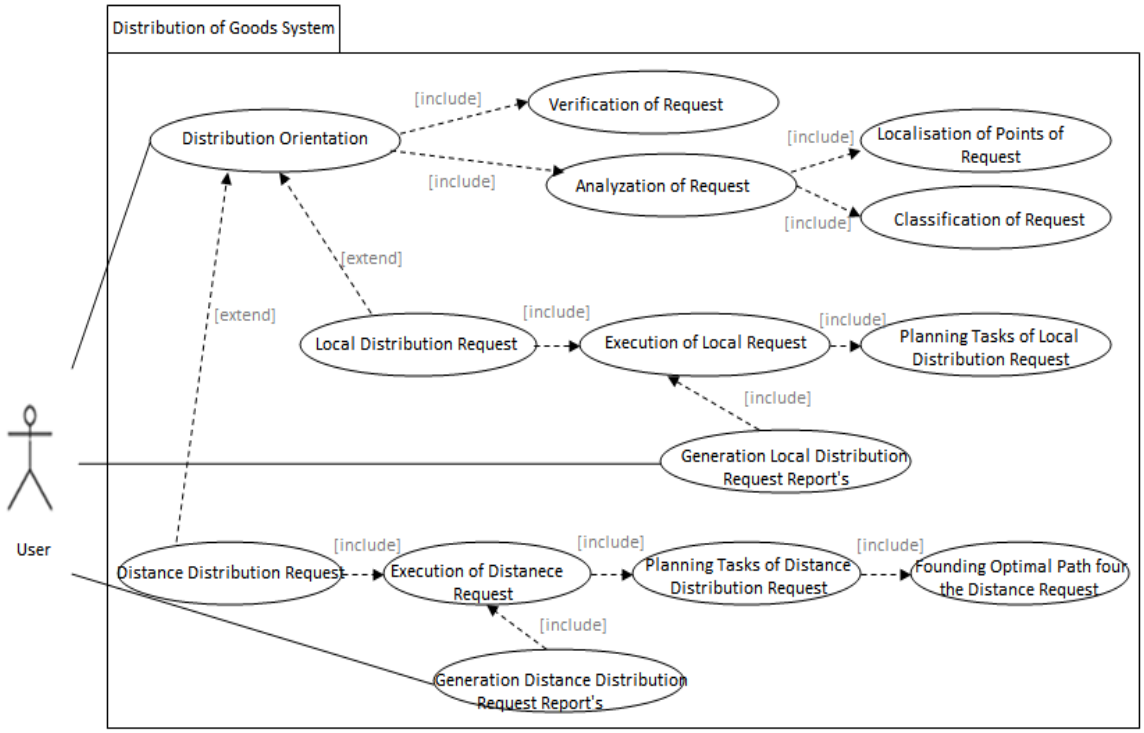


Figure 5.1: Diagramme de cas d'utilisation du "*Distribution of Goods System*".

b. Description de l'ontologie du problème

La description de l'ontologie du problème doit fournir une première définition du contexte de l'application et du vocabulaire spécifique au domaine. Elle vise à approfondir la compréhension du problème, en complétant l'analyse des besoins et les cas d'utilisation, avec la description des concepts qui composent le domaine du problème, et de leurs relations. L'ontologie de "*Distribution of Goods System*" est décrite dans (Figure 5.2).

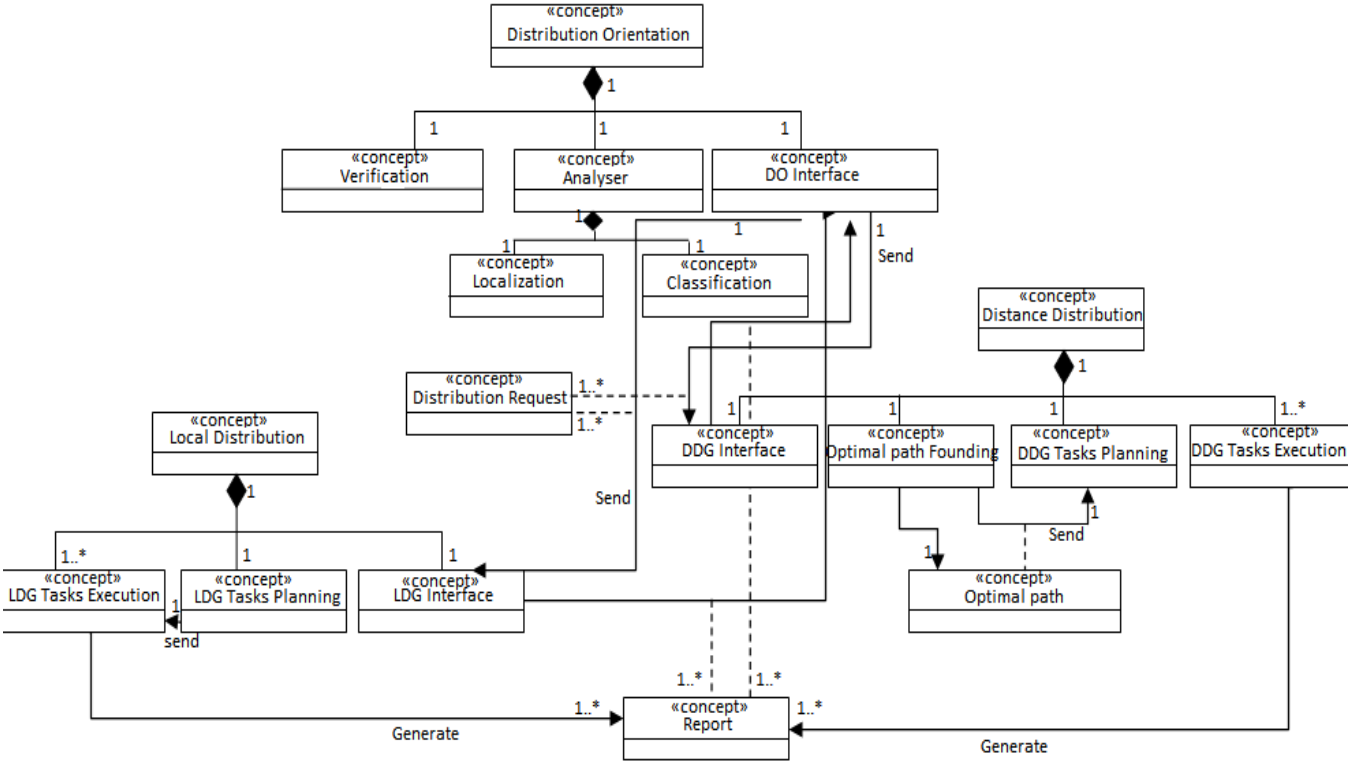


Figure 5.2: L'ontologie du "*Distribution of Goods System*".

c. Identification des organisations

L'identification des organisations doit établir une première décomposition organisationnelle du système et définir les objectifs de chaque organisation. Chacun des besoins, identifiés dans la première activité, se voit associer une organisation incarnant le comportement global en charge de le satisfaire ou de le réaliser. Différentes approches peuvent être adoptées pour partitionner les cas d'utilisation et ainsi faciliter l'identification des organisations du système. Dans notre cas, nous avons adopté une partition fonctionnelle des cas d'utilisation. Cette approche considère que les membres d'une même organisation partagent les mêmes objectifs. Les cas d'utilisation, dont les fonctionnalités sont proches ou liées, sont regroupés dans une même organisation. Les organisations, ainsi identifiées, sont directement ajoutées au diagramme de cas d'utilisation, sous la forme de paquets stéréotypés englobant les cas d'utilisation qu'elles sont en charge de satisfaire. Le diagramme de cas d'utilisation présenté dans la figure 5.3 présente une partie des organisations identifiées dans "*Distribution of Goods System*".

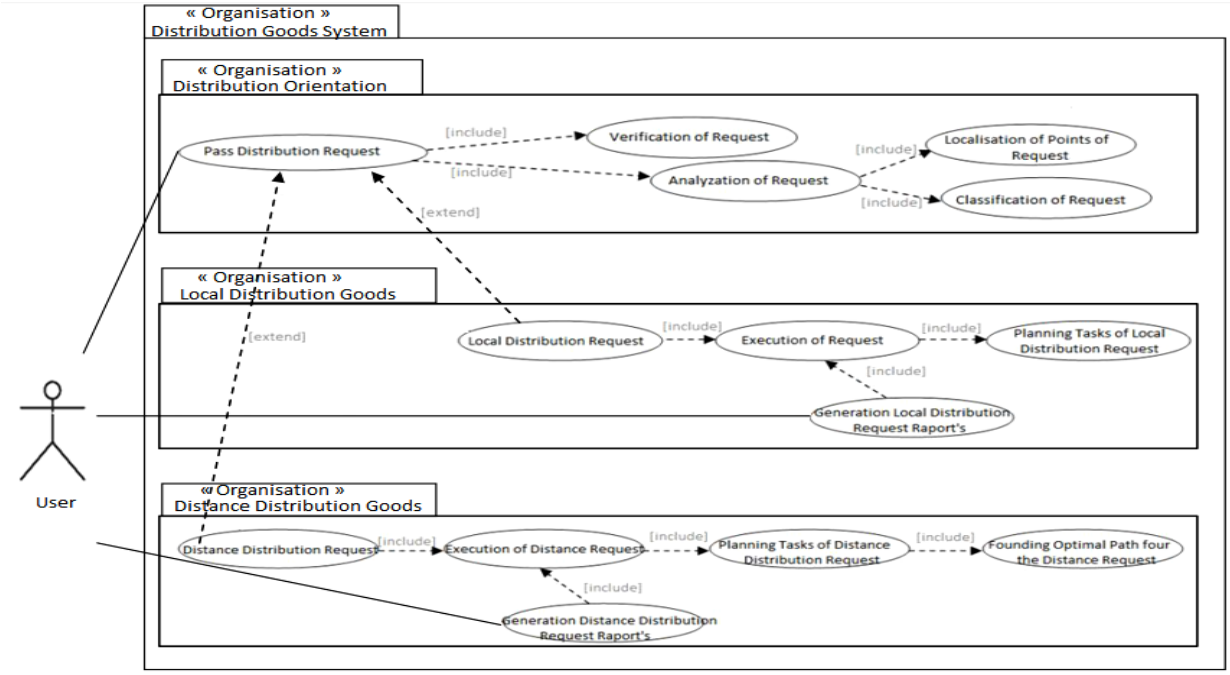


Figure 5.3: Une partie des organisations identifiées dans "*Distribution of Goods System*".

Comme il est montré dans la figure 5.3, nous avons trois organisations principales: Local *Distribution of Goods (LDG)*, *Distance Distribution of Goods (DDG)* et *Distribution Orientation (DO).*

- L'objectif de *"Local Distribution of Goods" (LDG)* est: (a) recevoir les requêtes de distribution à partir de "*Distribution Orientation*", (b) la planification des actions de distribution, (c) l'exécution des tâches de distribution, et (d) la génération du rapport de distribution.
- L'objectif de "*Distribution Orientation*" (DO) est: (a) recevoir les requêtes de distribution de différents point, (b) la vérification des requêtes, (c) localisation des requêtes (d) classification des requêtes, (e) l'envoi de la requête de distribution à (LDG) et / ou (DDG), et (f) recevoir les rapports d'exécution à partir de LDG et / ou DDG.
- L'objectif de "*Distance Distribution of Goods*" (DDG) est: (a) recevoir les requêtes de distribution à partir de "*Distribution Orientation*", (b) trouver le chemin optimal, (c) la planification des actions de distribution, (d) l'exécution des tâches de distribution, et (e) la génération du rapport de distribution.

L'identification des organisations est basée sur un processus itératif. Cet ensemble d'organisations peut être complété par itérations progressivement afin de déterminer la

hiérarchie organisationnelle représentant le système. Cette décomposition hiérarchique du système continue à un niveau où la complexité du comportement (des rôles) est suffisamment faible pour être exécutée par des entités considérées comme atomique. Par exemple, l'organisation *local distribution of goods* peut être décomposée en trois sous-organisations: *LDG Interface*, *Tasks Planning* et *Tasks Execution* comme le montre la figure 5.4.

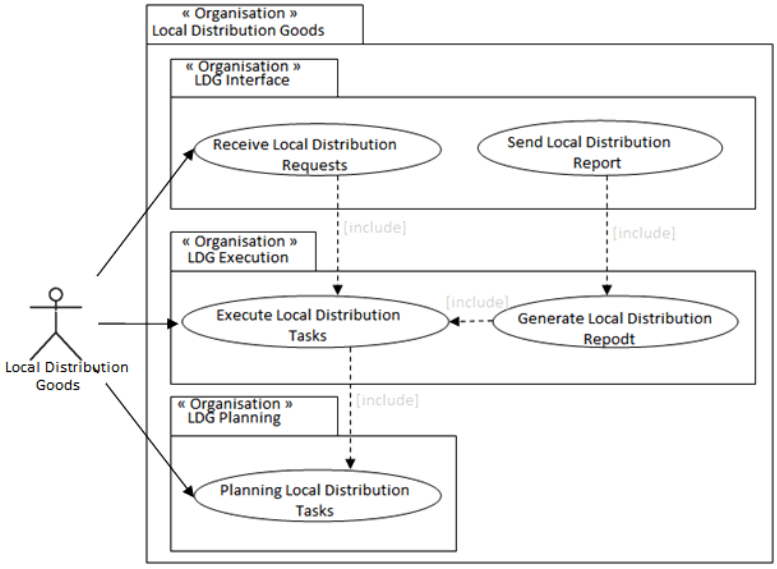


Figure 5.4: L'organisation *local distribution of goods*.

