RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITÉ 20 AOUT 1955-SKIKDA

Faculté des Sciences

Département d'Informatique

Pour l'obtention de diplôme de **MASTER EN INFORMATIQUE**

(Option : Génie Logiciel)

(Option : Système d'information)

THEME

La génération des cas de test basée sur le diagramme de séquence AUML

Mémoire présenté par :

Benoumechiara Assia

Bouzenad Halima

Encadré par :

Dr KISSOUM YACINE

Promotion Juin 2022.

REMERCIEMENTS

Avant tout, merci mon DIEU de m'avoir donné le courage, la
Volonté et la patience de mener ce travail à terme.
Je tien à remercier infiniment mon encadreur
Dr. KISSOUM Yacine, pour son encadrement avec patience et pour son aide et ces conseils.
Je remercie également les membres du jury,
qui ont accepté d'évaluer ce travail.

Résumé

Le test des systèmes Multi-Agent (SMA) a besoin de techniques appropriées pour évaluer les comportements autonomes de l'agent aussi bien que les propriétés de distribution, sociales et délibératives, qui sont particulières à ces systèmes. Parmi ces techniques, nous trouvons le test basé sur les modèles, il est basé sur un modèle de système afin de produire des cas de test abstraits. Pour que ces derniers puissent être soumis au système sous test, les cas de test abstraits doivent être transformés en des cas de test concrets.

Dans ce mémoire on a générer des cas de test basée sur le diagramme de séquence AUML qui est transformé en suite en diagramme de flux de données. L'approche proposée et appliquée en utilisant le problème de vente de livre comme cas d'étude.

Mots-clés : le test des systèmes Multi-agents, Test basé sur les modèles, Cas de test concrets, Diagramme de séquence AUML, Diagramme de flux de données.

ملخص

يتطلب اختبار الأنظمة متعددة العوامل (SMA) تقنيات مناسبة لتقييم السلوكيات المستقلة للوكيل بالإضافة إلى الخصائص التوزيعية والاجتماعية والتداولية الخاصة بهذه الأنظمة. من بين هذه التقنيات نجد اختبارًا قائمًا على النموذج ، يعتمد على نموذج نظام لإنتاج حالات اختبار مجردة. من أجل تقديمها إلى النظام قيد الاختبار، يجب تحويل حالات الاختبار المجردة إلى حالات اختبار ملموسة.

في هذه الأطروحة أنشأنا حالات اختبار بناءً على المخطط التسلسلي (AUML) والذي يتم تحويله بعد ذلك إلى مخطط تدفق بيانات. المنهج المقترح والمطبق باستخدام مشكلة بيع الكتاب كدراسة حالة.

كلمات مفتاحية

اختبار النظام متعدد العوامل، الاختبار القائم على النموذج، حالات الاختبار الملموسة، المخطط التسلسلي AUML، مخطط تدفق البيانات.

Abstract :

The testing of Multi-Agent Systems (MAS) needs appropriate techniques to assess the agent's autonomous behaviors as well as the distributional, social and deliberative properties that are particular to these systems. Among these techniques we find model-based testing, it is based on a system model in order to produce abstract test cases. In order for these to be submitted to the system under test, the abstract test cases must be transformed into concrete test cases.

In this thesis we have generated test cases based on the AUML sequence diagram which is then transformed into a data flow diagram. The approach proposed and applied using the book selling problem as a case study.

Key words :

The test of Multi-agent systems, Test based on models, Concrete test cases, AUML sequence diagram, Data flow diagram.

Table de matières

Chapitre3 : L'approche proposée et Etude de cas

Chapitre 4 : L'implémentation

Liste des figures :

Chapitre 1 : Les systèmes multi agents

Chapitre 2 : Test et systèmes multi-agents

Chapitre 3 : L'approche proposée et Étude de cas

Chapitre 4 : L'implémentation

Liste des tableaux :

Chapitre 1 : Les systèmes multi agents

Chapitre 2 : Test et systèmes multi-agents

Introduction Générale :

Le test en informatique désigne une procédure de vérification partielle d'un système. Son objectif principal est d'identifier un nombre maximal de comportements problématiques du logiciel. Il permet ainsi, dès lors que les problèmes identifiés seront corrigés, d'en augmenter la qualité.

D'une manière plus générale, le test désigne toutes les activités qui consistent à rechercher des informations quant à la qualité du système afin de permettre la prise de décisions.

Un test ressemble à une expérience scientifique. Il examine une hypothèse exprimée en fonction de trois éléments : les données en entrée, l'objet à tester et les observations attendues. Cet examen est effectué sous conditions contrôlées pour pouvoir tirer des conclusions et, dans l'idéal, être reproduit.

Par ailleurs, les systèmes multi-agents cherchent à appréhender la coordination de processus autonomes. Un agent est une "entité computationnelle", comme un programme informatique ou un robot, qui peut être vue comme percevant et agissant de façon autonome sur son environnement. On peut parler d'autonomie parce que son comportement dépend au moins partiellement de son expérience. Un système multi-agents (SMA) est constitué d'un ensemble de processus informatiques se déroulant en même temps, donc de plusieurs agents vivant au même moment, partageant des ressources communes et communicant entre eux. Le point clé des systèmes multi-agents réside dans la formalisation de la coordination entre les agents.

L e test des SMA est une tâche provocante parce que ces systèmes sont distribués, autonomes, et délibératifs. Ces caractéristiques très particulières des agents logiciels rendent l'application des méthodes de test existantes difficile. Il y a des questions au sujet de communication et d'interopérabilité sémantique, aussi bien que la coordination. Tous ces dispositifs sont difficiles non seulement a modélisé et a programmé, mais a testé aussi.

Les agents sont une technologie prometteuse pour traiter le développement de système de plus en plus complexe. Un agent peut avoir beaucoup de manières de réaliser une tâche donnée, et il choisit la manière la plus appropriée de traitement. Un agent mobile est une classe particulière d'agent avec la possibilité pendant l'exécution de déplacer d'une place à l'autre où il peut reprendre son exécution. Cette mobilité représente de nouveaux défis pour les systèmes dynamiques en toutes les phases du cycle de vie, comme la modélisation, l'implémentation et le test. Il existe plusieurs méthodes et techniques pour le test des SMA, Parmi ces techniques, nous trouvons le test basé sur les modèles, il est basé sur un modèle de système afin de produire des cas de test abstraits. Pour que ces derniers puissent être soumis au système sous test, les cas de test abstraits doivent être transformés en des cas de test concrets. Notre approche consiste à appliquer le test basé a fin de générer les cas de test automatique basé sur le diagramme de séquences AUML.

Le présent mémoire est organisé en quatre chapitres structurés comme suit :

- **Système multi-agents :** une synthèse de l'état de l'art sur les systèmes multi-agents est décrite dans ce chapitre qui se compose de deux parties :

  - ✓ La première partie concerne l'agent à savoir : sa définition, ses caractéristiques, les types d'agents.

  - ✓ L'autre partie concerne les systèmes multi agents à savoir : Leur définition, leurs caractéristiques, leur modélisation, leur méthodologie et leur implémentation.

- **Test des systèmes multi-agents :** une synthèse de l'état de l'art de test est décrite dans ce chapitre qui se compose de deux parties :

  - ✓ La première partie concerne le test de logiciel en général à savoir : sa définition, ses niveaux, ses méthodes.

  - ✓ La deuxième partie concerne le test basé model à savoir : La définition, La modélisation, La génération des cas de test, et L'exécution des cas de test.

- **L'approche proposée et Étude de cas** : Dans ce chapitre on présente notre approche proposée la génération des cas de test basé sur le diagramme de séquence ainsi que ses différentes étapes : La modélisation, la génération des cas de test, la concrétisation et l'exécution des cas de test.

- **L'implémentation :** ce dernier chapitre "réalisation" présente le résultat obtenu après implémentation ainsi que l'environnement et les outils déployés pour le développement de notre application.

Chapitre 01 : Les systèmes multi agents (SMA)

1 Introduction :

Les systèmes multi agents (SMA) ont été appliqués pour la résolution des problèmes d'intelligence artificielle dans les années 70. En effet à cette époque, l'évolution des domaines d'applications de l'intelligence artificielle a investi des univers complexes et hétérogènes tels que l'aide à la décision, la reconnaissance et l'interprétation des formes, la conduite et le contrôle du processus industrielle etc…

L'utilisation des SMA n'a pas été limitée, seulement, à l'intelligence artificielle mais a été élargie à plusieurs autres domaines tels que les systèmes distribués et la robotique. Ceci ayant été possible, grâce à des aspects sociaux introduits suite à un concept-clé particulier à ce paradigme que sont les interactions.

Dans ce chapitre nous présenterons la définition, les caractéristiques et la classification des agents. Ensuite nous présenterons les systèmes multi agents et certaines propriétés comme les caractéristiques, les interactions et les langages de communication des agents. Nous terminons par les méthodologies orientées agents et les plates-formes multi agents.

2 Notion d'agent :

2.1 Définitions d'un agent :

Il n'y a pas une seule définition pour la notion d'agent. Dans ce qui suit, on présente les définitions les plus courantes.

♣ « Un agent est une entité qui perçoit son environnement et agit sur celui-ci »
[Russell, 1997].

♣ « Un agent est un système informatique, situé dans un environnement, et qui agit d'une façon autonome pour atteindre les objectifs (buts) pour lesquels il a été conçu »
[Wooldrige et Jennings, 1995].

♣ « Un agent est une entité qui fonctionne continuellement et de manière autonome dans un environnement où d'autres processus se déroulent et d'autres agents existent.»
[Shoham, 1993].

♣ « Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi agent, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents »
[Ferber, 1995].

2.2 Les caractéristiques multidimensionnelles d'un agent :

En partant de l'ouvrage de [Wooldrige et Jennings, 1995], et des définitions citées, on peut identifier les caractéristiques suivantes pour la notion d'agent :

- **Situé :** l'agent est capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement.

- **Autonome :** l'agent est capable d'agir sans l'intervention d'un tiers (humain ou agent) et contrôle ses propres actions ainsi que son état interne.

- **Proactif :** l'agent doit exhiber un comportement proactif et opportuniste, tout en étant capable de prendre l'initiative au bon moment

- **Social :** l'agent doit être capable d'interagir avec des autres agents (logiciels ou humains) afin d'accomplir des tâches ou aider ces agents à accomplir les leurs.