Le même processus peut être utilisé pour décomposer les organisations : *Distribution Orientation* et *Distance Distribution of Goods*. Selon cette décomposition, la hiérarchie organisationnelle finale de "*Distribution of Goods System*" est décrite dans la figure 5.5.

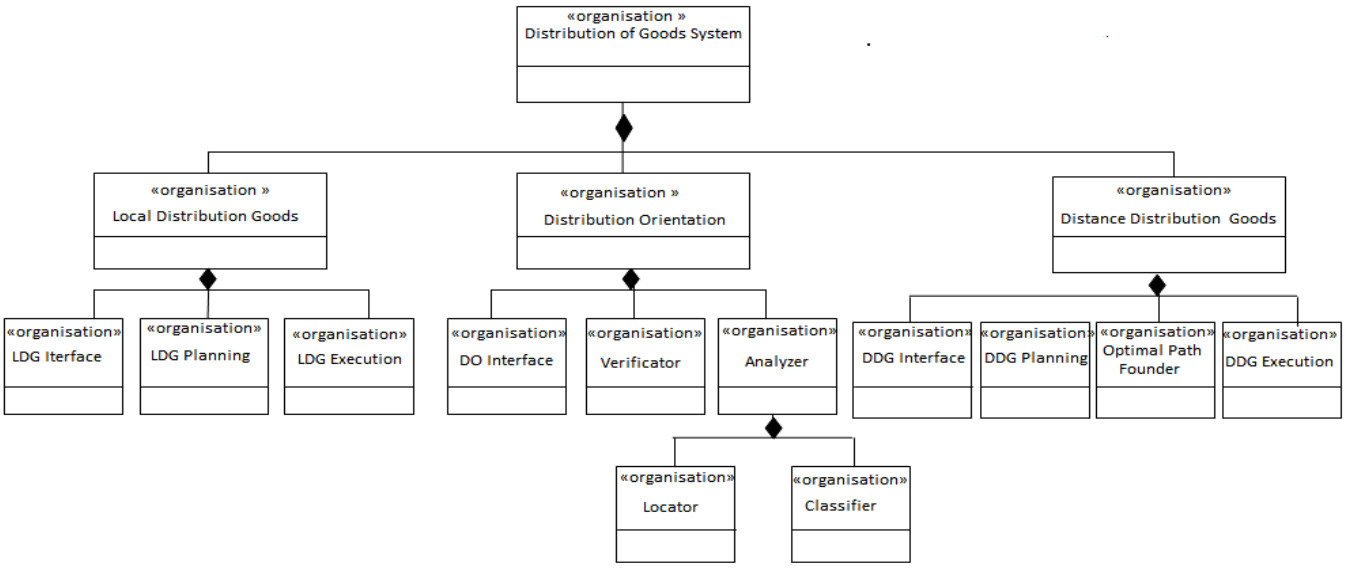


Figure 5.5: Hiérarchie organisationnelle du "*Distribution of Goods System*".

d. Identification des interactions et des rôles

L'identification des interactions et des rôles vise à décomposer le comportement global incarné par une organisation en un ensemble de rôles en interaction. Cette activité doit également décrire les responsabilités de chaque rôle pour la satisfaction des besoins associés à leurs organisations respectives. Les rôles et les interactions composant chaque organisation sont ajoutés à leurs diagrammes de classe comme le décrit la figure 5.6.

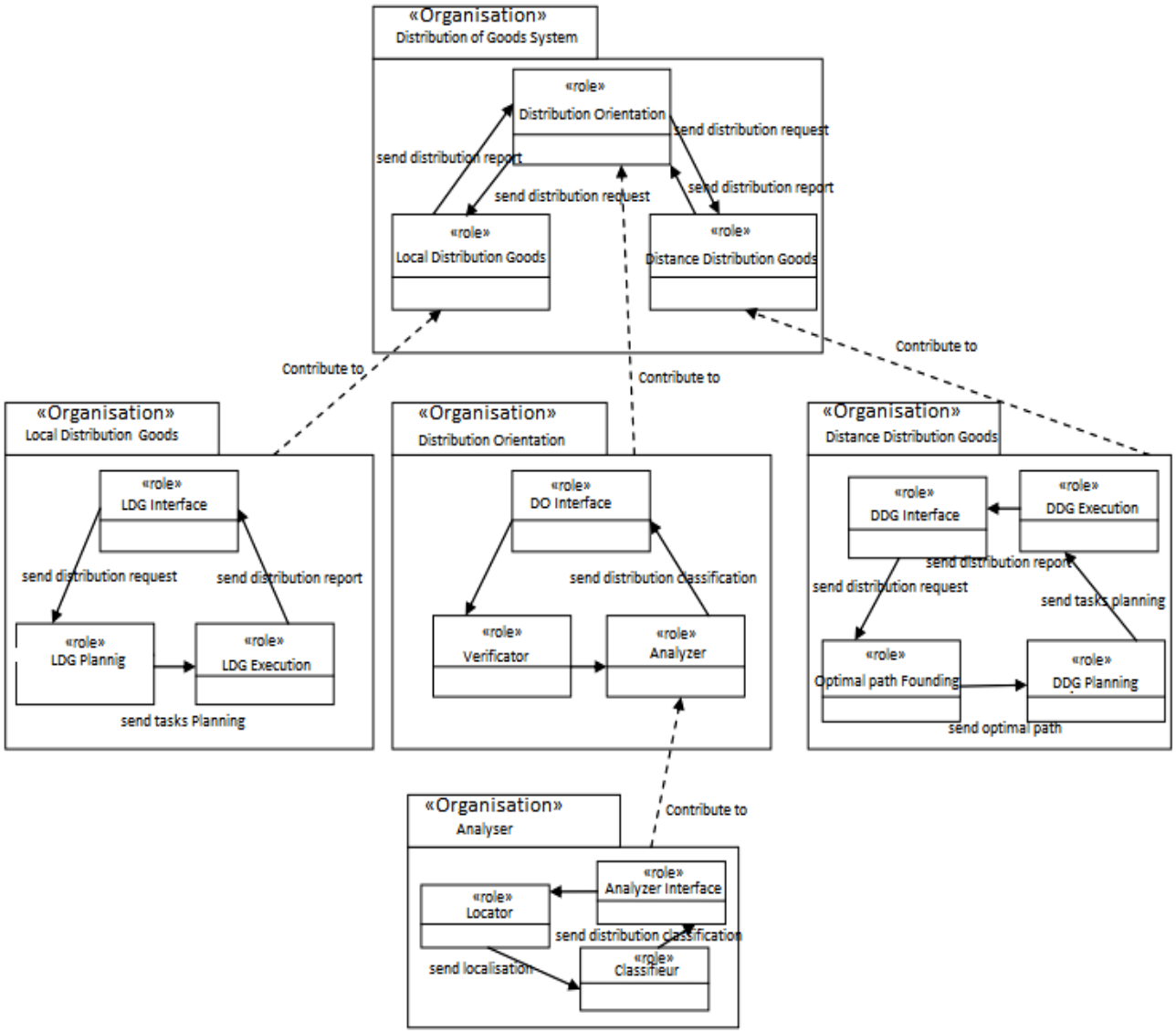


Figure 5.6: Description des rôles et des interactions de l'organisation "*Distribution of Goods System*".

Prenons, par exemple, l'organisation *Local Distribution Goods*. Dans cette organisation on a trois rôles:

- *LDG Interface*: charger de recevoir et d'organiser les requêtes locales.
- *LDG Planning*: charger de faire un plan pour la requête en tenant compte de l'état du système et la disponibilité des ressources.
- *LDG Execution*: charger de l'exécution de la requête, tout en respectant le plan et produire un rapport sur l'exécution de la requête.

e. Description des scénarios d'interaction

L'objectif de cette activité consiste à préciser les interactions entre les rôles pour donner naissance à un comportement de plus haut niveau. Elle décrit les interactions entre les rôles définis au sein d'une organisation donnée et précise les moyens de coordination entre eux pour satisfaire les objectifs assignés à leur organisation. La description des scénarios d'interaction est supportée par un ensemble de diagrammes de séquences UML. A titre d'exemple, la description des scénarios d'interaction de l'organisation LDG est représentée par la figure 5.7.

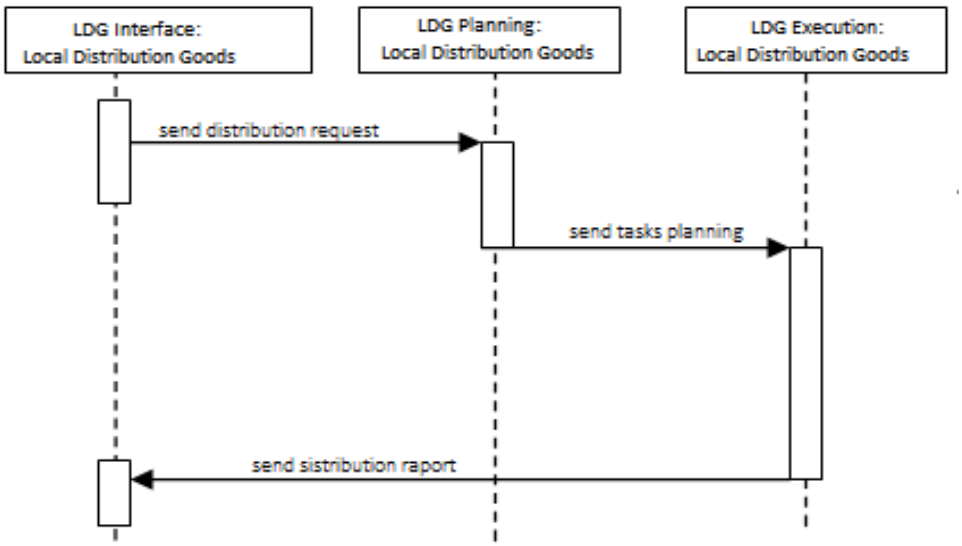


Figure 5.7: Description des scénarios d'interaction de l'organisation LDG.

Chacun des trois rôles de cette organisation dispose de sa propre ligne de vie: Ce scénario commence par la réception de requêtes locales par le rôle *LDG interface*. Ce dernier organise les requêtes reçues et les envoie, une par une, au rôle *LDG Planning*. Ce dernier, à son tour, génère un plan et l'envoie au rôle *LDG Execution* qui l'exécute et génère un rapport d'exécution.

f. Description des plans de comportement des rôles

La description des plans de comportement des rôles vise à spécifier le comportement de chaque rôle en accord avec les objectifs qui lui ont été assignés et les interactions dans lesquelles il est impliqué. Chaque plan de comportement décrit la combinaison et l'ordonnancement des interactions, des événements extérieurs et des tâches qui composent le

comportement de chaque rôle. La figure 5.8 montre les plans de comportements des rôles qui constituent l'organisation "*Distribution of Goods System*".

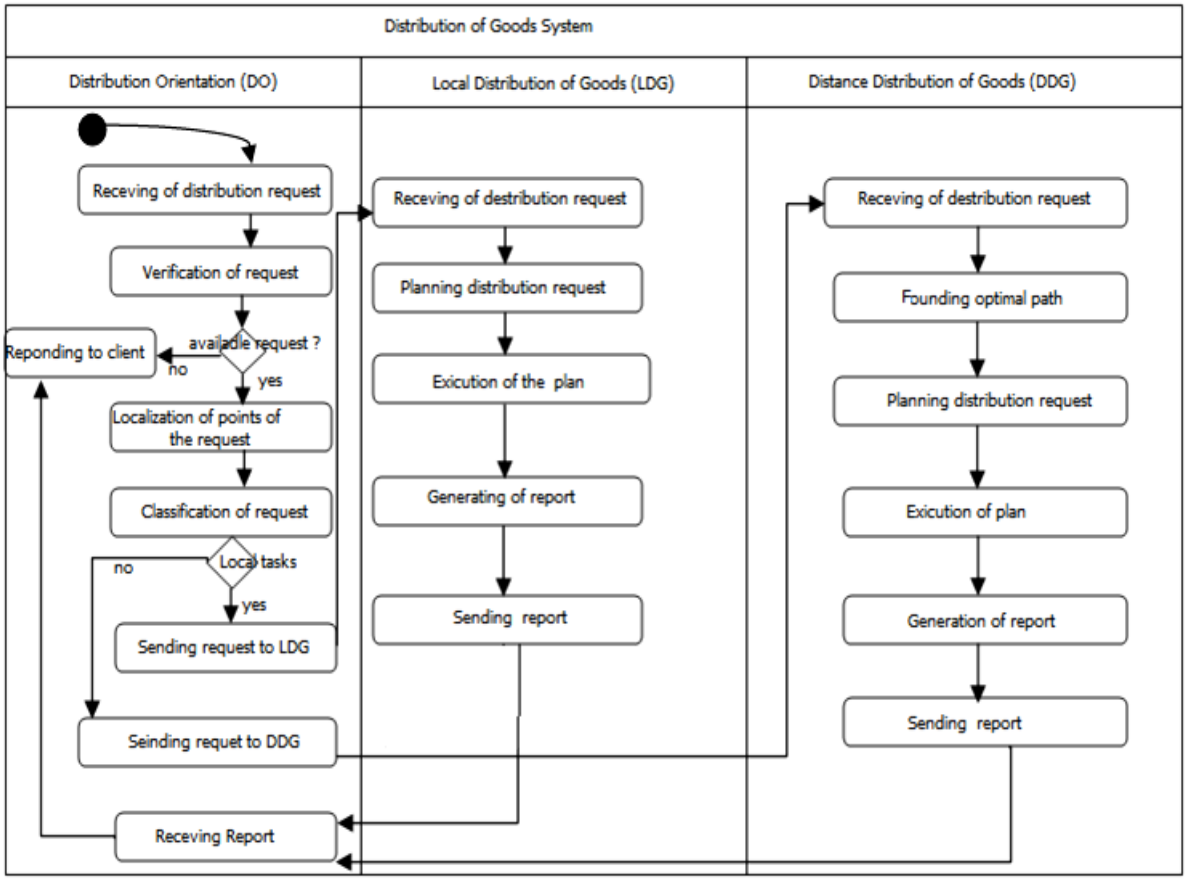


Figure 5.8: Description des plans de comportements des rôles de l'organisation "*Distribution of Goods System*".