- **Intentionnalité :** un agent intentionnel est un agent guidé par ses buts, une intention est la déclaration explicite des buts et des moyens d'y parvenir. Elle exprime donc la volonté d'un agent d'atteindre un but ou d'effectuer une action.

- **Rationalité :** un agent rationnel est un agent qui suit le principe suivant "Si un agent sait qu'une de ses actions lui permet d'atteindre un de ses buts, il la sélectionne". La notion de rationalité se rapporte au comportement cognitif de l'agent. Ce terme qualifie l'utilisation efficace des ressources par l'agent.

- **Engagement :** La notion d'engagement est une qualité essentielle des agents coopératifs. Un agent coopératif planifie ses actions par coordination et négociation avec les autres agents. En construisant un plan pour atteindre un but, l'agent se donne les moyens d'y parvenir et donc s'engage à accomplir les actions qui satisfont ce but : l'agent croit qu'il est en mesure d'exécuter tout le plan qu'il a élaboré, ce qui le conduit (ainsi que les autres agents) à agir en conséquence.

- **Adaptabilité :** Un agent adaptatif est un agent capable de contrôler ses aptitudes (communicationnelles, comportementales) selon l'agent avec qui il interagit. Un agent adaptatif est un agent d'un haut niveau de flexibilité.

- **Intelligence :** Un agent intelligent est un agent cognitif, rationnel, intentionnel et adaptatif.

2.3 Les type d'agent:

Il existe trois types d'agent :

a) ***Agent réactif :*** est un agent sans intelligence (sans anticipation, sans planification), il réagisse par stimulus-réponse à l'état courant de l'environnement. Des comportements intelligents peuvent émerger des associations entre agents réactifs.

- Pas de représentation explicite.
- Organisation implicite/induite.
- Communication via l'environnement.

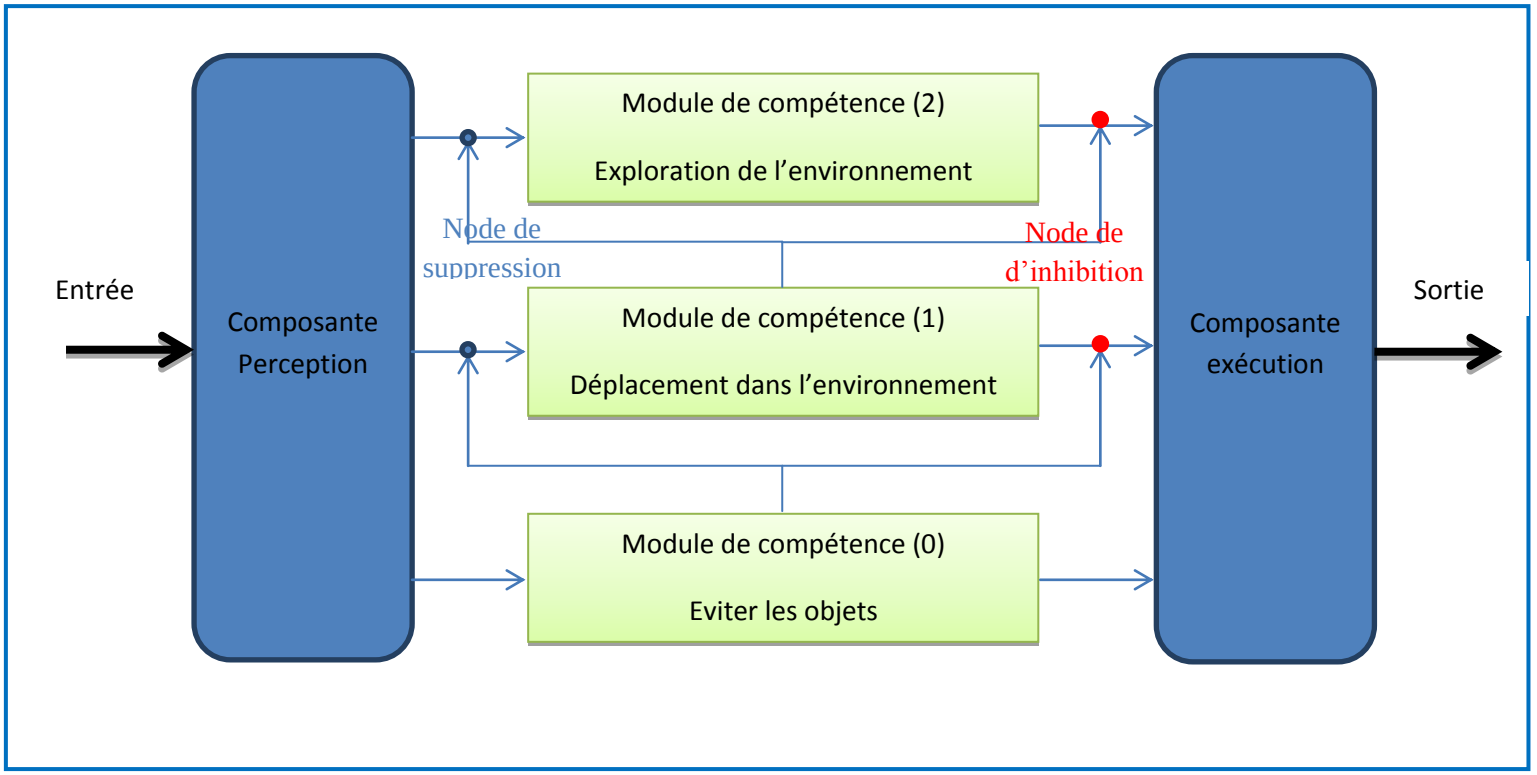

Figure 1.1 : Architecture réactive de subsomption

Cette architecture a été proposée par Rodney Brooks qui s'appelle architecture de subsomption. Cette architecture comporte plusieurs modules :

- M0 : Un module qui a la compétence d'éviter les obstacles.
- M1 : Un module qui est responsable des déplacements dans l'environnement tout en évitant les obstacles à l'aide de M0.
- M2 : un module qui a la compétence supérieure, la plus abstraite, de faire l'exploration systématique de l'environnement en se déplaçant grâce aux actions du module M1. [1]

Chaque module correspond à un comportement spécifique pour accomplir une tache particulière. Les modules sont structurés en couche hiérarchisée. Un module sur une couche inferieurs a une priorité plus grande qu'un module situé sur une couche plus élevé parce qu'il est responsable plus urgente.

b) Agent cognitif :

Est spécialisé dans un domaine et sait communiquer avec les autres. Ils possèdent des buts et des plans explicites leur permettant d'accomplir leurs buts. (Analogie avec les groupes en sociologie).

- Représentation explicite de soi, environnement et les autres agents.
- Organisation explicite.
- Interaction explicite et élaborée.

Architecture BDI

Cette architecture est conçue en partant sur le modèle (Croyance-Désir-intention) (Belief-Desire-Intention).

- Belief : Ce sont les informations que l'agent possède sur l'environnement et sur d'autres agents qui existent dans le même environnement.
- Desire : Représente les états que l'agent aimerait voir réaliser. Ces désires peuvent être contradictoire.
- Intention : Les intentions sont les actions qu'un agent a décidées de faire pour accomplir ses désirs.

L'agent BDI doit produire des plans, notamment une séquence d'action pour résoudre le problème. Il y a cinq fonctions qui sont exécuter durant le cycle de l'agent BDI (On note B : Believ, D : Desir, I : intetion, P : perception) :

- **B = revc (B,p)** : La composante révision de croyance exécute cette fonction lors de la reception de nouvelles perceptions.

- **I = option (D, I)** : Le procès de décision exécute cette fonction en prenant en compte les désirs et les intentions courantes de l'agent.

- **D= des (B, D, I)** : est la fonction qui peut changer les désirs d'un agent si ses croyances ou intentions changent, pour maintenir la consistance des désirs de l'agent, cette fonction est également réalisée par la composante Processus de décision.

- **I=filtre (B, D, I)** : est la fonction la plus importante car elle décide des intentions à poursuivre ; elle est réalisée par la composante Filtre.

- **PE=plan (B,I)** : c'est la fonction qui transforme les plans partiels en plans exécutables, PE étant l'ensemble de ces plans ; elle peut utiliser, par exemple, une bibliothèque de plans, représentée par le module LibP dans la figure1.2.