Chacune des interactions précédemment identifiées dans le scénario d'interaction doit apparaître dans le plan comportemental, soit sous la forme d'un flux de contrôle ou de données, soit sous la forme d'un événement. Cette correspondance permet de vérifier que le plan de comportement en cours de modélisation satisfait effectivement les différents scénarios dans lesquels le rôle correspondant est impliqué.

g. Identification des capacités

Cette activité vise à augmenter la généricité des comportements des rôles, en séparant clairement la définition de ces comportements des dépendances externes de leurs organisations. Elle consiste notamment à raffiner le comportement des rôles, pour faire abstraction de l'architecture des entités qui vont les jouer, et assurer leur indépendance à l'égard de tout élément extérieur à la définition du rôle lui-même. L'identification des

capacités doit déterminer l'ensemble des compétences nécessaires à chaque rôle. Les capacités sont ajoutées sous forme de classes stéréotypées dans les diagrammes de classe UML des organisations concernées; et elles sont reliées par une association aux rôles qui les requièrent. Dans notre cas, nous allons identifier les capacités requises par les rôles de l'organisation "*Distribution of Goods System*" (Figure 5.9).

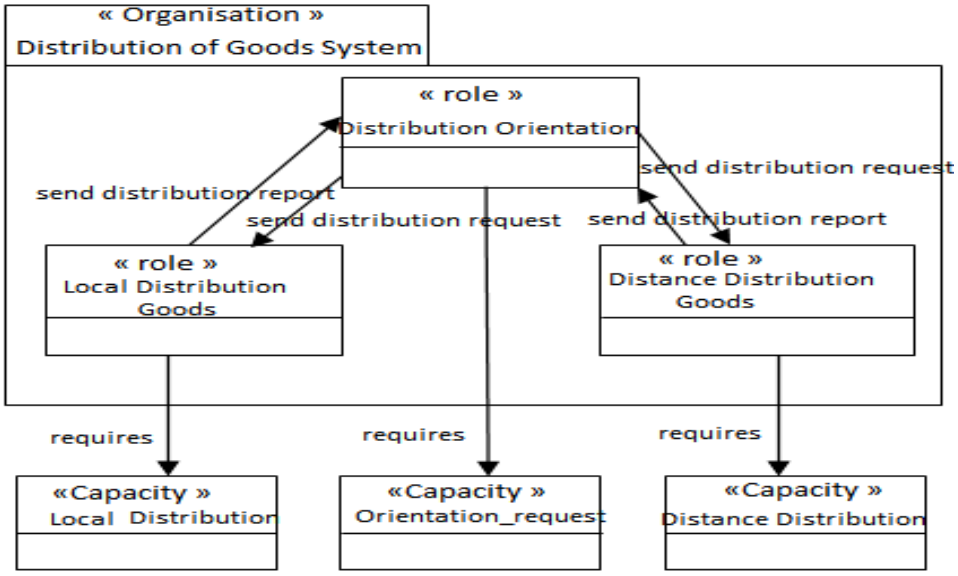


Figure 5.9: Identification des capacités requises par les rôles de l'organisation "*Distribution of Goods System*".

2.2. Phase de conception de la société agent

Cette phase vise à concevoir une société d'agents, dont le comportement global est en mesure de fournir une solution efficace au problème décrit dans la phase précédente et de satisfaire aux exigences associées. Dans cette phase, nous passons directement à l'activité de description de la conception de la holarchie. Cette activité est consacrée à l'agentification de la hiérarchie organisationnelle et à la définition des entités en charge de l'exécuter. Son objectif consiste à définir les holons du système et à en déduire la structure de la holarchie. Pour construire la holarchie de l'application, les organisations qui composent le système sont instanciées sous forme de groupes. Un ensemble de holons est ensuite créé à chaque niveau, chacun d'eux joue un ou plusieurs rôles dans un ou plusieurs groupes du niveau considéré. Les relations de composition entre super-holons et sous-holons sont ensuite spécifiées en accord avec les contributions entre organisations définies dans la hiérarchie organisationnelle. La hiérarchie organisationnelle est donc directement associée à une hiérarchie de holons (ou holarchie). Les règles qui régissent la dynamique de ces holons, ainsi que les types de

gouvernement de chaque type de holon composé, sont également décrits. Tous ces éléments sont ensuite synthétisés pour décrire la structure de la holarchie initiale du système. En tant que première version (V_0) de notre système, nous n'avons pas instancié l'organisation *Distance Distribution of Goods* (DDG). La structure holonique du "*Distribution of Goods System*" dans sa version initiale (V_0) est présentée dans la figure 5.10.

Au niveau 2 de la holarchie, nous trouvons deux super-holons H1 et H2, qui jouent respectivement les rôles *Distribution Orientation (DO)* et *Local Distribution of Goods* (LDG) dans le groupe G1: "*Distribution of Goods System*" (G1 est une instance de l'organisation "*Distribution of Goods System Organization*". Au niveau 1, les holons H3, H4, H5, sont des membres du holon H1. Ils jouent respectivement les rôles: *Interface, Verificator, Analyzer*, dans le groupe G2: "*Distribution Orientation"* (G2 est une instance de l'organisation "*Distribution Orientation Organization(DO)"*). Les holons H6, H7, H8, sont des membres du holon H2. Ils jouent respectivement les rôles: *LDG Interface, LDG Planning, LDG Execution*, dans le groupe G3: "*Locale Distribution of Goods"* (G3 est une instance de l'organisation "*Locale Distribution of Goods Organization* (LDG)"). Au niveau 0 les holons H9, H10, H11, sont des membres du holon H5. Ils jouent respectivement les rôles: *Analyzer Interface, Locator, Classifier*, dans le Groupe G4: "*Analyzer"* (G4 est une instance de l'organisation "*Analyzer organization"*).

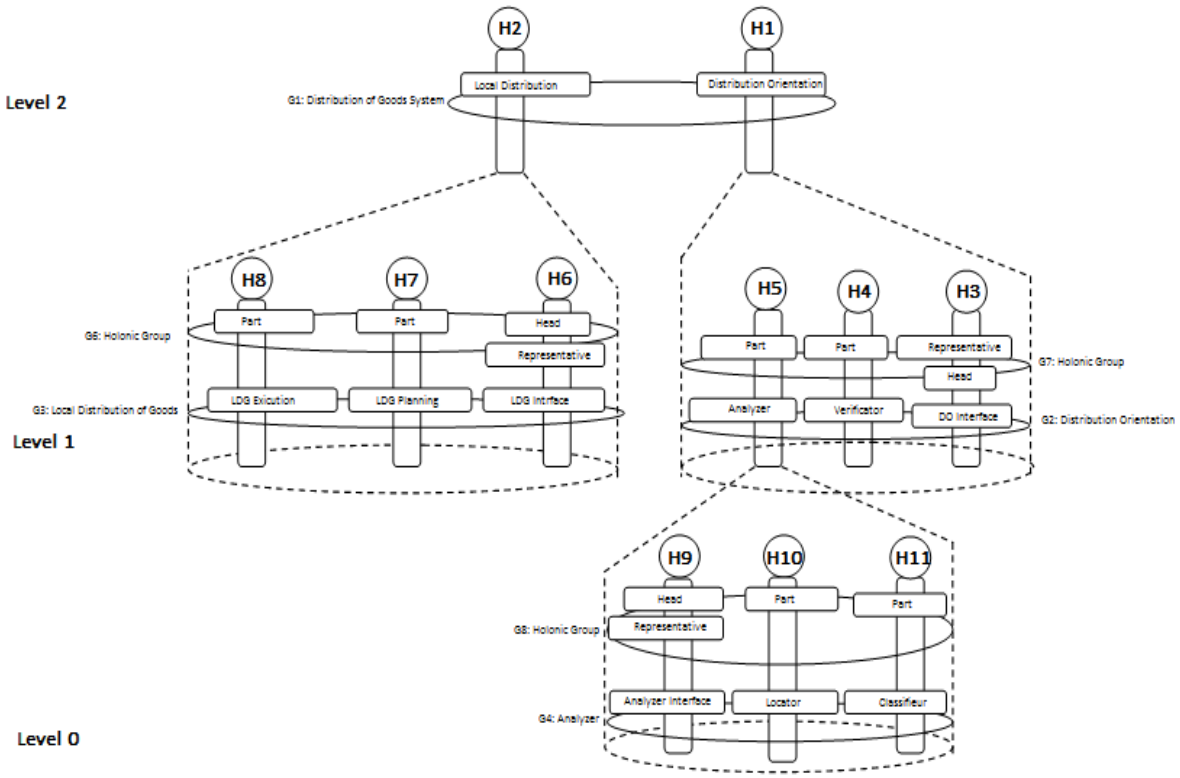


Figure 5.10: Structure holonique du "*Distribution of Goods System*" dans sa version initiale (V_0).

2.3. Phase d'implantation

Cette phase consiste à implémenter la solution orientée agent conçue dans les phases précédentes en l'adaptant à la plate-forme de mise en œuvre choisie. Pour cette phase, nous avons utilisé la plate-forme Janus. Elle est spécifiquement conçue pour traiter des aspects holoniques et organisationnels. Le but de Janus est de fournir un ensemble complet d'outils pour faciliter le travail du développeur. Les deux principales contributions de Janus sont : (i) sa gestion native de holons, (ii) elle considère le rôle comme une entité du premier ordre, indépendante de l'agent. Cependant, la plate-forme Madkit [Gut00] et son extension MOCA [Mat03] ne considère pas le rôle comme une entité du premier ordre. En effet, le comportement associé au rôle est directement implanté dans l'agent qui le joue. Les rôles sont donc fortement liés à l'architecture des agents. Cette approche nuit à la réutilisation et à la modularité des organisations. MOCA considère effectivement les rôles comme des entités du premier ordre, mais fixe des contraintes strictes concernant leur mise en œuvre. Par exemple,

un agent ne peut jouer plusieurs fois le même rôle. Un fragment du code source de la classe correspondante ou holon H1 est fourni dans la Figure 5.11, ou les méthodes activate(), live() et end() correspondent aux trois phases de son cycle de vie.

```
1   public class Holon_1 extends LightHolon {
2
3    //_______Attributes of the holon______
4    private GroupAddress G1_distribution_of_goods_system;
5
6    //The first step of the life cycle of holon: Activation
7    public void activate() {
8           // Capacity allocation to holon
9           addCapacity(Orientation_request_Capacity.class, new
10          Orientation_request_CapacityImpl(this));
11
12          //Creating a group or access to the address an existing group
13          //that implements the organization Distribution_Goods_System_Organization
14          G1_distribution_of_goods_system =
15          getOrCreateGroup(Distribution_Goods_System_Organization.getInstance());
16
17          //Ask for access to role R1_distribution_orientation in the previous group
18
19          if(requestRole(R1_distribution_orientation.class,
20          G1_distribution_of_goods_system)){
21          //Logging of information
22          getLogger().info("R1_distribution_orientation assigned");
23             }
24    }
25
26    // 2nd Stage of the lifecycle holon: behavior
27    public void live() {
28
29        //Execution of roles
30         while(true) {
31         for (Role role : getRoles())
32         role.behavior();
33              }
34     }
35     // 3rd Stage of the lifecycle holon: Termination
36     public void end() {
37          //ask of release of Role
38          if(leaveRole(R1_distribution_orientation.class,
39          G1_distribution_of_goods_system)){
40           getLogger().info("role Distribution_Goods_System_Organization:
41                            R1_distribution_orientation disassigned");
42           } else {
43                    getLogger().error ("Pb during disassignation of
44                                       Distribution_Goods_System_Organization:
45                                       R1_distribution_orientation assigned role");
46                  }
47     }
48
49  }
```

Figure 5.11 : Un fragment du code source de la classe correspondante au holon H1.

3. Application de l'approche proposée

Dans le cadre de notre approche, le test de l'étude de cas considéré sera effectué par versions, où la version de départ est la version actuelle V_0. Dans ce qui suit, nous décrivons d'abord les étapes de test de la version V_0, puis à partir de cette version nous allons essayer de détecter une version plus développée, d'un point de vue comportementale, V_1. Nous allons décrire par la suite les étapes de test de la nouvelle version V_1 en référence à leur relation avec les étapes de test de la version V_0.