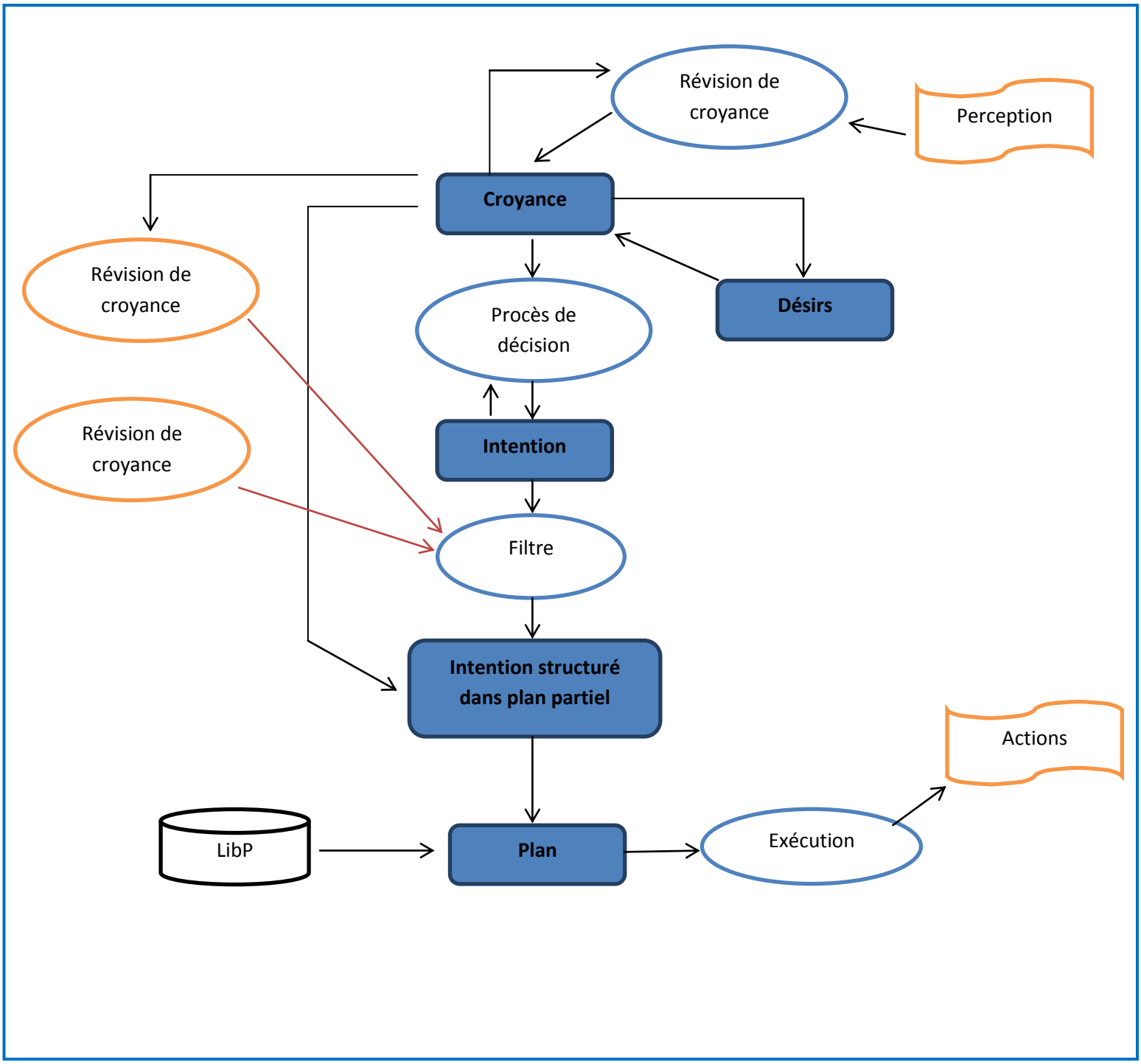

Figure 1.2 : Architecture agent BDI

c - ***Agent Hybride***

Un agent Hybride est un agent qui est à la fois cognitif et réactif au même temps. Cet agent est composé d'un ensemble de modules organisés hiérarchiquement, ces modules peuvent être soit une composante cognitive soit une composante réactive.

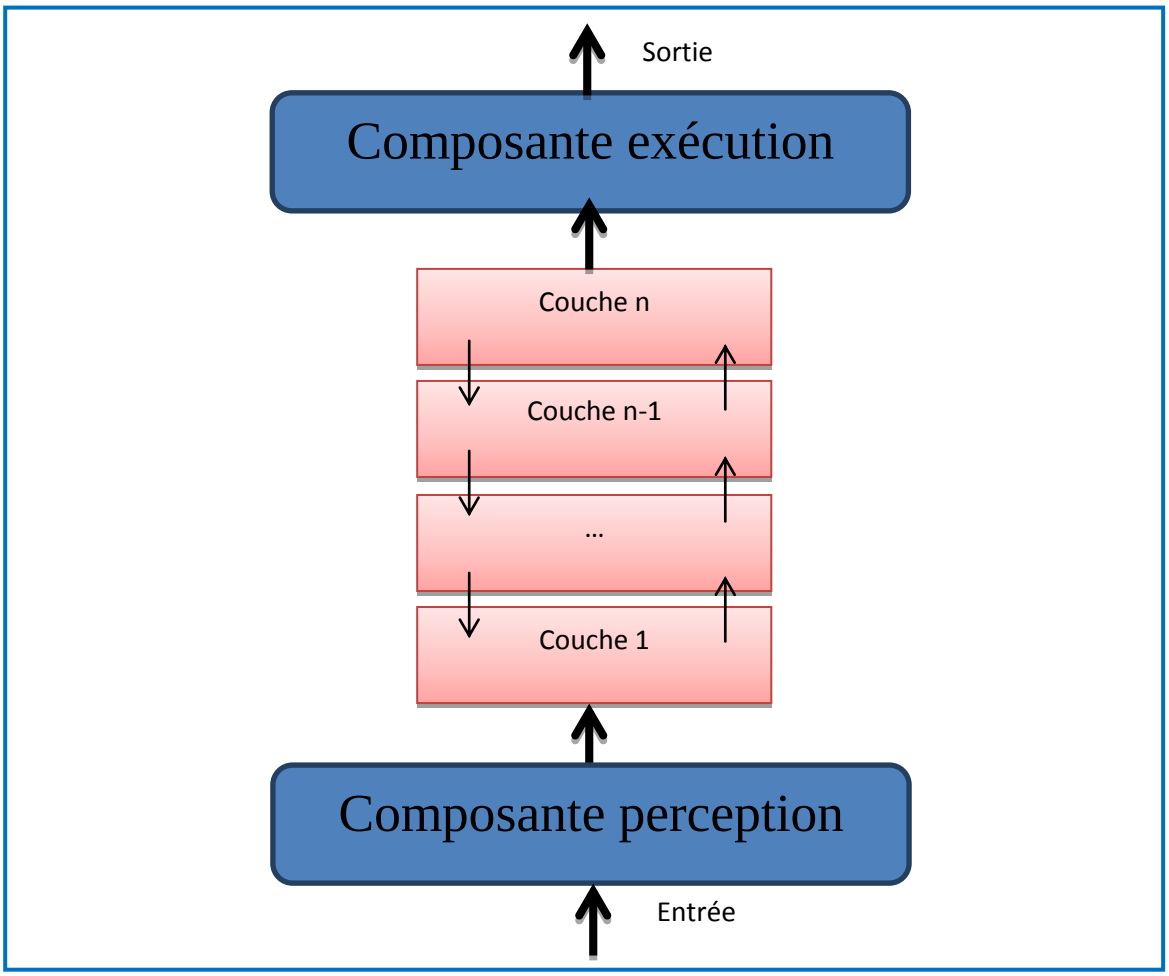

Figure 1.3 : Architecture Agent Hybride (modèle vertical)

Ce qu'il faut noter dans cette architecture verticale c'est que les données entrantes depuis la composante de perception doivent passer par toutes les couches avant d'arriver à la composante d'exécution, donc l'agent effectue à la fois un raisonnement réactif et cognitif.

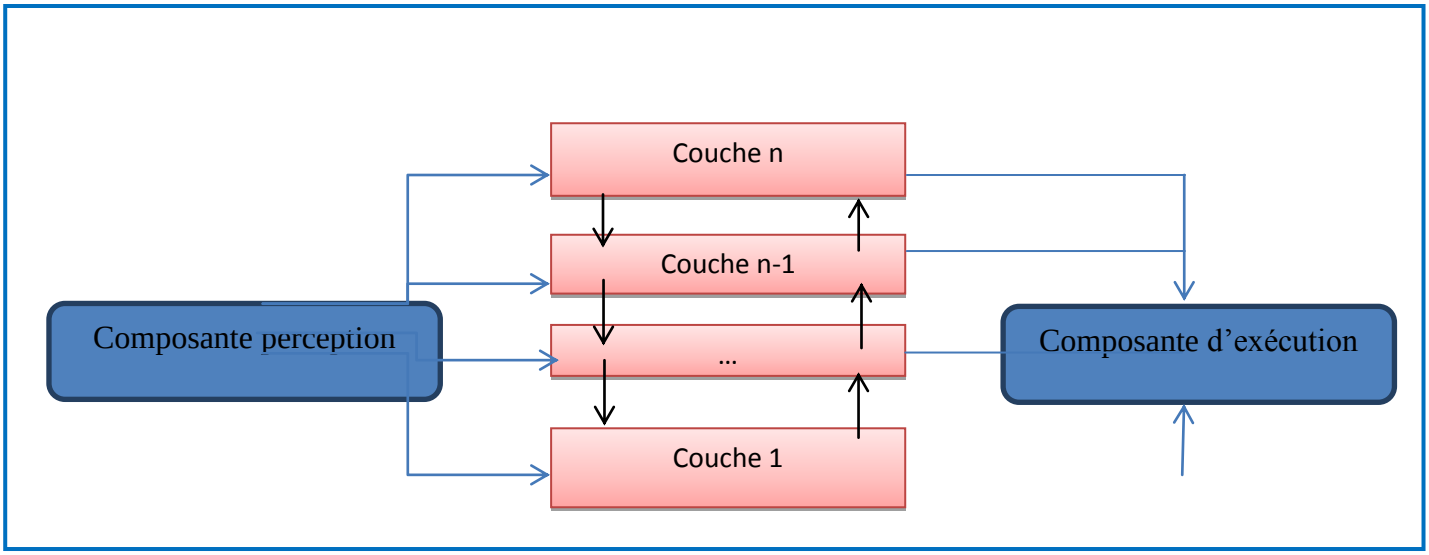

Figure 1.4 : Architecture agent hybride (modèle horizontal)

Dans ce modèle et contrairement au modèle verticale, les données de la perception passent seulement par une seule couche avant d'arriver à la composante d'exécution, et dans ce cas l'agent hybride effectue soit un raisonnement réactif soit un raisonnement délibératif.

3 Système multi-agent (SMA) :

L'approche multi-agents considère que le concept d'agent ne prend tout son intérêt qu'au sein d'un univers d'agents dans lequel les agents se communiquent pour résoudre un problème complexe difficile à résoudre par un seul agent. On parle donc des systèmes multi-agents (SMA).
Dans un tel système, l'agent a un comportement à la fois individuel et collectif. Le caractère individuel d'un agent se manifeste à travers des objectifs qui lui sont propres et qu'il essaye d'atteindre avec les ressources et les compétences qu'il dispose. L'agent est donc une entité intentionnelle [Ferber 95]. Toutefois, un SMA ne se résume pas à un ensemble d'agents agissants de façon individuelle. L'intérêt des SMA provient des interactions entre ses agents. Le caractère collectif des agents se traduit par leurs facultés à gérer leurs interactions : coopération, négociation et coordination.

3.1 Définition :

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

1. Un environnement E, c'est-à-dire un espace disposant généralement d'une métrique. et modifiés par les agents.
2. Un ensemble d'objets O situé dans E. Ces objets peuvent être perçus, créés, détruits
3. Un ensemble A d'agents, représentent les entités actives du système.
4. Un ensemble de relations R qui relient des agents entre eux.
5. Un ensemble d'opérations Op permettant aux agents de percevoir, produire, consommer, transformer et manipuler des objets.
6. Des opérateurs chargés de représenter l'application de ces opérations et la réaction de l'environnement envers les tentatives de modification.

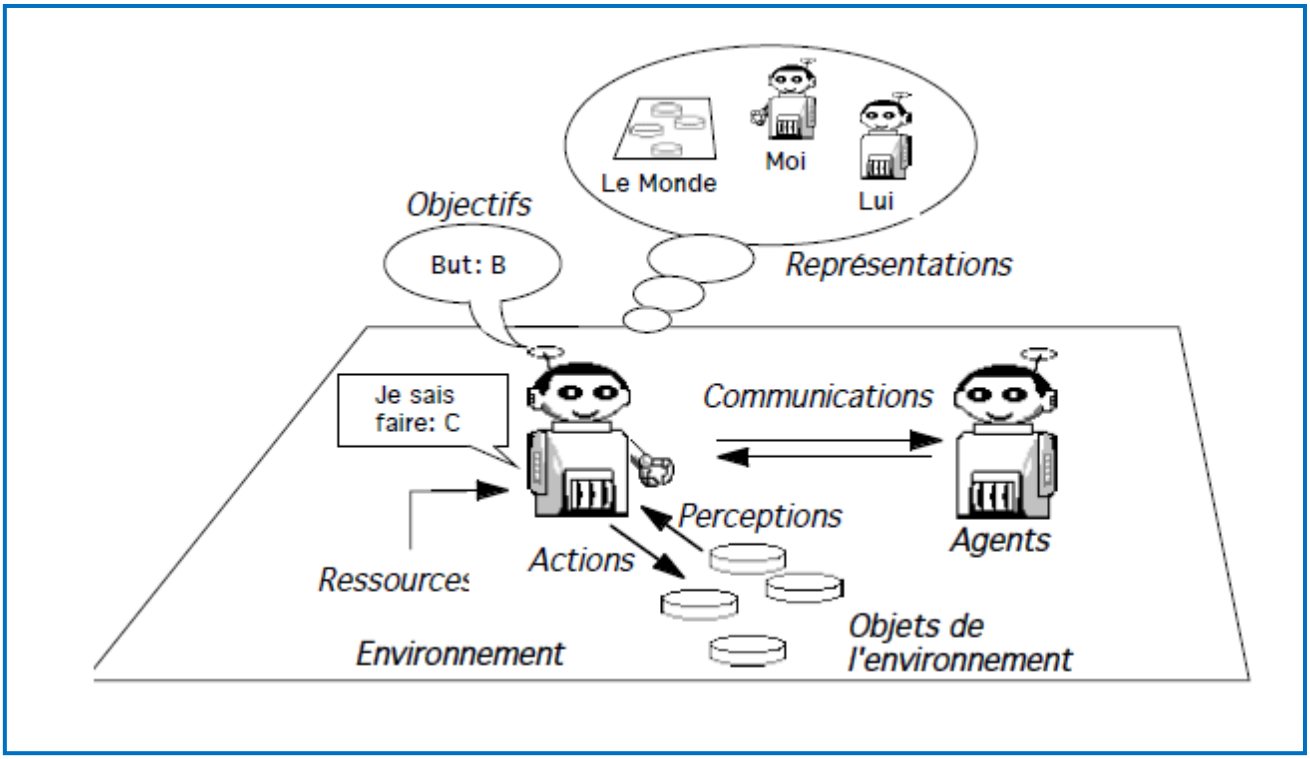

Figure 1.5 : Système multi agent

3.2 Domaines d'application:

Les systèmes multi-agents ont déjà montré des promesses particulières dans une variété de systèmes technico-sociaux comme : la gestion du trafic aérien, la gestion des ressources, les applications boursières, la télémédecine, l'environnement de robotique cognitive ou les systèmes de commande et de contrôle en temps réel, etc.

Aussi, la technologie agent a trouvé sa place dans les systèmes dynamiques, où l'action autonome, la flexibilité et l'intelligence sont requises quand les quantités de données à traiter dépassent l'envergure de traitement sur une seule machine. Cette approche constitue un bon moyen de résolution et d'appréhension des problèmes. Cela, explique l'utilisation actuelle des systèmes multi-agents dans divers domaines, dont nous avons cité quelques-uns brièvement :

- le web sémantique et la gestion documentaire sur le web ;
- les réseaux tels que Peer to Peer (P2P) ;
- les systèmes distribués concurrents et les bases de données reparties ;
- le commerce électronique sur le net et la gestion des processus d'affaires ;
- les systèmes d'information coopératifs ;
- les télécommunications ;

3.3 Les avantages des SMA:

Les systèmes multi-agents offrent beaucoup d'avantages dans le développement des systèmes complexes. En fait, le paradigme agent est considéré comme le paradigme idéal pour le développement de ces systèmes. La puissance des systèmes multi-agents est une conséquence de l'interaction des différents domaines (l'intelligence artificielle, le génie logiciel et les systèmes distribués). Parmi les avantages des systèmes multi-agents, on peut citer :

- La modularité
- La réutilisabilité
- La facilité de maintenance
- La fiabilité
- L'efficacité
- L'adaptation à la réalité
- Les modes d'interaction sophistiqués

3.4 Les défis des systèmes multi-agents:

Malgré les avantages des systèmes multi-agents, ce paradigme confronte plusieurs défis. En fait, ces défis sont des résultats de caractéristiques d'agent et de systèmes multi-agents. Prenant l'exemple de l'autonomie comme une caractéristique intrinsèque des agents, cette caractéristique fait du comportement de l'agent imprédictible. En conséquence, la validation des systèmes développés représente un problème réel. Ainsi, les défis des systèmes multi-agents résident dans :

- La formulation des problèmes, l'allocation des ressources aux différentes entités et la synthèse des résultats.
- L'interaction entre les agents (quand et comment) et le raisonnement sur les autres agents durant le processus d'interaction.
- L'assurance de cohérence des comportements en assurant un compromis entre les actions locales et le traitement distribué et en éliminant les effets indésirables (comme les comportements chaotiques).
- L'ingénierie des systèmes multi-agents en développant des méthodes, des méthodologies, des techniques et des outils facilitant le développement de ces systèmes.

3.5 Interaction entre agents:

Les interactions sont au centre de la problématique des SMA, car elles permettent aux agents de produire ensemble le comportement attendu du système. Une interaction entre agents est une mise en relation dynamique de deux ou plusieurs agents. Dans [Ferber 95], Ferber classe les types d'interactions entre agents dans un SMA selon la compatibilité de leurs buts, la disponibilité des ressources et leurs aptitudes propres. On aura donc les types d'interaction suivants :

- **indifférence des agents :** le cas d'indépendance entre agents ;
- **Coopération des agents :** situation de collaboration (simple ou coordonnée) ou d'encombrement ;
- **Antagonisme entre les agents :** situation de compétition (individuelle ou collective) ou de conflits (individuels ou collectifs) pour des ressources.

La figure 1.6 représente une classification des différentes méthodes d'interactions selon le type et la situation [2]. Cette classification n'est pas la seule possible, elle peut varier selon que l'on se place au niveau des méthodes ou au niveau de situations.

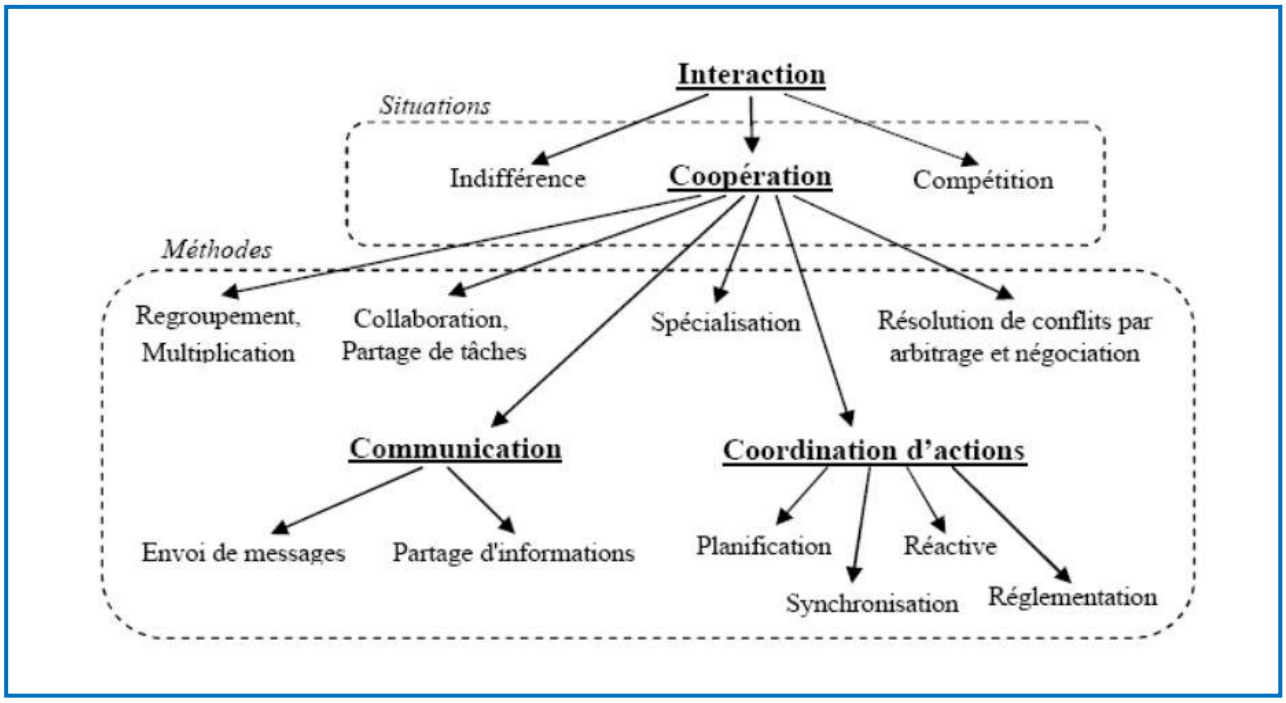

Figure 1.6 : Situations et méthodes d'interactions dans un SMA.