3.1. Test de la version V_0

Le test de la version V_0 avec notre approche se concentre sur deux étapes. La première étape consiste à générer un modèle comportemental M_0 associé à cette version. La deuxième étape consiste à générer un ensemble de cas de test à partir du modèle M_0 capable de couvrir tous les chemins possibles de la version V_0.

a. M_0: Le modèle de la version V_0

Cette phase consiste à générer un modèle de comportement M_0 associé à cette version (V_0), représenté par un diagramme de séquence hiérarchique. Pour obtenir M_0 il faut relier tous les diagrammes de séquences des groupes contenant les rôles qui sont attribués à des agents à l'instant t0, selon leurs dépendances hiérarchiques et comportementales. Pour cela, nous comptons sur le principe suivant: chaque interaction forçant l'agent récepteur à exécuter une capacité implémentée par un service publié par les membres du niveau inférieur, sera remplacée par une référence vers le diagramme de séquences qui représente les interactions entre ses membres à partir desquelles le service est publié (la phase de la description des dépendances entre les organisations de processus ASPECS sera utilisée). En conséquence, on obtient le diagramme de séquences AUML hiérarchique (M_0) suivant (Figure 5.12).

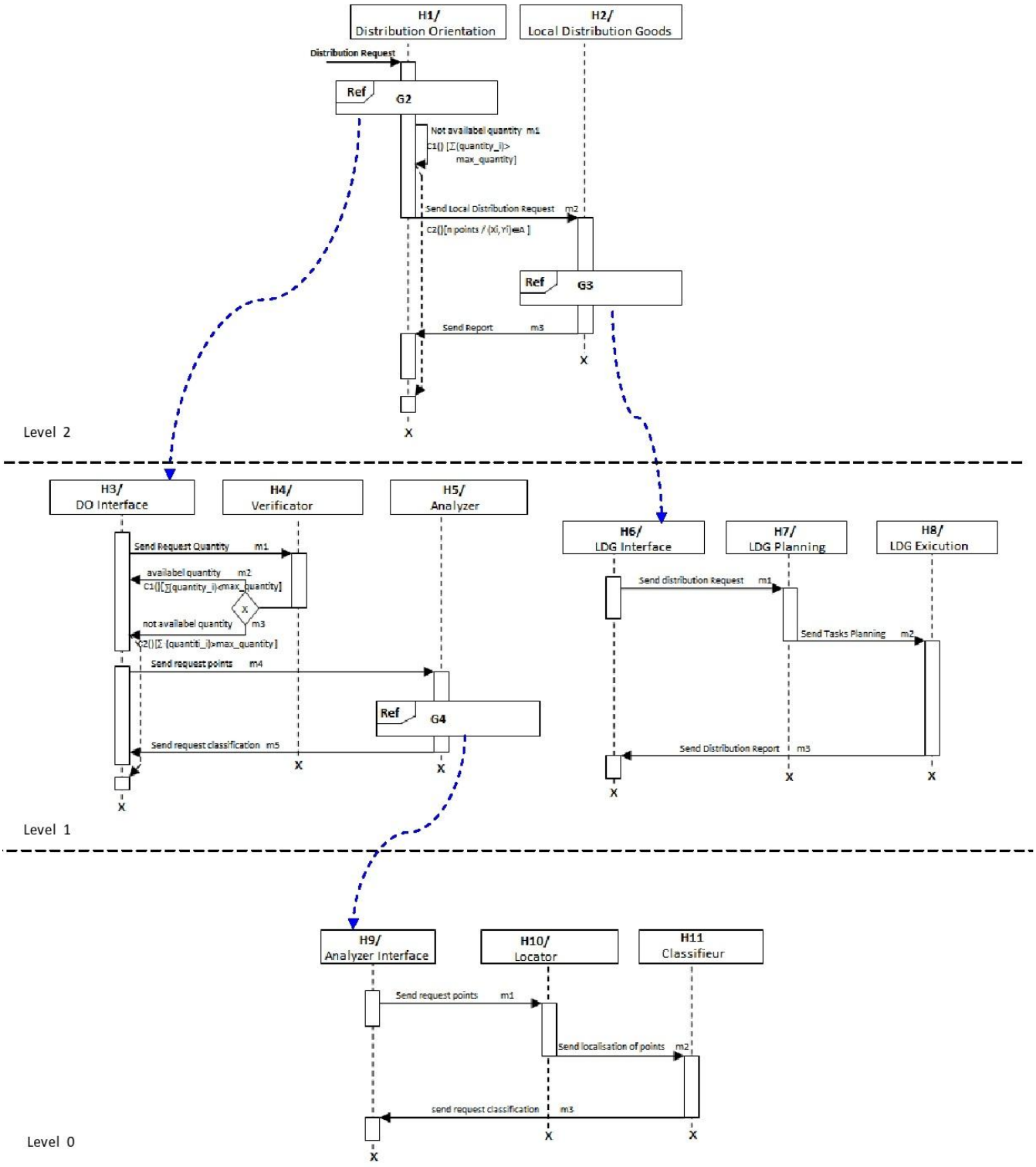


Figure 5.12: Diagramme de séquences AUML hiérarchique (M_0) de (V_0).

b. Génération des cas de test pour la version V_0

Cette phase est organisée en deux étapes. La première étape consiste à transformer le diagramme de séquences hiérarchiques associé à la version V_0en un Graphe de diagramme de séquences hiérarchiques ($HSDG_0$), ceci en présence d'un ensemble d'informations récupérées à partir d'autres diagrammes associés au système. La deuxième étape consiste à générer les cas de test à partir de $HSDG_0$ créé dans la première étape, ceci à travers l'exécution du programme de génération des cas de test avec $HSDG_0$ comme entrée initiale.

b.1.$HSDG_0$: Hierarchical Sequence Diagram Graphde la version V_0

Les figures 5.13 et 5.14 représentent le $HSDG_0$ (Hierarchical SequenceDiagram Graph) de la version V_0 et les différents scénarios associés à ce graphe respectivement. Chaque nœud représenté dans le $HSDG_0$ représente une interaction. Il contient toutes les informations nécessaires pour la génération des cas de test. Les nœuds d'où émerge une flèche bleue (en pointillé) vers les niveaux inférieurs indiquent des nœuds complexes qui représentent des graphes correspondant à ces niveaux. Tous les nœuds situés dans le cercle en pointillé sont des nœuds inclus dans un type de nœud AUML.

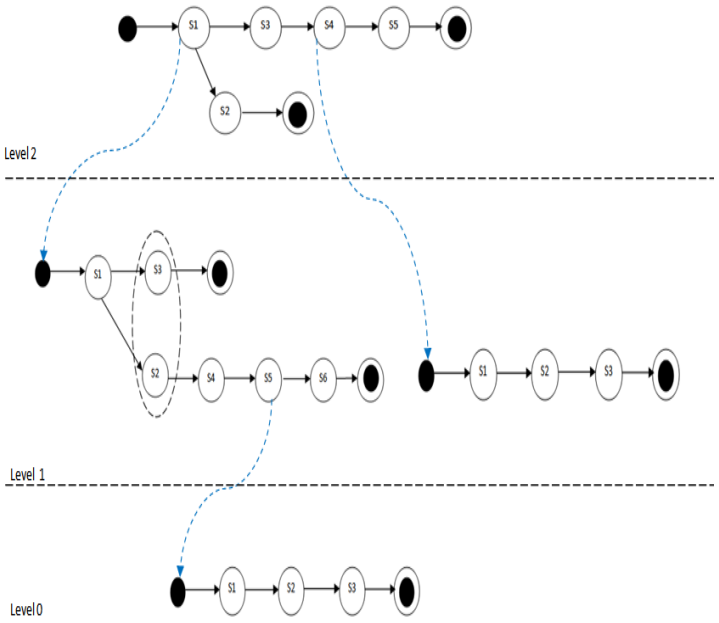


Figure 5.13: Hierarchical Sequence Diagram Graph ($HSDG_0$) de V_0

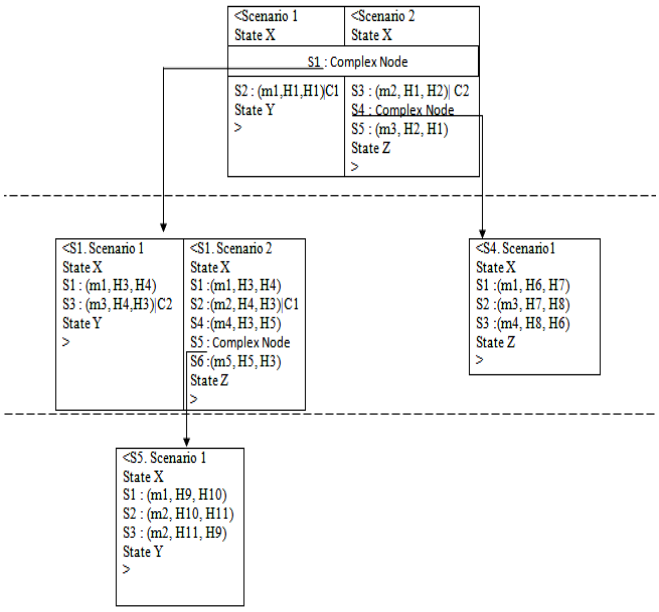


Figure 5.14: Les différents scénarios associés au $HSDG_0$

b.2. T_0: les cas de test de la version V_0

Sachant que V_0 est la 1ère version du système, dans ce cas, l'exécution du programme de génération des cas de test avec le $HSDG_0$ revient à générer les cas de test qui couvrent tous les chemins générés à partir de $HSDG_0$. Pour cette exécution nous avons choisi les valeurs suivantes : $width_A$= 25 ; $lengthe_A$= 25; la langueur des vecteurs n = 5; Max quantité =500000 (les même valeurs sont prises pour tout le reste de cette étude de cas). Sachant que pour chaque chemin, nous pouvons avoir plusieurs cas de test, nous avons pris 05 cas de test pour chaque scénario. Les résultats de l'exécution du programme montrent les cas de test capables de couvrir les deux scénarios détectés dans la version V_0 (Figure 5.15).

```
_____T0 Test Case of V0_____

Preconditions: The system is ready to receive request
               The user send a Request
Test case of Scenario_1
T_Scenario_1 = {Input: the sum of (quantity_i) >max_quantity , Output: Displays “Not available
quantity” ,Postcondition: Displays “repeat your request”}
5 possible test case for scenario_1
Case1
       Input: Quantity1= 116819, Quantity2= 41957, Quantity3= 235467,
Quantity4= 615741, Quantity5= 1038
       Output: Displays “Not available quantity”
Postcondition: Displays “repeat your request”
     Case2
Input: Quantity1= 45391, Quantity2= 367103, Quantity3= 68236,
               Quantity4= 694581, Quantity5= 17294
Output: Displays “Not available quantity”
Postcondition: Displays “repeat your request”
Case3
       Input: Quantity1= 116819, Quantity2= 41957, Quantity3= 235467,
Quantity4= 615741, Quantity5= 1038
       Output: Displays “Not available quantity”
Postcondition: Displays “repeat your request”
    Case4
Input: Quantity1= 750116, Quantity2= 94637, Quantity3= 299456,
               Quantity4= 34752, Quantity5= 10945
Output: Displays “Not available quantity”
Postcondition: Displays “repeat your request”
Case5
       Input: Quantity1= 18462, Quantity2= 6294526, Quantity3= 935267,
Quantity4= 463789, Quantity5= 128456
       Output: Displays “Not available quantity”
Postcondition: Displays “repeat your request”
Test case of Scenario_2
T_Scenario_2 = {Input: (0 < the sum of (quantity_i) <max_quantity , n points / (0 < xi <width_A , 0 <
yi<lengthe_A)) , Output: (Displays “available local request” , Displays the report) , Postcondition:
Displays “Return to the initial state”}
5 possible test case for scenario_2
Case1
       Input: Quantity1= 3827, Quantity2= 4583, Quantity3= 1845,
Quantity4= 9463, Quantity5= 238
              Vector = ((19.40 , 22.55) , (15.79 , 12.34) , (20.65 , 13.07) ,
(5.61 , 19.88) , (22.42 , 1.91))
       Output: Displays “available local request”
                Displays the report
Postcondition: Displays “Return to the initial state”
Case2
       Input: Quantity1= 4922, Quantity2= 9381, Quantity3= 3821,
Quantity4= 8574, Quantity5= 958
              Vector = ((24.71 , 23.18) , (24.96 , 0.81) , (6.41 , 7.98) ,
(5.44 , 21.84) , (23.59 , 6.12))
       Output: Displays “available local request”
                Displays the report
Postcondition: Displays “Return to the initial state”
Case3
       Input: Quantity1= 3648, Quantity2= 946, Quantity3= 9283,
Quantity4= 939, Quantity5= 9574
              Vector = ((8.63 , 19.97) , (18.96 , 5.22) , (4.64 , 6.97) ,
(1.71 , 1.08) , (14.08 , 11.01))
       Output: Displays “available local request”
                Displays the report
Postcondition: Displays “Return to the initial state”
Case4
       Input: Quantity1= 1925, Quantity2= 87, Quantity3= 4784,
Quantity4= 481, Quantity5= 3842
              Vector = ((10.18 , 11.80) , (24.96 , 12.93) , (14.20 , 23.75) ,
(8.68 , 23.19) , (0.86 , 2.03))
       Output: Displays “available local request”
                Displays the report
Postcondition: Displays “Return to the initial state”
Case5
       Input: Quantity1= 102, Quantity2= 3917, Quantity3= 9483,
Quantity4= 611, Quantity5= 2935
              Vector = ((20.50 , 22.98) , (18.76 , 19.76) , (18.47 , 2.26) ,
(23.37 , 11.50) , (2.78 , 0.93))
       Output: Displays “available local request”
                Displays the report
Postcondition: Displays “Return to the initial state”
Postcondition: Displays “Enter your request”
```

Figure 5.15: Cas de test de la version $V_{0.}$

3.2. La détection de la nouvelle version V1

Le comportement de notre système dans sa version V_0 se limite à résoudre des requêtes locales. Dans le but de trouver une version plus développée capable de résoudre des requêtes à distance, nous exécutons la version V_0 avec des vecteurs de nombre réel ($x_{1,}$ y_1,…$x_{i,}$ $y_{i,\ldots}$ $x_{n,}$ y_n) ($x_{i,}$ y_i) représentant les coordonnées d'un point de distribution/ x_i<width, y_i<lengthe) générés automatiquement par un algorithme génétique jusqu'à atteindre un développement dans le comportement du système signalé par la fonction H, ce qui exprime la détections de V_1. Pour cela, nous avons adopté un codage réel et nous avons choisi les valeurs suivantes : Taille de la population initiale= 20 (généré aléatoirement / x_i<$width_A$, y_i<$lengthe_A$); nombre maximum de générations = 20; probabilité de mutation = 2%, probabilité de croisement = 80%. La figure 5.16 exprime les changements apportés sur H pendant l'exécution de la version V_0.