3.6 Communication entre agents:

3.6.1 Définition:

Pour coordonner l'activité d'un ensemble hétérogène d'agents autonomes, il faut que les agents communiquent dans un langage compréhensible par les autres. On observe que dans un système ouvert un tel langage peut constituer une interface entre les agents. L'utilisation d'un langage commun implique que tous les agents comprennent son vocabulaire sous tous ses aspects concernant :

- **La syntaxe :** qui précise le mode de structuration des symboles ;
- **La pragmatique :** qui permet d'interpréter les symboles ;
- **L'ontologie :** qui permet d'utiliser les mêmes mots d'un vocabulaire commun.

3.6.2 Les modes de communications:

Les communications, dans les systèmes multi-agents, sont à la base des interactions et de l'organisation. Si les agents peuvent coopérer, coordonner leurs actions, réaliser des tâches en commun, c'est parce qu'ils communiquent. Koning et Pesty [Kon01] distinguent deux modes de communication : la communication directe par passage de message et la communication indirecte via l'environnement ou par partage de ressources.

- **Communication par envoi de messages :**

Les échanges entre agents se font par envoi de messages selon un protocole bien défini. Deux types de transmissions sont possibles : la transmission directe (point à point) et la transmission par diffusion (**broadcasting**). Dans le premier, l'émetteur du message connaît et précise l'adresse de (ou des) destinataires. Dans le second, le message est envoyé à tous les agents du système. Les deux modes sont souvent combinés (**multicasting**) : le premier permet d'établir une liste de correspondants privilégiés, utilisés ensuite par le second.

- **Communication par partage d'information :**

Les agents ne sont pas en liaison directe mais communiquent via une structure de donnés partagée. Cette structure contient les connaissances relatives à la résolution (état courant du problème) qui évoluent durant le processus d'exécution. L'architecture de Blackboard (tableau noir) constitue un exemple parfait de l'utilisation de ce mode de communication. Dans un système à Blackbord, on trouve les trois éléments suivants :

- Un ensemble d'agents indépendants (sources de connaissances) ;
- Une structure de données partagées (Blackbord), divisée souvent en plusieurs niveaux d'abstractions ;
- Un mécanisme de contrôle assurant l'intervention des différentes sources de connaissances pour la résolution du problème.

3.7 Les langages de communications dans les SMA:

Les agents doivent avoir des capacités à manipuler un langage commun. Parmi ces langages on trouve :

3.7.1 Le langage KQML:

KQML, proposé en 1993 par le consortium DARPA-KSE (Knowledge Sharing Effort), est un langage permettant de structurer des messages afin de partager de l'information entre agents [3] KQML est basé sur la théorie des actes de langage. Il propose une encapsulation des messages dans une performative qui définit l'acte illocutoire. Il est constitué d'un nombre important de performatifs qui sont en quelque sorte les opérations permises entre les agents. Les 36 performatifs de KQML peuvent être classés en trois grandes catégories :

- Les 18 performatives de discours : servent à échanger des connaissances et des informations contenues dans la base de connaissance de l'agent. Exemples: demander-if, ask-one, tell, describe, stream-all, etc.
- Les 11 performatifs d'interconnexion : aide à la mise en relation des agents entre eux. Exemples : s'inscrire, se désinscrire, diffuser, etc.
- Les 7 performatives d'exception : servent à changer le déroulement normal des échanges. Exemples : erreur, désolé, veille, etc.

3.7.2 Le langage ACL (Agent Communication Langage) :

Le langage ACL de FIPA (Foundation for Intelligent Physical Agents) est fondé également sur la théorie des actes de langage.
Les performatifs de ce langage peuvent être regroupés en cinq groupes selon leurs fonctionnalités [3] :

- Actes pour le passage d'information : Inform, Inform-if, Inform-ref,Confirmer, Interdire.
- Actes pour la réquisition d'information : Query-if, Query-ref, Subscribe.
- Actes pour la négociation : Accept-proposal, Cfp, Propose, Reject-proposition.
- Actes pour la distribution des tâches : Request, Request-when, Request-chaque fois, d'accord, annuler, refuser.
- Actes pour le traitement des erreurs : Echec, Incompris.
- Un message ACL dispose obligatoirement des champs suivants (Tableau 1.1) :

Champs	Description
Performative	type de l'acte de communication
Sender	expéditeur du message
Receiver	destinataire du message
reply-to	participant de la communication
content	contenu du message
language	description du contenu
encoding	description du contenu
ontology	description du contenu
protocol	contrôle de la communication
conversation-id	contrôle de la communication
reply-with	contrôle de la communication
in-reply-to	contrôle de la communication
reply-by	contrôle de la communication

Tableau 1.1 : Contenu d'un message ACL

3.7.3 Les plateformes de développement d'un SMA:

Il existe actuellement de nombreuses plates-formes multi-agents qu'on peut classer en plusieurs catégories.

- Les plates-formes pour agents mobiles (**Voyager, Odissey, Aglet, ...etc**.) qui fournissent la mobilité à des agents
- Les plates-formes pour agents cognitifs (AgentBuilder, etc.) dans lesquels on trouve les plates- formes FIPA (**Jade, FIPA-OS, etc**.) ou KQML (**JAT, JAT-Lite, ...etc**.),
- Les plates-formes pour agents collaboratifs (Zeus, JAFMAS, KAoS, JAFIMA, ...etc.),
- Les plates-formes pour la simulation multi-agents (Cormas, Swarm, ...etc.).

Toutes ces plates-formes sont dédiées soit à un certain type d'agents (agents à base de règles pour AgentBuilder ou agents collaboratifs pour Zeus par exemple) soit à un certain type d'applications (simulation, mobilité). Il n'existe pas à notre connaissance de plate-forme générique qui permette de réaliser des applications dans tous ces domaines.

4. Conclusion :

Dans ce chapitre, nous avons représenté les concepts agent et le système multi-agents, nous avons défini leurs caractéristiques, et les notions fondamentales de ce domaine. Il est intéressant de noter, que l'ensemble des concepts de l'approche multi-agents font de celle-ci une approche très riche et très puissante en termes de modélisation.

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. 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. Dans ce chapitre, nous aborderons c'est quoi un test logiciel, pourquoi on fait un test, c'est quoi un test basé model et nous décrirons quelques travaux sur le test des agents.

2. Définition du test :

En informatique, un **test** désigne une procédure de vérification partielle d'un système. Son objectif principal est d'identifier un nombre maximal de comportements problématiques du logiciel. Il permet ainsi, dès lors que les problèmes identifiés seront corrigés, d'en augmenter la qualité.

D'une manière plus générale, le test désigne toutes les activités qui consistent à rechercher des informations quant à la qualité du système afin de permettre la prise de décisions.

Un test ressemble à une expérience scientifique. Il examine une hypothèse exprimée en fonction de trois éléments : les données en entrée, l'objet à tester et les observations attendues. Cet examen est effectué sous conditions contrôlées pour pouvoir tirer des conclusions et, dans l'idéal, être reproduit.

3 .Terminologie liée au test :

Pour cerner la notion de test, il convient de comprendre des notions que nous présentons ci-après :

✓ **Spécification :** En informatique, la spécification est un modèle d'un logiciel. C'est aussi l'étape en génie logiciel qui consiste à décrire ce que le logiciel doit faire. On distingue : Les spécifications informelles, par exemple un texte en français, Les spécifications semi-formelles avec une syntaxe plus précise ou comportant des diagrammes plus ou moins standardisés et Les spécifications formelles qui présentent une syntaxe et une sémantique.

✓ **Satisfaction :** Un programme satisfait sa spécification lorsqu'il est en tout point conforme aux exigences de celle-ci.

✓ **La vérification** est « le processus d'évaluation d'un produit issu d'une des activités du développement logiciel, pour déterminer la correction et la cohérence avec les produits ou les normes fournies comme entrée de cette activité ».

La vérification sert à répondre à la question : Est-ce que le logiciel fonctionne correctement ?
Le test est un cas particulier de vérification.

✓ **La validation** est « le processus d'évaluation d'un logiciel pour déterminer sa conformité avec les besoins spécifiés » La validation nous tentons de répondre à la question : est-ce le bon système, fonctionne selon les attentes de l'utilisateur ?

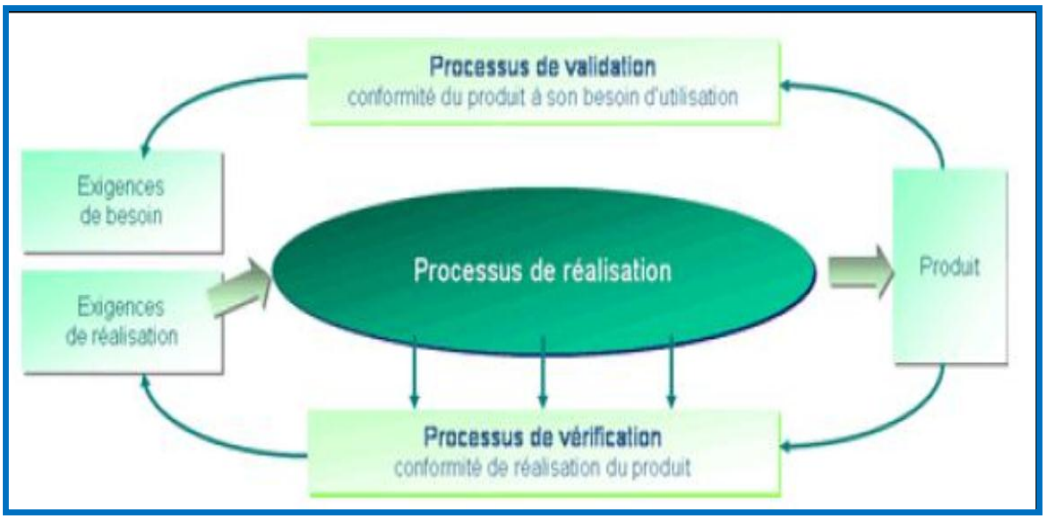

Figure 2.1 : Concepts de vérification et validation

✓ **Une erreur** (en anglais : error) est une différence entre une valeur ou une condition calculée, observée ou mesurée et la valeur ou condition spécifiée, qui est vraie ou théoriquement correcte

✓ **Un défaut** (en anglais: Bug, defect) une imperfection dans un composant ou un système qui peut conduire à ce qu'un composant ou un système n'exécute pas les

fonctions requises, par exemple une définition de données incorrecte. Un défaut peut causer la défaillance d'un composant ou d'un système