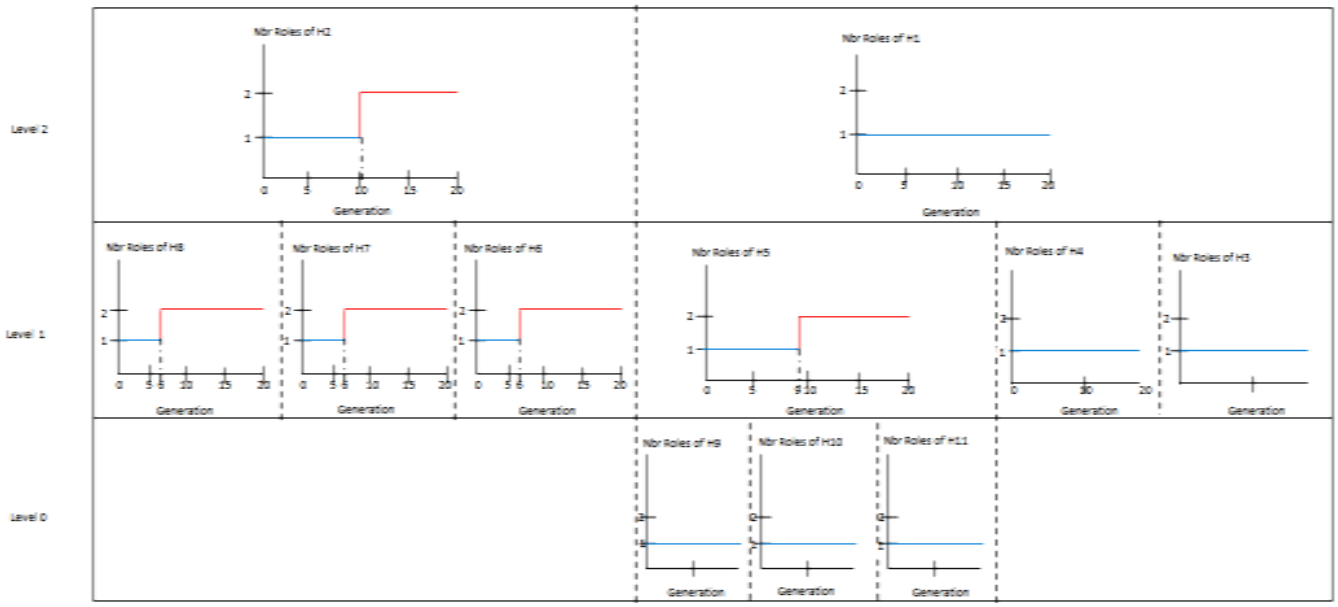


Figure 5.16: Modifications apportées à la fonction H durant l'exécution de la version V_0.

L'augmentation de la fonction de fitness, comme le montre la figure 5.16, indique que l'agent holonique a commencé d'obtenir de nouveaux rôles, ce qui signifie qu'il est entré dans la phase de développement. L'entrée de l'agent holonique dans cette phase est à partir de la génération N°6. À ce point, il a commencé à rencontrer de nouveaux vecteurs générés par l'algorithme génétique, qui, il est incapable de résoudre (Figure 5.17), car il ne possède pas les rôles requis pour cette tâche. Pour résoudre ces nouveaux vecteurs, l'agent holonique est forcé à obtenir de nouveaux rôles, comme observé dans les éléments H6, H7, H8, H5, H2 et indiqué par l'incrémentation de la fonction de fitness tel que représenté sur la figure 5.16. à partir de

la génération N°10, l'incrémentation de la fonction de fitness est à un état de maximisation caractérisé par le fait que l'agent holonique est devenu capable de résoudre les vecteurs générés par l'algorithme génétique comme le montre la figure 5.17, ce qui indique qu'il a obtenu tous les rôles nécessaires pour résoudre ces vecteurs. Cela indique la fin de cette phase et la détection d'une version plus développée V_1.

```
---The execution of the holonic agent with the generations---
Generation 1
all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 1, 1, 1), (1, 1))
Generation 2
all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 1, 1, 1), (1, 1))
Generation 3
all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 1, 1, 1), (1, 1))
Generation 4
all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 1, 1, 1), (1, 1))
Generation 5
all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 1, 1, 1), (1, 1))
Generation 6
not all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 2, 2, 2), (1,
1))
Generation 7
not all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 2, 2, 2), (1,
1))
Generation 8
not all vectors of this generation are solved  H = ((1, 1), (1, 1, 1, 2, 2, 2), (1,
1))
Generation 9
not all vectors of this generation are solved  H = ((1, 1), (1, 1, 2, 2, 2, 2), (1,
1))
Generation 10
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 11
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 12
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 13
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 14
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 15
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 16
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 17
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 18
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 19
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
Generation 20
all vectors of this generation are solved  H = ((1, 2), (1, 1, 2, 2, 2, 2), (1, 1))
```

Figure 5.17: Résultats obtenus après l'exécution de l'agent holonique avec les générations.

Pour connaitre les nouveaux rôles obtenus dans la nouvelle version V_1, il suffit de récupérer les tables T_{ij} associées à chaque agent. La figure 5.18montre T_{ij} associé à chaque agent.

```
--- Tij associated to each agent ---
Level 2 ------------------------
T2.1= {Distribution_Orientation}
T2.2= {Local_Distribution, Distance_Distribution}
Level 1 ------------------------
T1.3= {DO_Interface}
T1.4= {Verificator}
T1.5= {Analyzer, Optimal_path_founding}
T1.6= {LDG_Interface, DDG_Interface}
T1.7= {LDG_Planning, DDG_Planning}
T1.8= {LDG_Exicution, DDG_Exicution}
Level 0 ------------------------
T0.9= {Analyzer_Iterface}
T0.10= {Locator}
T0.11= {Classifieur}
```

Figure 5.18: la table T_{ij} associé à chaque agent.

Selon les tables T_{ij}, (Figure 5.18), nous trouvons que l'agent H5 du niveau 1 a obtenu le rôle " *Optimal path founding* ", l'agent H6 du niveau 1 a obtenu le rôle "*DDG Interface* ", l'agent H7 du niveau 1 a obtenu le rôle "*DDG Planning* ", l'agent H8 du niveau 1 a obtenu le rôle " *DDG Execution* " et l'agent H2 du niveau 2 a obtenu le rôle "*Distance Distribution* ".Ceci peut être exprimé par le fait que l'agent H2 a tenté de jouer le rôle " *Distance Distribution* " pour pouvoir répondre aux requêtes à distance qu'il commence à recevoir. Ne disposant pas de la capacité nécessaire pour jouer ce rôle, l'agent H2 a dû réaliser une acquisition dynamique de capacité. Il a instancié l'organisation *Distance Distribution of Goods* (DDG) sous forme de groupe (G5) dont il a distribué les rôles sur ses membres H6, H7, H8 selon leur capacité. Il a aussi recruté l'agent H5 pour lui donner le rôle "*Optimal path founding* " pour palier à l'absence de la capacité nécessaire pour ce rôle chez ses membres. Les interactions introduites entre les agents H5, H6, H7, H8 (définies par le group G5) ont engendré la publication d'un service qui permettra à H2 d'acquérir une nouvelle capacité grâce à laquelle il pourra jouer le rôle "*Distance Distribution* " dans le groupe G1. Le H2 de l'agent a été obligé d'effectuer une acquisition dynamique de capacité car elle était la meilleure solution pour lui. La structure holonique du "*Distribution of Goods System*" dans sa version (V_1) est présentée dans la figure 5.19.

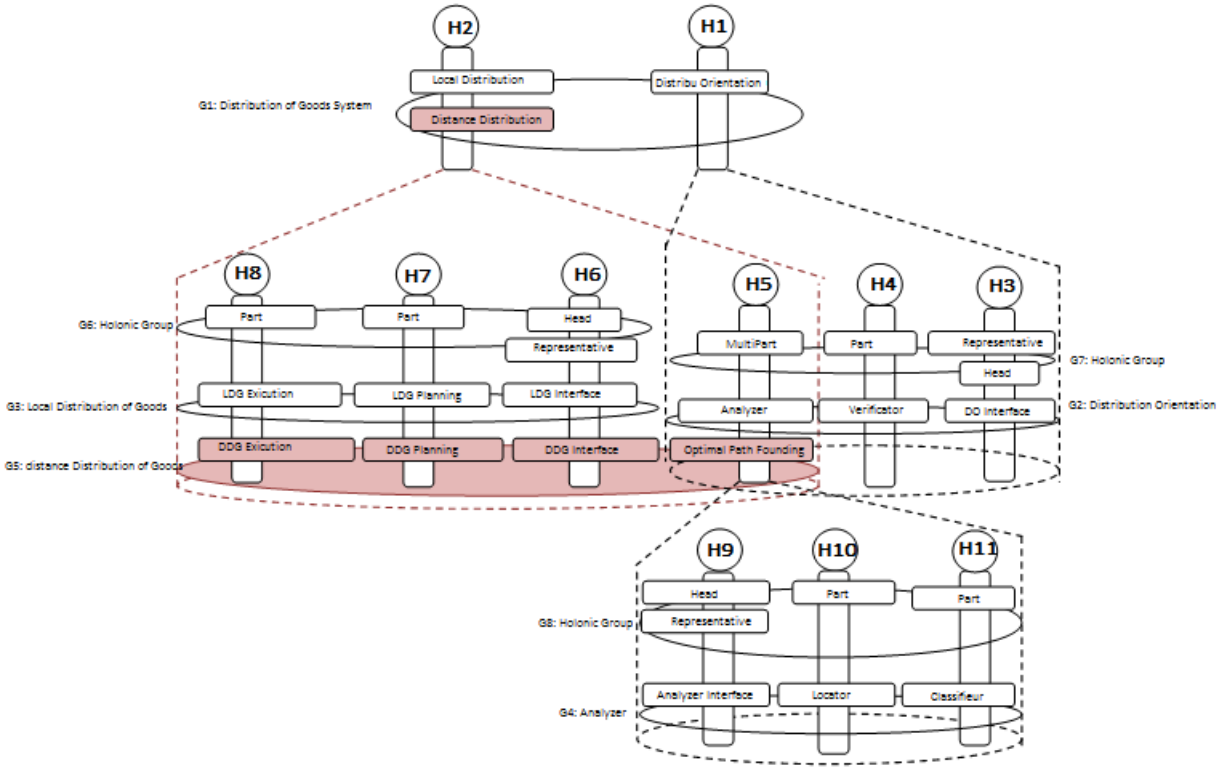


Figure 5.19: Structure Holonique du "*Distribution of Goods System*" dans sa version (V_1).

La connaissance des rôles obtenus nous permet de connaitre les interactions introduites dans la nouvelle version V_1. Pour cela, il suffit de récupérer les diagrammes de séquences décrits dans la phase (Description des scénarios d'interaction d'ASPECS) où les nouveaux rôles obtenus sont invoqués. L'ensemble de ces diagrammes de séquences représente le delta entre V_0 et V_1, tel que le montre la figure 5.20.

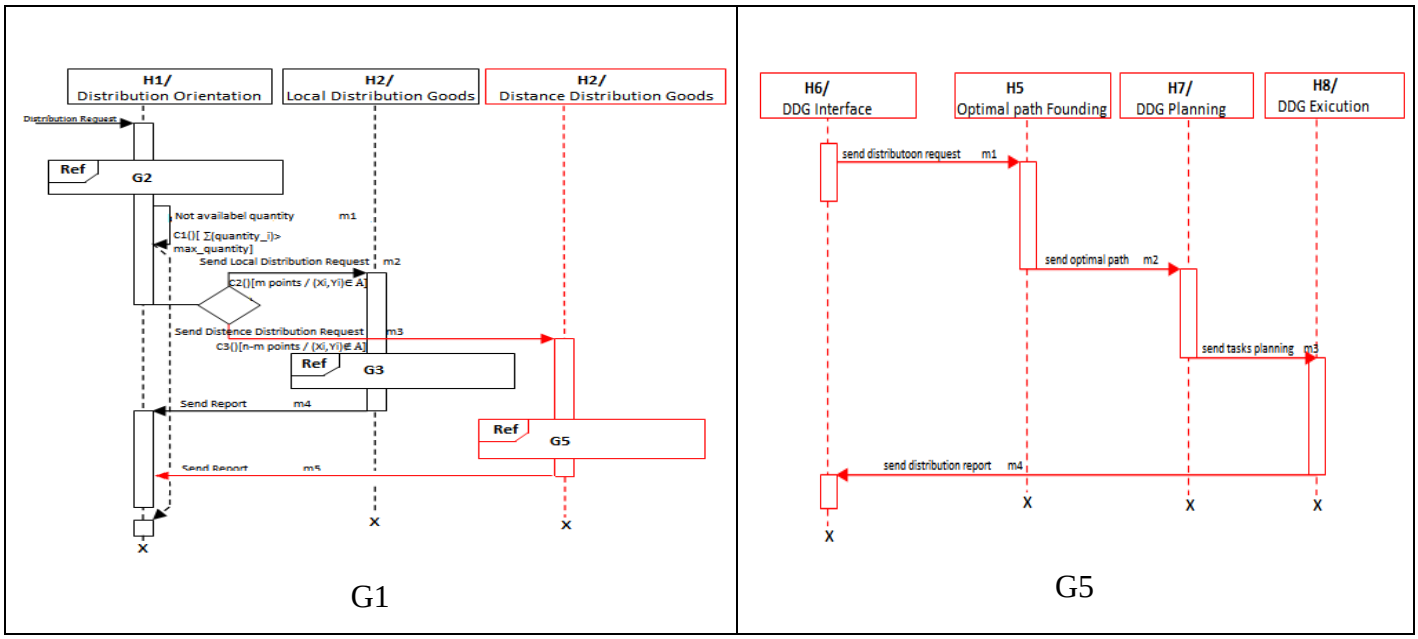


Figure 5.20: Delta (V_0, V_1).