- ✓ **Une panne** ou défaillance (en anglais: failure) est l'incapacité d'un système ou d'un de ses composants d'effectuer les fonctions demandées dans les conditions de performance spécifiées

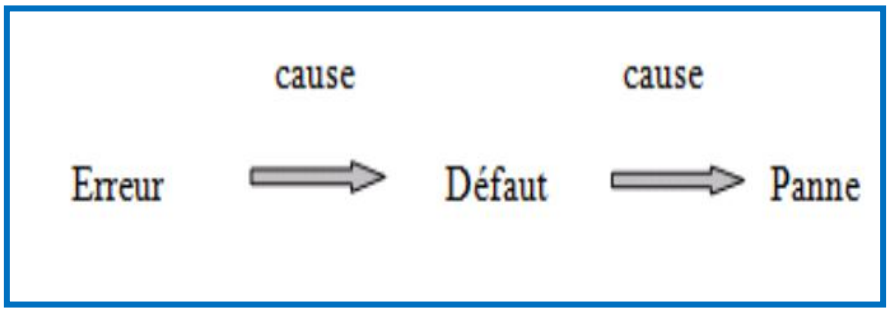

Figure 2.2 : Relation entre défaillance / faute / erreur

- ✓ **Cas de test:** C'est un ensemble de valeurs d'entrée, de préconditions d'exécution, de résultats attendus et de post-conditions d'exécution, développés pour un objectif particulier, tel qu'exécuter un chemin particulier d'un programme ou vérifier le respect d'une exigence spécifique

  Cas de test : Chemin fonctionnel à mettre en œuvre pour atteindre un objectif de test. Un cas de test se définit par le jeu de test à mettre en œuvre, le scénario de test à exécuter et les résultats attendus

- ✓ **Jeu de test** : Données en entrée d'un cas de test : valeurs à saisir, données réelles (base de données existante ou de test), génération automatique (aléatoire ou à partir de spécifications). Le même jeu d'essai peut servir à plusieurs cas de test

4. Pourquoi le test des logiciels :

L'importance du test dans le processus de développement d'un logiciel n'est plus à démontrer. D'une part, en raison de son importance économique croissante (60 % du coût total du développement) et d'autre part, parce que le test constitue une tâche essentielle dans l'élaboration de la qualité d'un logiciel. Considéré longuement comme une activité de «second rang» dans le processus de développement d'un logiciel, le test connaît actuellement une véritable révolution fondée sur une industrialisation de ses processus, une professionnalisation des métiers du test, l'arrivée à maturité d'une chaîne outillée allant des exigences au référentiel de tests, et enfin la mise en place de centres de service, internes ou externes, dédiés aux activités de test. Son objectif est de mettre en œuvre le logiciel en utilisant des données similaires aux données réelles, pour observer les résultats, détecter les anomalies et en déduire l'existence d'erreurs.

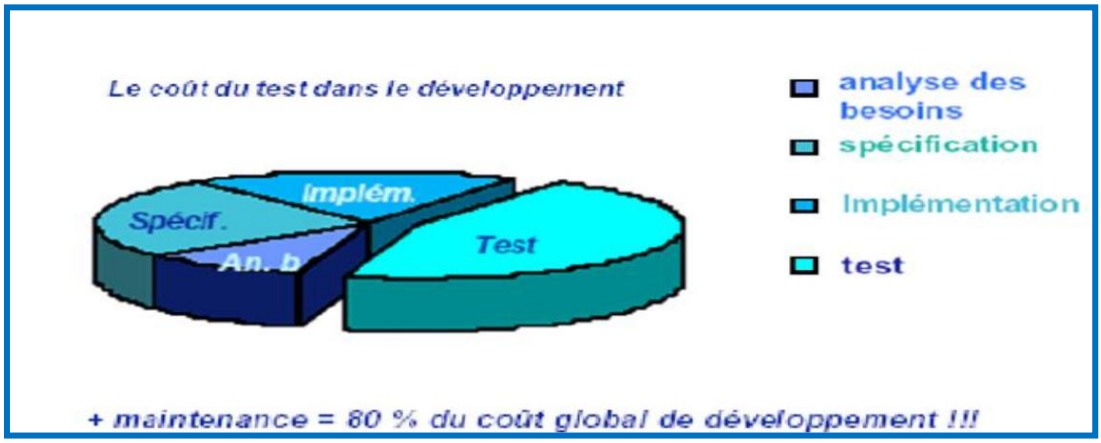

Figure 2.3 : Coût du test

5. 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 2.4 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.

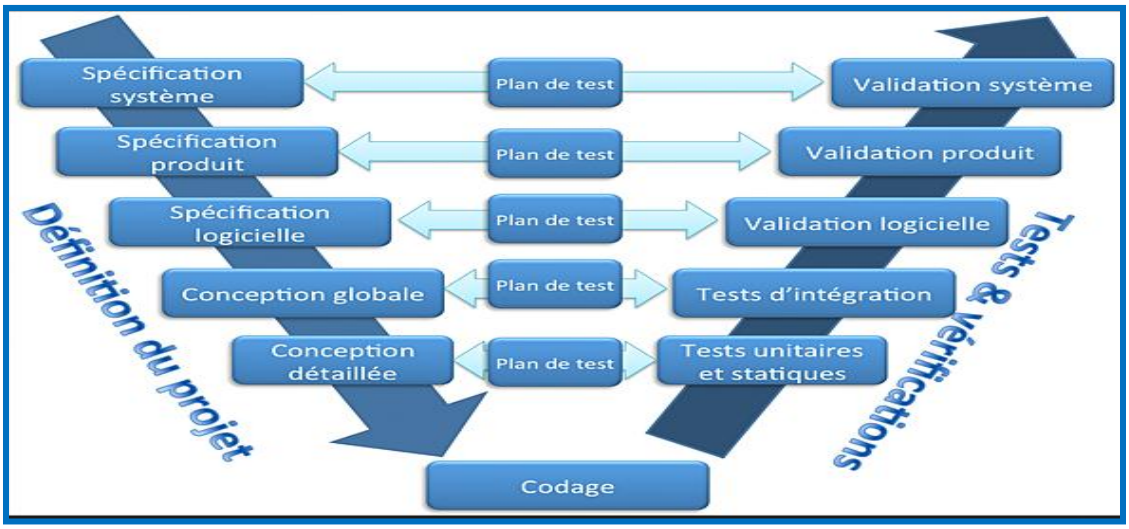

Figure 2.4 : Les niveaux de test

5.1. Test unitaire :

Est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme.

Il s'agit pour le programmeur de tester un module, indépendamment du reste du programme, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles et qu'il fonctionne correctement en toutes circonstances. Cette vérification est considérée comme essentielle, en particulier dans les applications critiques. Elle s'accompagne couramment d'une vérification de la couverture de code, qui consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions présentes dans le code à tester. L'ensemble des tests unitaires doit être rejoué après une modification du code afin de vérifier qu'il n'y a pas de régressions (l'apparition de nouveaux dysfonctionnements).

Dans les applications non critiques, l'écriture des tests unitaires a longtemps été considérée comme une tâche secondaire. Cependant, la méthode Extrême programming (XP) a remis les tests unitaires, qu'elle nomme maintenant Tests du Programmeur, au centre de l'activité de programmation.

La méthode XP préconise d'écrire les tests en même temps, ou même avant la fonction à tester (TDD). Ceci permet de définir précisément l'interface du module à développer. En cas de découverte d'un bogue logiciel, on écrit la procédure de test qui reproduit le bogue. Après correction on relance le test, qui ne doit indiquer aucune erreur. **[4]**

5.2. Test d'intégration :

Est une phase de tests, précédée par les tests unitaires et généralement suivie par les tests de validation, vérifiant le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module ») ; dans le test d'intégration, chacun des modules indépendants du logiciel est assemblé et testé dans l'ensemble.. **[5]**

L'objectif de chaque phase de test est de détecter les erreurs qui n'ont pas pu être détectées lors de la précédente phase.

Pour cela, le test d'intégration a pour cible de détecter les erreurs non détectables par le test unitaire. Le test d'intégration permet également de vérifier l'aspect fonctionnel, les performances et la fiabilité du logiciel. L'intégration fait appel en général à un système de gestion de versions, et éventuellement à des programmes d'installation. Cela permet d'établir une nouvelle version, fondée soit sur une version de maintenance, soit sur une version de développement.

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.

5.3. Test de validation :

Le test de validation 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.

5.4. Test de non-régression :

Le test de non-régression consiste à ré-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 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.

6. Les méthodes de test :

Il existe plusieurs techniques et stratégies de contrôle couvrant la majorité des phases du cycle de développement du logiciel. Par exemple :

- Test basé sur les modèles
- Le test structurel statique
- Le test structurel dynamique (boîte blanche)
- Le test fonctionnel (boîte noire) …
- Test de mutation
- Test aléatoire

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.

6.1. 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)". 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 2.5) 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.

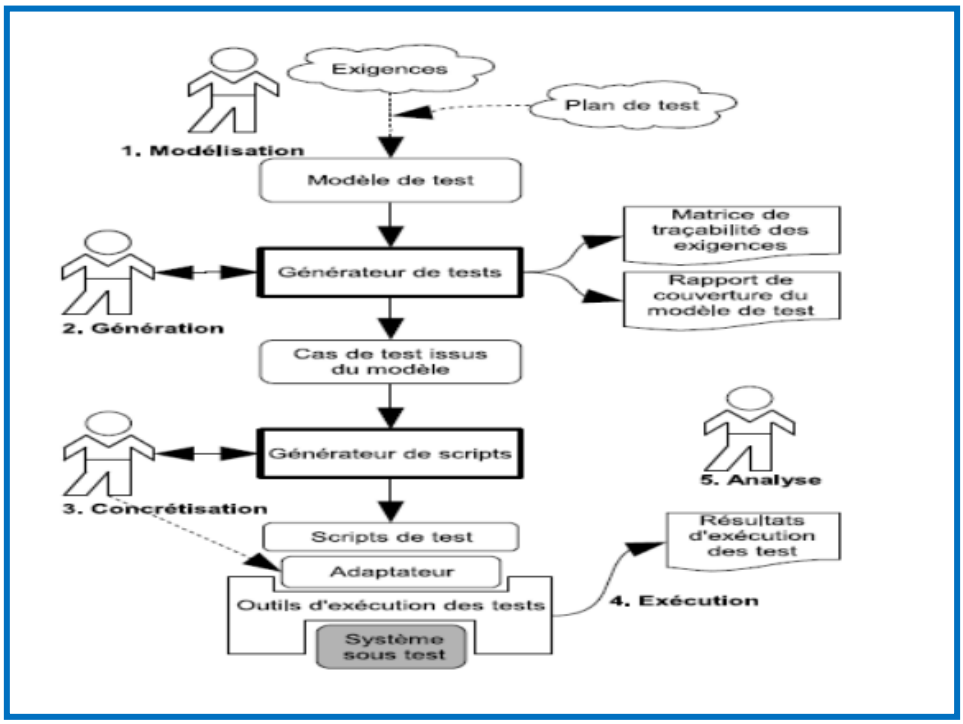

Figure 2.5 : Le processus de Model-based Testing

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.