Les nouvelles interactions introduites entre les membres de l'agent holonique ainsi que les scénarios générés suite à l'introduction de ces dernières n'ont pas été testés dans la version V_0.Ils doivent l'être dans la nouvelle version afin d'être sûr du degré de confiance du comportement de l'agent holonique dans sa nouvelle version. De plus, ce test permet de corriger, à temps, les éventuelles erreurs introduites dans cette nouvelle version et éviter qu'elles ne soient la cause de certains biais dans le développement du comportement de l'agent holonique.

3.3. Test de la version V_1

Le test de la version V_1 suivra les mêmes étapes que pour la version V_0, sauf que le modèle comportemental de V_1 sera le modèle comportemental de la version V_0 plus le delta (la différence entre les deux versions d'un point de vue interaction). En plus, la génération de cas de test sera appliquée seulement sur les nouveaux chemins introduits dans la nouvelle version.

a. M_1: Le modèle de la version V_1

Après avoir relié M_0 avec le delta (V_0, V_1), nous utilisons le même principe utilisé pour la génération du M_0. Nous obtenons le $HSDG_1(M_1)$ montré dans la figure 5.21. Le modèle comportemental (M_1) de la nouvelle version V_1 est caractérisé par de nouvelles interactions qui vont générer, automatiquement, de nouveaux scénarios qui seront couverts dans cette nouvelle version V_1.

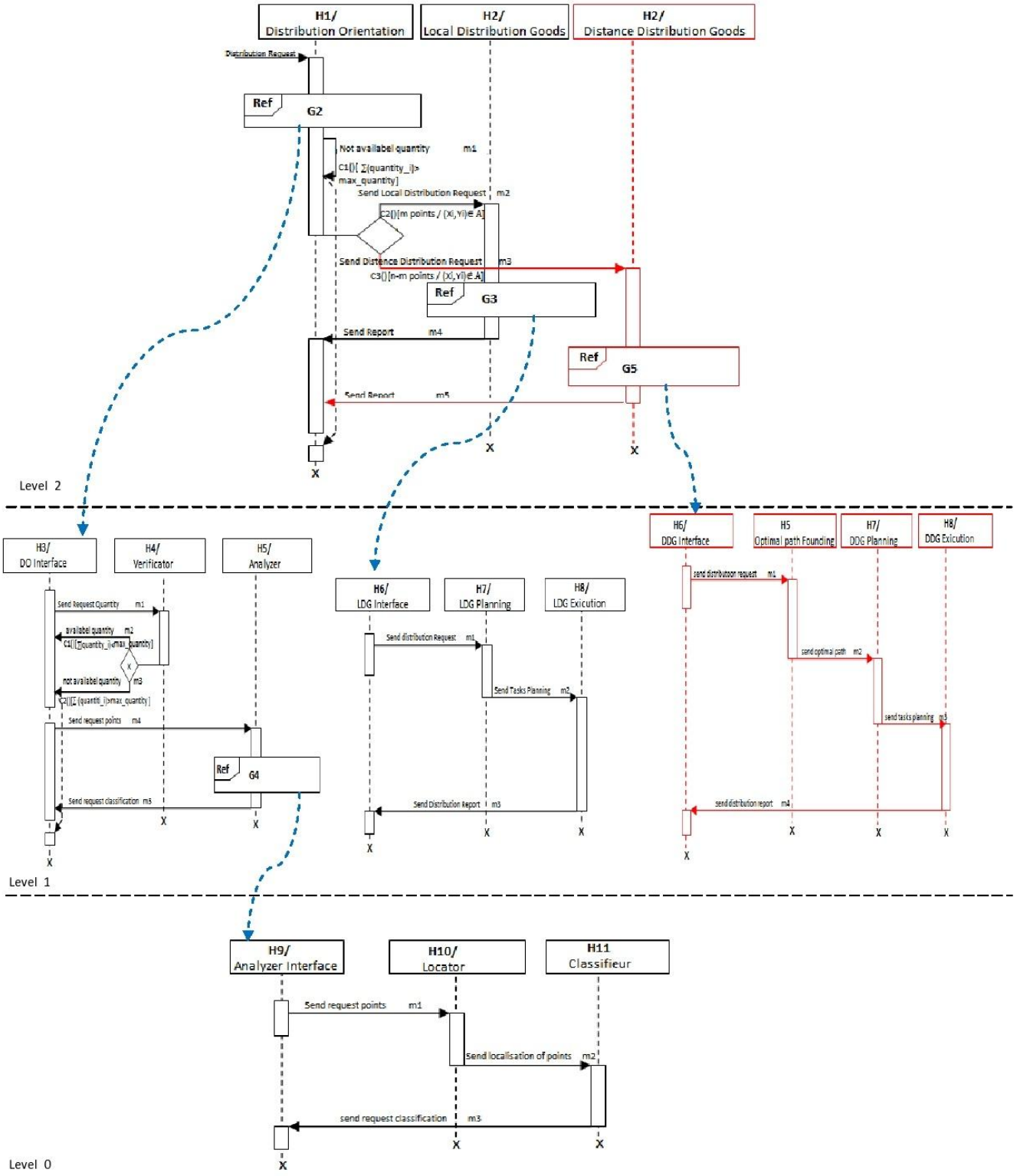


Figure 5.21: Diagramme de séquences AUML hiérarchique (M_1) de (V_1).

b. La génération des cas de test pour la version V_1

b.1.$HSDG_1$: Hierarchical Sequence Diagram Graphde la version V_1

Les figures 5.22 et 5.23 représentent le $HSDG_1$ de la version V_1 et les différents scénarios associés à ce graphe respectivement. Les nœuds en rouge représentent de nouvelles interactions introduites dans la version V_1. Les nouvelles interactions introduites sont à l'origine de la génération d'un nouveau scénario "Scénario 3" représenté dans la figure 5.23.

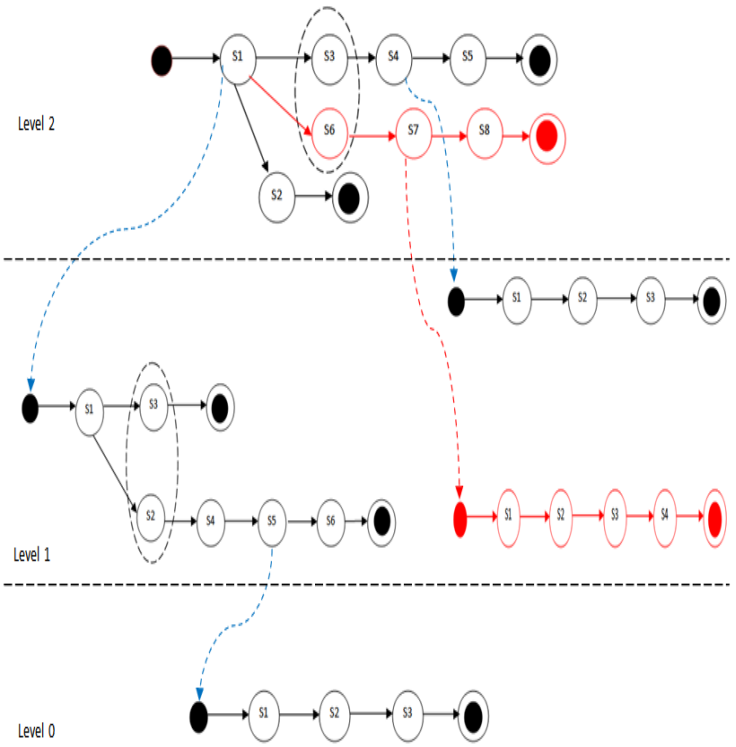


Figure 5.22: $HSDG_1$ de V_1

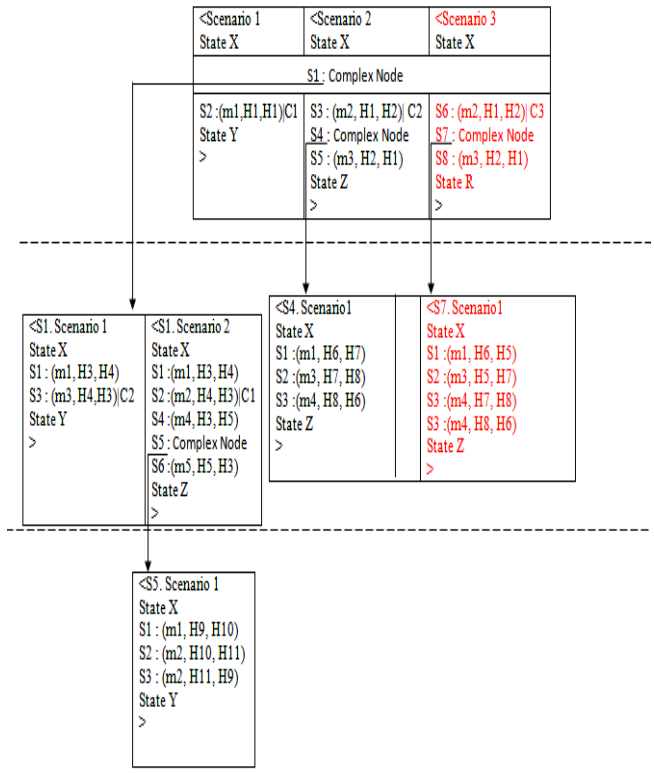


Figure 5.23: Les différents scénarios associés au $HSDG_1$

b.2.T1: Les cas de test de la version V_1

L'union de cas de test pouvant couvrir les chemins P_β nouvellement créés dans la nouvelle version V_1 avec des cas de test recouvrant les chemins existants Pα dans la version précédente V_0, nous donne un ensemble de cas de test pouvant couvrir tous les chemins de V_1. L'exécution du programme de génération des cas de test avec $HSDG_1$ associé à la version V_1 portera uniquement sur la génération des cas de test capables de couvrir les nouveaux chemins (P_β) introduits dans V_1. Les résultats de l'exécution du programme montrent les cas de test

capables de couvrir le scénario (Scénario 3) nouvellement crées dans la nouvelle version V_1 (Figure 5.24).

```
_____________________Test Case (Tβ) of new path (Pβ) created in V1_____________________
Preconditions: The system is ready to receive request
               The user send a Request
Test case of Scenario_3
T_Scenario_3 = {Input: (0 < the sum of (quantity_i) <max_quantity , ((n-m points / (xi >width_A , yi
>lengthe_A)) , (m points / ! (xi <width_A ,yi<lengthe_A)))) , Output: (Displays “available
distance request” , Displays the report) , Postcondition: Displays “Return to the initial state”}
5 possible test case for scenario_3
Case1
        Input: Quantity1= 3827, Quantity2= 4583, Quantity3= 3845,
Quantity4= 9463, Quantity5= 238
               Vector = ((30.60 , 82.53) , (67.29 , 68.83) , (46.02 , 29.91) ,
(66.60 , 56.79) , (31.44 , 52.82))
        Output: Displays “available distance request”
                 Displays the report
Postcondition: Displays “Return to the initial state”
Case2
        Input: Quantity1= 495, Quantity2= 4893, Quantity3= 937,
Quantity4= 610, Quantity5= 328
               Vector = ((54.50 , 52.45) , (37.18 , 36.20) , (63.01 , 58.11) ,
(39.34 , 27.68) , (48.81 , 76.50))
        Output: Displays “available distance request”
                 Displays the report
Postcondition: Displays “Return to the initial state”
Case3
        Input: Quantity1= 194, Quantity2= 2302, Quantity3= 281,
Quantity4= 201, Quantity5= 3921
               Vector = ((99.54 , 71.71) , (63.20 , 27.99) , (96.80 , 75.98) ,
(79.26 , 58.98) , (26.11 , 51.97))
        Output: Displays “available distance request”
                 Displays the report
Postcondition: Displays “Return to the initial state”
Case4
        Input: Quantity1= 3109, Quantity2= 921, Quantity3= 493,
Quantity4= 194, Quantity5= 81
               Vector = ((94.72 , 66.21) , (76.79 , 82.29) , (80.38 , 46.10) ,
(56.275 , 75.81) , (39.01 , 75.39))
        Output: Displays “available distance request”
                 Displays the report
Postcondition: Displays “Return to the initial state”
Case5
        Input: Quantity1= 950, Quantity2= 78, Quantity3= 9423,
Quantity4= 94, Quantity5= 638
               Vector = ((80.24 , 41.62) , (52.55 , 61.61) , (53.37 , 52.31) ,
(33.89 , 66.09) , (50.47 , 94.08))
        Output: Displays “available distance request”
                 Displays the report
Postcondition: Displays “Return to the initial state”
Postcondition: Displays “Enter your request”
```

Figure 5.24: Cas de test des nouveaux chemins créés dans V_1

L'application de notre approche de test sur l'étude de cas choisie nous a permis de montrer la nécessité de test par version pour les systèmes avec un comportement qui se développe avec le temps. En effet, le test par la version nous permet de couvrir tous les comportements acquis avec le temps et de corriger, à temps, toute erreur éventuelle qui pourrait biaiser le comportement de l’agent holonique.

4. Conclusion

Dans ce chapitre, nous avons appliqué notre approche de test sur un cas concret, ce qui nous a permis de montrer que notre approche offre plusieurs avantages, à savoir:

- L'approche considère l'évolution du comportement et de la structure de l'agent holonique et prend en compte sa nature hiérarchique.
- L'approche utilise la technique de test basé-modèle, qui offre plusieurs avantages.
- L'approche adopte une stratégie itérative pour le test des SMA et utilise la technique des algorithmes génétiques pour l'évaluation et le suivi du développement de l'agent holonique.
- L'approche supporte l'incrémentation lors de la génération du modèle comportemental ainsi que lors de la génération des cas de test. De cette manière, le temps de génération des cas de test est minimisé.
- L'approche prend en compte la compatibilité entre les scénarios de différents niveaux.
- L'approche est capable de générer des cas de test qui recouvrent chaque scénario individuellement, ce qui implique que chaque erreur détectée dans un scénario donné n'est pas associée à un autre scénario qui s'exécute en parallèle avec lui.
- L'approche a une bonne fonction de fitness qui a un double objectif. Elle permet, d'une part, la détection du comportement et de la structure de l'agent holonique et, d'autre part, de déterminer exactement le niveau et membre causant le développement.
- L'approche est basée sur un algorithme qui est capable de se comporter avec différents types de nœuds d'une manière appropriée, de manière à générer les cas capables de couvrir les chemins de test souhaités.

Conclusion générale

L'activité de test représente une tâche importante dans le processus d'assurance qualité des SMA. Malgré l'évolution rapide des SMA, le test des SMA comme systèmes autonomes est encore un domaine clé ouvert. En fait, seules quelques propositions portant sur le test des SMA ont été proposées dans la littérature. Bien qu'ils aient permis de réels progrès dans le domaine du test des SMA, ils ne prennent pas en compte l'évolution des SMA. En outre, ils ne tiennent pas compte des spécificités des agents holoniques (par exemple, l'évolution de leur comportement, leur structure et leur nature hiérarchique).

Dans cette thèse, nous avons proposé une nouvelle approche de test pour les agents holoniques pouvant s'adapter avec son comportement et sa structure qui se développent avec le temps et avec sa nature hiérarchique. Cette approche consiste à tester l'agent holonique par version successives ; ceci nous permet de couvrir tous les comportements acquis avec le temps et de corriger, à temps, toute erreur éventuelle qui pourrait biaiser le comportement de l'agent holonique. Cette approche se décompose en deux sous processus. Le premier processus porte sur la détection des nouvelles versions à tester. Dans ce processus, la fonction de fitness proposée, nous permet de suivre le développement de l'agent holonique et de déterminer avec exactitude le niveau et le membre ayant causé le développement. Ceci facilite la détermination du delta entre deux versions successives et nous fait gagner du temps en nous évitant la comparaison entre les versions. Le deuxième processus porte sur le test de chaque nouvelle version détectée. La nouvelle version de l'agent est analysée afin de générer un modèle comportemental sur lequel est basée la génération des cas de test. Le processus de génération des cas de test se concentre sur les nouvelles (et / ou modifiés) parties du

comportement de l'agent. De cette manière, la technique prend en charge une mise à jour incrémentielle des cas de test, qui est un problème crucial.

L'approche proposée offre plusieurs avantages. En particulier, elle :

- Considère l'évolution du comportement et de structure de l'agent holonique et prend en compte sa nature hiérarchique.
- Utilise la technique de Model-Based Testing, qui offre plusieurs avantages.
- Adopte une stratégie itérative pour le test des SMA et utilise la technique des algorithmes génétiques pour l'évaluation et le suivi du développement de l'agent holonique.
- Supporte l'incrémentation lors de la génération du modèle comportemental ainsi que lors de la génération des cas de test. De cette manière, le temps de génération des cas de test est minimisé.
- Prend en compte la compatibilité entre les scénarios de différents niveaux.
- Est capable de générer des cas de test qui recouvrent chaque scénario individuellement, ce qui implique que chaque erreur détectée dans un scénario donné n'est pas associée à un autre scénario qui s'exécutait en parallèle avec lui.
- Offre une bonne fonction de fitness qui a un double objectif. Elle permet, d'une part, la détection du développement du comportement et de la structure de l'agent holonique et, d'autre part, à déterminer exactement le niveau et membre causés le développement.
- Est basée sur un algorithme qui est capable de se comporter avec différents types de nœuds d'une manière appropriée, de manière à générer les cas de test capables de couvrir les chemins souhaités de test.

L'approche proposée, supportée par l'outil que nous avons développé, a été validée sur une étude de cas concrète: *Distribution of Goods System*. Selon les résultats obtenus, il serait intéressant d'intégrer notre outil dans le processus ASPECS pour supporter les tests d'évolution des HMAS. Comme perspectives, nous prévoyons à court et moyen termes :

- La généralisation du principe de test par versions pour tous les agents caractérisés par un comportement qui se développe avec le temps.

- L’optimisation du temps d’arrivée à toutes les versions possibles en se basant sur la dépendance qui existe entre les entrées nécessaires pour l’exécution du système et les directions de développement possible
- L’étude de la charge imposée à l’agent à la suite du fait de jouer plusieurs rôles, pour laquelle nous poserons comme règle de départ qu’avant d’obtenir un rôle, l’agent doit prendre en considération l’impact de l’obtention de ce dernier sur son objectif personnel, l’objectif de son super holon, l’objectif du système et la charge qui sera imposée suite à l’obtention du rôle.

Bibliographie

[Ada00a] Adam, E., Mandiau, R., and Kolski, C. 2000. Homascow: a holonic multi-agent system for cooperative work. In 11th International Workshop on Database and Expert Systems Applications, 247–253.

[Ada00b] Adam, E. 2000. Modèle d'organisation multi-agent pour l'aide au travail coopératif dans les processus d'entreprise : application aux systèmes administratifs complexes. PhD thesis, Univ. de Valenciennes et du Hainaut-Cambresis.

[Adr03] Adriana, G.,Vicente, B. 2003. Agent-Oriented Software Engineering IV, volume 2935 of Lecture Notes in Computer Science, chapter Towards a Recursive Agent Oriented Methodology for Large-Scale MAS, 25–35. Springer Berlin / Heidelberg.

[Adr04] Adriana, G., Vicente, B. 2004. Holons and agents. Journal of Intelligent Manufacturing, 15, 645–659.

[Ale02] Alexander, R., Bieman, M., Sudipto, G., Bixia, J. 2002. Mutation of Java Objects. In ISSRE'02 (Int. Symposium on Software Reliability Engineering), Annapolis, MD, USA.

[And03] Andrews, A., France, R., Ghosh, S., Craig, G. 2003. Test adequacy criteria for UML design models. Softw. Test., Verif. Reliab., 13(2) :95127.

[Apf97] Apfelbaum, L., Doyle, J. 1997. Model Based Testing. In the 10th International Software Quality Week Conference, CA, USA, 296-300.

[Bac04] Bach,J., Shroeder,P. 2004. Pairwise Testing - A Best Practice That Isn't. In Proceedings of 22nd Pacic Northwest Software Quality Conference, 180196.

[Bar01] Barbat,B., Candea,B., Zamfirescu, C. 2001. Holons and agents in robotic teams. a synergistic approach. In Proceedings of ENAIS'2001, 654–660, ISBN 3-906454-25-8.

[Bot08] Botti, V., Giret, A. 2008. ANEMONA: A Multi-Agent Methodology for Holonic Manufacturing Systems. Springer Series in Advanced Manufacturing 2008th Edition. ISBN:978-1-84800-309-5.

[Bur97] Burckhart, R., Company, D. 1997. Schedules of activity in the swarm simulation system. In Proceedings in the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Workshop on OO Behavioral Semantics.

[Bür98] Bürckert, H.J., Fischer,K., Vierke, G. 1998. Transportation scheduling with holonic MAS - the teletruck approach. In Conference on Practical Applications of Intelligent Agents and Multiagent, 577–590.

[Bur02] Burnstein,I. 2002. "Practical software testing: A Process-oriented approach", Ed. Springer, 2002.

[Cai04] Caire, G., Cossentino, M., Negri, A., Poggi, A., Turci, P. 2004. Multiagent Systems Implementation and Testing. In Proc. of the 4th From Agent Theory to Agent Implementation Symposium, AT2AI-4

[Che01] Chevalley, P. 2001. Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach. In Eighth Asia-Pacific Software Engineering Conference, Macao, China.

Bibliographie

[Che04] Chella, A., Cossentino, M., Sabatucci, L., Seidita, V. 2004. From PASSI to Agile PASSI: tailoring a design process to meet new needs. In proceedings of the IEEE/WIC/ACM International Joint Conference on Intelligent Agent Technology (IAT'04), 471-474. IEEE Computer Society.

[Cor01] Correa e Silva, F., Kelly, C.2001. Hybrid Multi-Agent Systems : An Approach for Complex Systems Design. PhD thesis, Université Joseph Fourier- Grenoble 1.

[Col04] Colin, S., Legeard, B., Peureux, F. 2004. Preamble Computation in Automated Test Case Generation using Constraint Logic Programming. The Journal of Software Testing, Verication and Reliability, 14(3) : 213-235.

[Cos04] Cossentino, M., Sabatucci, L., Chella, A. 2004. Patterns reuse in the PASSI methodology. In Engineering Societies in the Agents World IV, 4 th International Workshop, ESAW, volume XIII of Lecture Notes in Artificial Intelligence. Springer-Verlag,. Agile passi, 294-310

[Coe06] Coelho, R., Kulesza, U., vonStaa, A., Lucena, C. 2006. Unit testing in multi-agent systems using mock agents and aspects. In SELMAS 2006: Proceedings of the International Workshop on Software Engineering for Large-scale Multi-agent Systems,83-90. ACM Press, New York.

[Deh15] Dehimi, N.,Mokhati, F., Badri,M. 2015. Testing HMAS-based applications: An ASPECS-based approach. In Engineering Applications of Artificial Intelligence, Volume 46, Part A, 232–257 , Elsevier journal.

[Dik05] Dikenelli, O., Erdur, R.C., Gumus, O. 2005. Seagent: a platform for developing semantic web based multi agent systems. In: AAMAS 2005: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, 1271–1272. ACM Press, New York.

[Eki08] Ekinci, E.E., Tiryaki, A.M., Cetin, O., Dikenelli, O. 2008. Goal-Oriented Agent Testing Revisited. In Proc. of the 9th Int. Workshop on Agent-Oriented Software Engineering, 85–96.

[Elf01] Elfar, I. K., Whittaker, J. A. 2001. Encyclopedia of Software Engineering, chapter Model-Based Software Testing, 825-837.Wiley.

[Fer98] Ferber, J., Gutknecht, O., Aalaadin. 1998. A meta-model for the analysis and design of organizations in multi-agent systems, In ICMAS'98.

[Fer95] Ferber, J.1995. Les Systèmes Multi-Agents : Vers une Intelligence Collective. InterEditions.

[Fer04] Ferber, J., Gutknecht, O., Michel, F. 2004. From agents to organizations: An organizational view of multi-agent systems. In AOSE-IV@AAMAS03, LNCS, volume. 2935, 214–230. Springer Verlag.

[Fis03] Fisher, K., Michael, S., and Jörg, S. 2003. Holonic multiagent systems : A foundation for the organisation of multiagent systems. In Holonic and Multi-Agent Systems for Manufacturing, volume 2744 of LNCS, 71–80. Springer Berlin, Heidelberg.

[Fra88] Frankl, k., Weyuker, E. 1988. An Applicable Family of Data Flow Testing Criteria. IEEE Trans. Softw. Eng., 14(10) :1483-1498.

[Gam00] Gamma, E., Beck, K. 2000. JUnit: A Regression Testing Framework, http://www.junit.org.

[Ger99] Gerber,C., Siekmann, J., Vierke, G.1999. Holonic multi-agent systems. Technical Report DFKI-RR-99-03, Deutsches Forschungszentrum für Künztliche Inteligenz - GmbH, Postfach 20 80, 67608 Kaiserslautern, FRG.

[Gho00] Ghosh, S., Mathur, A. 2000. Interface Mutation to assess the adequacy of tests for components and systems. In TOOLS, Santa Barbara, CA, USA.

[Goo75] Goodenough, J., Gerhart, S. 1975. Toward a theory of test data selection, IEEE transactions on software engineering, 1(2), pp. 156-173

[Got98] Gotlieb,A., Botella, B., Rueher, B. 1998. Automatic Test Data Generation using Constraint Solving Techniques. In ACM SIG-SOFT, editor, Proceedings of International Symposium on Software Testing and Analysis (ISSTA), volume 2, 53-62

[Gru92] Gruber, T.1992. Ontolingua: A mechanism to SupportPortable Ontologies. http://www-ksl.stanford.edu/knowledge-sharing/papers/README.html#ontolingua-long.