6.1.1 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, 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.

6.1.2 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 2.6 donne une classification liée à la génération des cas de test selon les deux dimensions.

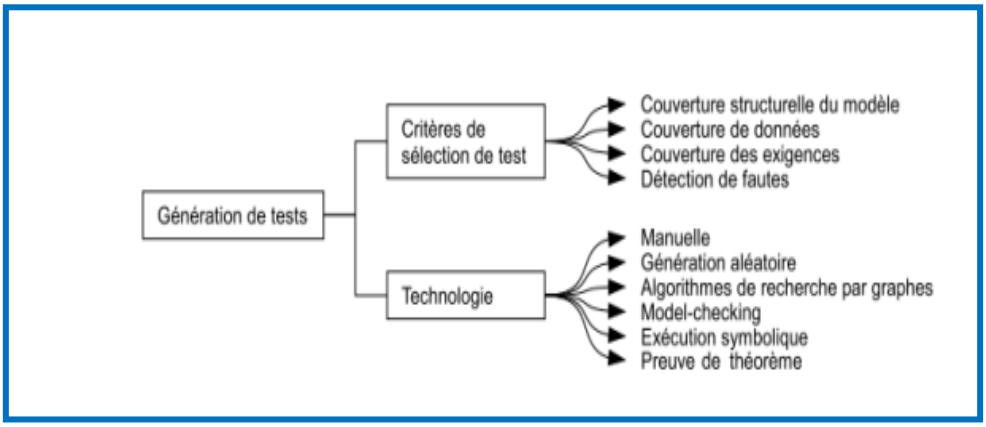

Figure 2.6 : Classification des problématiques liée à la génération de cas de test

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.

- 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 .
- 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
- 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).

- 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.

- 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)

- 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 satisfiabilité 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 : **[6]**. Utilise l'assistant de preuve de théorème Isabelle/HOL **[7]**

- 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 : **[8]** propose une génération automatique de données de test dédiées au test structurel **[9]**. Utilise cette logique pour le test de systèmes réactifs. **[10]**

Utilise un solveur de contraintes ensemblistes pour calculer des préambules de test à partir d'une machine B.

6.1.3 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.

7. 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.

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.

- Une grande quantité de données manipulées par ces agents chacun ayant ses propres objectifs et ses propres buts.

- 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.

- Ils sont également non déterministes, puisqu'il n'est pas possible de déterminer a priori toutes les interactions d'un agent pendant son exécution.

- Les agents sont autonomes et coopèrent avec d'autres agents, ainsi ils peuvent fonctionner correctement seuls mais inexactement dans une communauté.

- 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. 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.

7.1. Les niveaux de test agent :

Le test des SMA se compose de cinq niveaux : 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. 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

- **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 les buts des dépositaires avec la participation de ces derniers.

7.2. 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. 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

- 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. **[11]**

- Le travail **d'Ekinci et al.** 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

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 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. **[12]**

c. Niveau d'intégration

- 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. **[13]**

d. Multi niveaux de test

Dans multi niveaux de test on trouve le travail de **Nguyen et al**, 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.

Le tableau (Tableau 2.1) suivant montre une classification, par niveau, des travaux qui existent sur le test des agents.

Travail / **Critère**	**Test unitaire**	**Test agent**	**Test système**
Zhang, Z., Thangarajah, J., Padgham, L	☑		
Zhang, Z., Thangarajah, J., Padgham, L	☑		
Zhang, Z., Thangarajah, J., Padgham, L	☑		
Ekinci, E.E., Tiryaki, A.M., Cetin, O., Dikenelli, O	☑		
Nguyen, C.D., Perini, A., Tonella, P	☑		
Savarimuthu S., Winikoff M	☑		
Coelho, R., Kulesza, U., vonStaa, A., Lucena, C		☑	
Lam, D.N., Barber, K.S		☑	
Nguyen, C.D., Miles, S., Perini, A., Tonella, P		☑	
Nguyen, C.D., Perini, A., Tonella, P		☑	
Nunez, M., Rodriguez, I		☑	
Huang Z., Alexander R., Clark J		☑	
De Wolf, T., Samaey, G., Holvoet, T			☑
Dehimi, NEH., Mokhati, F., Badri, M.		☑	☑
Dehimi, NEH, Mokhati, F.			☑

Tableau 2.1 : Résumé sur les travaux qui existent sur le test agent.

8. Conclusion

Le test des systèmes Multi-Agent (SMA) a besoin de techniques appropriées pour évaluer les comportements autonomes de l'agent aussi bien que les propriétés de distribution, sociales et délibératives, qui sont particulières à ces systèmes. Parmi ces techniques, nous trouvons le test basé sur les modèles, il est basé sur un modèle de système afin de produire des cas de test abstraits. Pour que ces derniers puissent être soumis au système sous test, les cas de test abstraits doivent être transformés en des cas de test concrets. Ce dernier est proposé et appliquée dans ce mémoire en utilisant l'application de vente de livres comme cas d'étude.

Chapitre3 : L'approche proposée et Etude de cas

1 Introduction

Dans ce chapitre nous présentons l'approche proposée qui consiste à générer des cas de test basé sur le diagramme de séquence AUML d'un système multi agents, ainsi que ses étapes en l'appliquant sur une étude de cas concrète : ***Une application multi-agents pour les ventes des livres.***

2 L'approche proposée

Notre approche consiste à appliquer le test basé model (model based testing) (MBT). Cette technique 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.

L'approche proposée est constituée des étapes suivantes (Figure 3.1) :

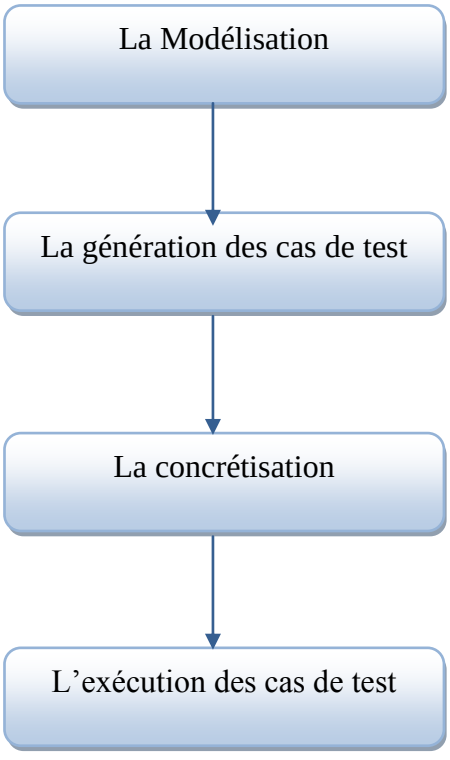

Figure 3.1 : Les étapes de l'approche proposée.

3 La modélisation

La modélisation est le processus de produire un modèle .Un modèle est une abstraction d'un système construite dans un but précis. On dit alors qu'un modèle est une représentation du système. Un modèle est une abstraction dans la mesure où il contient un ensemble restreint d'informations sur un système. Il est construit dans un but précis et les informations qu'il contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui sera faite du modèle. Pour modéliser le système on utilise les méthodes **formelles** ou **méthodes semi formelles.**

- **Les méthodes formelles** :

Sont des techniques permettant de raisonner rigoureusement, à l'aide de logique mathématique, sur un programme informatique ou du matériel électronique numérique, afin de démontrer leur validité par rapport à une certaine spécification. Elles reposent sur les sémantiques des programmes, c'est-à-dire sur des descriptions mathématiques formelles du sens d'un programme donné par son code source (ou, parfois, son code objet). **[14]**

Ces méthodes permettent d'obtenir une très forte assurance de l'absence de bug dans les logiciels (*Evaluation Assurance Level*, *Safety Integrity Level*). Elles sont utilisées dans le développement des logiciels les plus critiques. Leur amélioration et l'élargissement de leurs champs d'application pratique sont la motivation de nombreuses recherches scientifiques en informatique. Exemple des méthodes formelles : Réseau de Pétri, Maude ….

- **Les méthodes semi-formelles** :

Un langage de degré semi-formel dans le sens d'Uschold et Gruninger (1996) est un formalisme d'expression de la connaissance dont la souplesse grammaticale et sémantique rend son utilisation conviviale. Ce type de formalisme, qui fait généralement appel au format graphique, est parfois employé pendant l'étape de conception d'un système (Rumbaugh et al., 1999) ou encore pour favoriser le transfert d'expertise dans les organisations (Basque et al., 2004) ou l'apprentissage dans des situations éducatives (Basque et Pudelko, 2010b ; Kinshuk et al., 2008 ; Paquette, 2002) ainsi que pour susciter les échanges pendant une séance de remue-méninges, ou plus simplement pour représenter graphiquement un énoncé. Bien qu'un modèle semi-formel contienne toujours des éléments d'ambigüité, sa souplesse d'expression, surtout lorsqu'il fait appel à un format graphique, permet d'accéder plus facilement à l'identification des connaissances tacites des experts. Dans un tel cadre, la spontanéité n'est pas bloquée par une charge cognitive trop lourde associée à une formalisation poussée de la pensée (Basque et al. 2008). Ainsi, le langage semi-formel graphique fournit un certain guidage représentationnel (Suthers, 2003) qui structure le processus de modélisation et qui peut servir de première itération d'élicitation pour la conception d'une ontologie formelle (Héon, 2010). Exemple des méthodes semi-formelles : UML, AUML, les diagrammes de flux de données … **[15]**

Dans notre cas d'étude on va construit un modèle abstrait dédié au test d'un système de vente de livres en utilisant le diagramme de séquences (AUML) et le graphe de flux de donnés.