Bibliographie

[Gut00] Gutknecht, O., Ferber, J. 2000. The MADKIT Agent Platform Architecture. In Agents Workshop on Infrastructure for Multi-Agent Systems , 48-55.

[Gut01] Gutknecht, O., Ferber, J., Michel, F. 2001. Integrating tools and infrastructures for generic multi-agent systems. In: AGENTS 2001: Proceedings of the Fifth International Conference on Autonomous Agents, 441–448. ACM, New York .

[Ham93] Hamlet, R.1993. Test du logiciel & confiance, Génie logiciel et systèmes experts, 30.

[Hay01] Hayhurst, K., Veerhusen, D. 2001. A Practical Approach To Modified Condition/decision Coverage. In Proceedings of 20th Digital Avionics Systems Conference (DASC), Daytona Beach, Florida, USA, volume 1, 1B2/1-1B2/10.

[Hel97] Helke, S., Neustupny, T., Santen, T. 1997. Automating Test Case Generation from Z Specications with Isabelle. In ZUM, 52-71.

[Hil00a] Hilaire, V. Koukam, A., Gruer, P., Müller, J. 2000. Formal specification and prototyping of multi-agent systems. In Andrea Omicini, Robert Tolksdorf, and Franco Zambonelli, editors, Engineering Societies in the Agents' World, number 1972 in Lecture Notes in Artificial Intelligence. Springer Verlag.

[Hil00b] Hilaire, V. 2000. Vers une approche de spécification, de prototypage et de vérification de Systèmes Multi-Agents. PhD thesis, Université de Technologie de Belfort-Montbéliard.

[Hou11] Houhamdi, Z. 2011. Multi-Agent System Testing: A Survey, (IJACSA) International Journal of Advanced Computer Science and Applications, volume 2.

[Hug04] Huget, M., Demazeau, Y. 2004. Evaluating multiagent systems: a record/replay approach. Intelligent Agent Technology, 2004. (IAT 2004). Proceedings. IEEE/WIC/ACM International Conference : 536-539.

[IEEE90] IEEE standard glossary of software engineering terminology - IEEE Std 610.12-1990 available athttp://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=49C6B9AB6A35DD79A7D7E87A1BD7273A?

[Jin00] Jin-Lung, C., Duncan, C. ,McFarlane. 2000. A Holonic Component-Based Approach to Reconfigurable Manufacturing Control Architecture. DEXA Workshop , 219-223.

[Jug13] Jugnesh, K., Yadav, A. July. 2013. Novel Approach to Generate Test Cases from UML Sequence Diagrams. In the International Journal of IT, Engineering and Applied Sciences Research (IJIEASR) ISSN: 2319-4413 volume 2, No. 7, 9-14.

[Kos04] Kosmatov, N., Legeard, B., Peureux, F., Utting, M. 2004. Boundary Coverage Criteria for Test Generation from Formal Models. In ISSRE, 139150.

[Lam05] Lam, D.N., Barber, K.S. 2005. Debugging agent behavior in an implemented agent system. In Bordini, R.H., Dastani, M.M., Dix, J., El FallahSeghrouchni, A. (eds.) PROMAS 2004. LNCS (LNAI), volume 3346, 104-125. Springer, Heidelberg.

[Lap92] Laplante, A. 1992. Real-Time Systems Design and Analysis : An Engineer's Handbook. IEEE Press, Piscataway, NJ, USA.

[Lei02] Leitao, P., Restivo, F. 2002. Holonic Adactive Production Control Systems. In Proceedings of special session on Agent-based Intelligent Automation and Holonic Control Systems of the 28th Anual Conference of the IEEE Industrial Society. 2968–2973. Sevilla.

[Mat03] Matthieu, A. 2003. MOCA : un modèle componentiel dynamique pour les systèmes multi-agents organisationnels. PhD thesis, Université de Neuchâtel

[Mas07] Massimo, C., Nicolas, G., Stéphane, G., Vincent, H., Abder K. 2007. A Holonic Metamodel for Agent-Oriented Analysis and Design. In HoloMAS , 237-246

[Mas09] Massimo, C., Nicolas, G., Vincent, H., Stéphane, G., Abderrafiâa, K.2009. ASPECS: an agent-oriented software process for engineering complex systems :How to design agent societies under a holonic perspective. In Auton Agent Multi-Agent SystSpringerScience+Business Media, LLC.260-304.

[DeM04] DeMoura,L., Hamon, G., Rushby, J. 2004. Generating E-cient Test Sets with a Model Checker. In 2nd International Conference on Software Engineering and Formal Methods, 261-270, Beijing, China, 2004. IEEE Computer Society.

Bibliographie

[Mic95] Michael, W., and Nicholas R. Jennings. 1995. Intelligent Agents : Theory and Practice. Knowledge Engineering Review, 10(2), 115–152.

[Mil78] Millo, R., Lipton, R., Sayward, R. 1978. Hints on Test Data Selection : Help For The Practicing Programmer. In IEEE Computer, volume 11, 34 - 41.

[Mor09] Moreno, M., Pavon, J., Rosete, A. 2009. Testing in agent oriented methodologies. In: Omatu, S., Rocha, M.P., Bravo, J., Fern´andez, F., Corchado, E., Bustillo, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, volume 5518, 138–145. Springer, Heidelberg .

[Mye04] Myers, G., Sandler, C. Badgett, T., Thomas, M. 2004. The Art of Software Testing. In Second Edition, Wiley, 2004.

[Ngu08a] Nguyen, C.D. 2008. Testing Techniques for Software Agents, PhD thesis, International Doctorate School in Information and Communication Technologies - University of Trento.

[Ngu08b] Nguyen, C.D., Perini, A., Tonella, P. 2008. Ontology-based Test Generation for Multi Agent Systems. In Proceedings of the International Conference on Autonomous Agents and Multiagent Systems, 1315-1320.

[Ngu10] Nguyen, C.D., Perini, A., Tonella, P. 2010. Goal oriented testing for MASs. Int. J. Agent-Oriented Software Engineering 4(1), 79–109.

[Ngu12] Nguyen, C.D., Miles, S., Perini, A., Tonella, P., Harman, M., Luck, M. 2012. Evolutionary testing of autonomous software agents. In Autonomous Agents and Multi-Agent Systems, volume 25(2), 260-283.

[Nic07] Nicolas, G. 2007. Système multi- agent holonique : de l'analyse à l'implantation. PhD thesis, Université de Franche-Comté et de l'Université de Technologie de Belfort-Montbéliard.

[Nic08] Nicolas, G., Galland, S.,Vincent, H.,Koukam, A. 2008. An organisational platform for holonic and multiagent systems. In PROMAS-6@AAMAS'08, Estoril, Portugal, 104-119.

[Nta88] Ntafos, S. 1988. A comparaison of some structural testing strategies, IEEE transactions of software engineering. 14(6): 868-873.

[Nun05] Nunez, M., Rodrıguez, I., Rubio, F. 2005. Specification and testing of autonomous agents In e-commerce systems. Software Testing, Verification and Reliability 15(4), 211-233.

[Off99] Offutt, J., Xiong, Y., Liu, S. 1999. Criteria for Generating Specication-Based Tests. In ICECCS '99 : Proceedings of the 5th International Conference on Engineering of Complex Computer Systems, 119.

[Ost88] Ostrand, T., Balcer, M. 1988. The category partition method for specifying and generating functional test, communication of the ACM, 31(6):676-686.

[Pad04] Padgham, L., Winikoff, M. 2004. Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons, Chichester.

[Pav05] Pavón, J., Gómez-Sanz, J., Fuentes, R. 2005. The INGENIAS methodology and tools. In Brian Henderson-Sellers and Paolo Giorgini, editors, Agent-Oriented Methodologies, 236–276. Idea Group Publishing, NY, USA,. ISBN 1-59140-581-5.

[Pau93] Paulson, L. 1993. The Isabelle Reference Manual. Technical Report 283.

[Pau09] Paulo, L., Paul, V., Emmanuel, A. 2009. Self-Adaptation for Robustness and Cooperation in Holonic Multi-Agent Systems.T. Large-Scale Data- and Knowledge-Centered Systems 1, 267-288.

[Pou09] Poutakidis, D., Winikoff, M., Padgham, L., Zhang, Z. 2009. Debugging and Testing of Multi-Agent Systems using Design Artefacts. In: Multi-Agent Programming, 215–258.

[Pre01] Pretschner, A., Lötzbeyer, H. 2001. Model Based Testing with Constraint Logic Programming : First Results and Challenges. In Proceedings of 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verication (WAPATV), 1-9, Toronto.

[Rod05] Rodriguez, S. 2005. From analysis to design of Holonic Multi-Agent Systems: A Framework, Methodological Guidelines and Applications. PhD thesis, Université de Technologie de Belfort-Montbéliard.

Bibliographie

[Rod06] Rodriguez,S., Nicolas, G., Vincent, H., Stephane, G., and Koukam, A. 2006. An analysis and design concept for self-organization in Holonic Multi-Agent Systems. In the 4th International Workshop on Engineering Self-Organizing Applications (ESOA'06), Eds. Sven Brueckner, Salima Hassas, Mark Jelasity, Danial Yamins, 62-75, Japan. Springer-Verlag

[Rod07] Rodriguez, S., Gaud, N., Hilaire, V., Galland, S., Koukam, A. 2007. An analysis and design concept for self-organization in holonic multi-agent systems. In S. Bruckner, S. Hassas, M. Jelasity, and D. Yamins, editors, Engineering Self-Organising Systems, volume 4335 of LNAI, 15–27. Springer-Verlag.

[Rou02] Rouff, C. 2002. A Test Agent for Testing Agents and Their Communities, Aerospace Conference Proceedings, 2002. IEEE volume 5, 5 - 2638 volume 5.

[Sal00] Saleh, K., Ural, H., Williams, A. 2000. Test generation based on control and data dependencies within system specications in SDL. In Computer Communications 23, 609627.

[Sch04] Schillo, M. Multiagent Robustness. 2004. Autonomy vs. Organisation. PhD thesis, Department of Computer Science, Universität des Saarlandes.

[Ser08] Serrano, E., Botia, J.A. 2008. Infrastructure for forensic analysis of multi-agent systems. In: Hindriks, K.V., Pokahr, A., Sardina, S. (eds.) ProMAS 2008. LNCS, volume 5442, pp. 168–183. Springer.

[Ser09] Serrano, E., Gomez-Sanz, J.J., Botıa, J.A., Pavon, J. 2009. Intelligent data analysis applied to debug complex software systems. Neurocomputing 72(13-15), 2785–2795.

[Sou00] Souza, S., Maldonado, J., Fabbri, S., Masiero, P. 2000. Statecharts specifications : A family of coverage testing criteria. In CLEI 2000 - XXVI Conferncia Latinoamericana de Informatica, 2000, Monterrey,http://www.labes.icmc.usp.br/plavis/les/references/uepg/statecharts_specications_a_family_of_coverage_testing_criteria.pdf.

[SPEM07] SPEM, 2007. Software Process Engineering Metamodel Specification, v2.0, Final Adopted Specification, ptc/07-03-03. Object Management Group (OMG).

[Tir07] Tiryaki, A.M., Oztuna, S., Dikenelli, O., Erdur, R.C. 2007. SUNIT: A unit testing framework for test driven development of multi-agent systems. In: Padgham, L., Zambonelli, F. (eds.) AOSE VII / AOSE 2006. LNCS, volume 4405, pp. 156–173. Springer, Heidelberg.

[Uli02] Ulieru, M., Geras,A. 2002. Emergent holarchies for e-health applications : a case in glaucoma diagnosis. In IEEE IECON'02, volume 4, 2957–2961.

[Utt06] Utting, M., Pretschner, A., Legeard, B. 2006. A taxonomy of model-based testing. Technical Report 04/2006, Department of Com- puter Science, The Universiy of Waikato (New Zealand).

[Van98] Van Brussel, H., Wyns, J., Valckenaers, P., Bongaerts, L., and Peeters, P. 1998. Reference architecture for holonic manufacturing systems : Prosa. Computers in Industry, 37, 255–274.

[Wei88] Weis, S., Weyuker, E. 1988. An extended domain based model of software reliability, Transactions on software engineering, 14(10):1512-1524.

[Wie01] Wiegers, K., 2001. Peer Reviews in Software: A Practical Guide, Addison-Wesley Information Technology Series.

[Woo90] Woodward M .1990. Mutation testing: an evolving technique, in colloquium on software testing for critical systems.

[Xan00] Xanthakis, S., Régnier, P., Karapoulios, C. 2000. Le test des logiciels, Hermes sciences publications.

[Zha07] Zhang, Z., Thangarajah, J., Padgham, L. 2007. Automated unit testing for agent systems. In: 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Barcelona, Spain, 10–18;

[Zha08] Zhang, Z., Thangarajah, J., Padgham, L. 2008. Automated unit testing intelligent agents in pdt. In: AAMAS (Demos), 1673–1674.

[Zha09a] Zhang, Z., Thangarajah, J., Padgham, Lin. 2009. Automated Testing for Intelligent Agent Systems. In AOSE 2009. Budapest, Hungary, 66-79.

[Zha09b] Zhang, Z., Thangarajah, J., Padgham, L. 2009. Model based testing for agent systems.In: AAMAS, volume 2, 1333–1334.