3.1 Les diagrammes AUML :

AUML est une extension de la notation UML pour la modélisation d'agents. Cette extension concerne :

- Le diagramme de classes, qui représente la structure statique du SMA.
- Les diagrammes d'interactions comme le diagramme de séquence. Ceux-ci permettent de modéliser le comportement dynamique du système.

3.2 Le diagramme de séquence AUML :

Les diagrammes d'interaction sont des diagrammes dynamiques, qui décrivent le comportement collectif des agents. On retrouve dans les diagrammes d'interaction deux types de diagramme [16] :

1. Le diagramme de séquence.
2. Le diagramme de collaboration.

Le diagramme de séquence est défini comme suit en UML : "*le diagramme de séquence est un diagramme qui montre les interactions des objets disposés dans la séquence temporelle. En particulier, elle montre les objets participant à l'interaction et la séquence de messages échangés. Contrairement à un diagramme de collaboration, un diagramme de séquence comprend des séquences de temps, mais ne comprend pas les relations entre objets.*"

Le diagramme de séquence AUML se compose de deux dimensions :

- Verticale : pour le temps.
- Horizontale : pour les différents rôles ou bien pour les agents jouant un rôle spécifique.

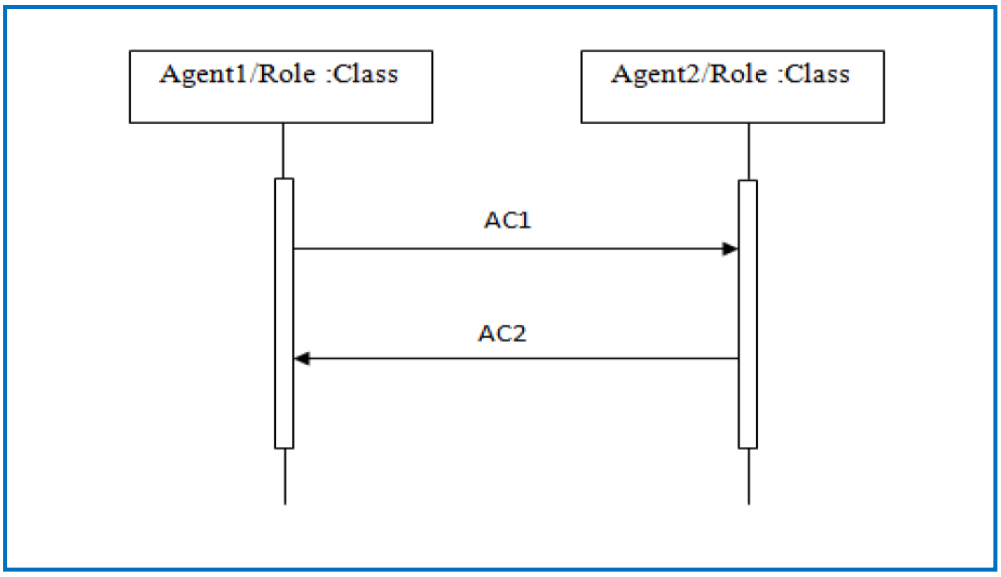

Figure 3.2 : format de base pour la communication

3.3 Le diagramme de flux de données

Le **diagramme de flux de données** (Data **Flow Diagram**) (**DFD)** est un type de représentation graphique du flux de données à travers un système d'information. Cet outil est souvent utilisé comme étape préliminaire dans la conception d'un système afin de créer un aperçu de ce système d'information. De plus, il est également utilisé pour visualiser le traitement de données (*structured design*).

Il montre quel type d'informations entre (*input*) ou sort (*output*) du système, d'où elles proviennent et où elles sont stockées. Cependant, il n'indique ni la temporalité des transmissions de données, ni l'ordre dans lequel les données circulent. **[17]**

4 La génération des cas de test

4.1 Test Manuel

Test Manuel (référence) était le premier style de test, mais il est encore largement utilisé. Le plan de test donne un aperçu de haut niveau des objectifs de test, tels que quels aspects du SUT doivent être testés, quels types de stratégies de test doit être utilisé, à quelle fréquence les tests doivent être effectués et combien de tests sera fait.

La conception du test est effectuée manuellement, sur la base des exigences informelles documents. Le résultat de la phase de conception est un document lisible par l'homme qui décrit les cas de test souhaités. La description des cas de test peut être assez concis et de haut niveau ; de nombreux détails de bas niveau sur l'interaction avec le système sous test peut être laissé au bon sens de l'exécution du test personne, qui s'appelle un testeur manuel. Cependant, la conception manuelle du les tests prennent du temps et ne garantissent pas une couverture systématique du SUT (le système sous test). **[18]**

4.2 Fonctionnalité du Test Manuel

L'exécution du test se fait également manuellement. Pour chaque cas de test, *l'homme- testeur* suit les étapes de ce cas de test, interagit directement avec le SUT, compare la sortie SUT avec la sortie attendue et enregistre le test verdict.
Dans les applications embarquées, où il n'est souvent pas possible d'interagir directement avec le SUT (il peut s'agir simplement d'une boîte noire avec des fils out), un environnement d'exécution de test peut être utilisé pour permettre au testeur d'entrer entrées et observer les sorties. Cependant, l'exécution de chaque cas de test est toujours effectuée manuellement.

Notez que les compétences requises du concepteur de test et du testeur manuel sont Plutôt différent. Le concepteur de test doit avoir une connaissance approfondie des SUT, plus quelques compétences en stratégies de conception de tests. Le testeur manuel a une tâche beaucoup plus subalterne, qui nécessite une certaine connaissance de la façon d'interagir avec le SUT, mais consiste principalement à suivre simplement les étapes du cas de test et enregistrer les résultats). **[19]**

Ce processus d'exécution de test manuel est répété chaque fois qu'une nouvelle version de le SUT doit être testé. Cela devient rapidement très ennuyeux et chronophage. Tâche consommatrice. Puisqu'il n'y a pas d'automatisation de l'exécution des tests, le coût de tester chaque version SUT est constante et importante. En fait, le coût de la répétition l'exécution manuelle des tests est si élevée que, pour maintenir les coûts de test dans les limites du budget, il est souvent nécessaire de couper les coins ronds en réduisant le nombre de tests qui sont exécutés après chaque évolution du code. Cela peut entraîner des logiciels être livré avec des tests incomplets, ce qui introduit un risque important concernant la maturité, la stabilité et la robustesse du produit.

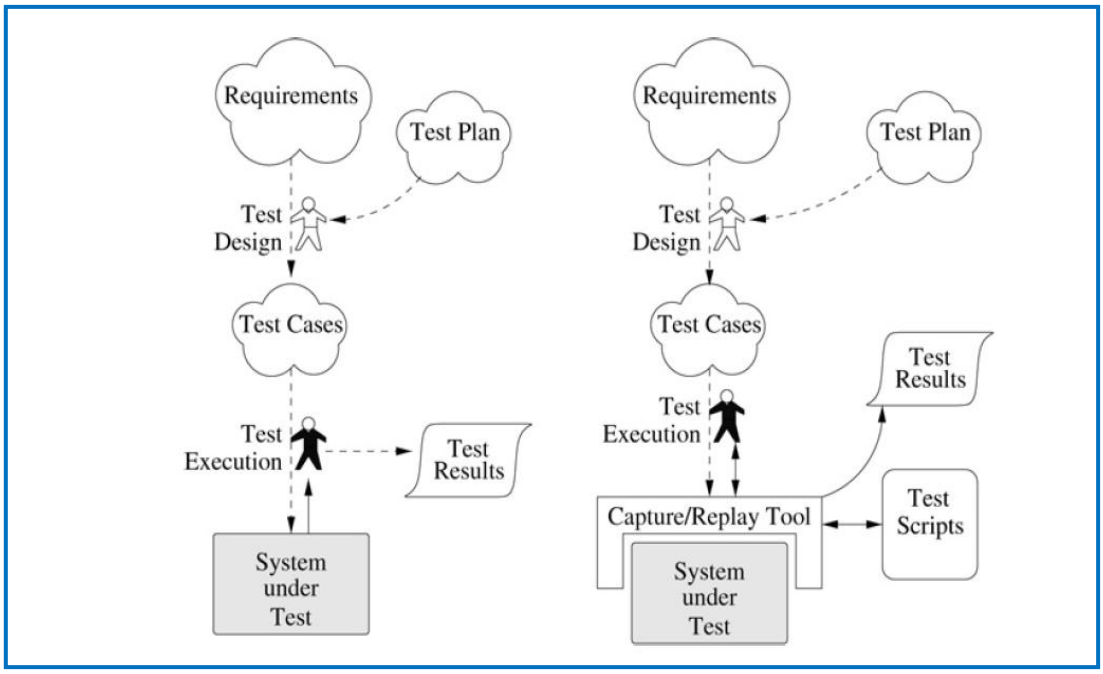

Figure 3.3 : Un processus de test manuel (à gauche) et un processus de test de capture/relecture (droit).

4.3 Test automatisé

Contrairement au test manuel, le test automatisé s'exécute sans l'intervention d'un humain. Cette méthode nécessite l'utilisation de solutions informatiques afin d'exécuter les actions prédéterminées dans un script et analyser le produit sur des parcours bien précis. **[20]**

4.4 Pourquoi opter pour le test automatisé ?

Le test automatisé a pour objectif de simplifier autant que possible les efforts de test grâce aux scripts. Le test est alors exécutés selon celui-ci, les résultats sont signalés et comparés aux résultats des essais antérieurs.

Son principal intérêt réside dans le fait qu'il permet de gagner du temps et de l'argent. En termes de budget, il rend possible des économies sur les charges car l'humain est moins sollicité, si ce n'est pour effectuer la maintenance du test.

En ce qui concerne le temps, le caractère répétitif du test automatisé permet de tester les applications en continu mais aussi de tester plus et mieux. C'est notamment le cas pour les tests de non-régression. Cela favorise une mise en production plus rapide ainsi qu'une réduction des délais de livraison. Aussi, le test automatisé permet une flexibilité au niveau du temps : les tests peuvent être exécutés en dehors des horaires de travail.

Enfin, les campagnes de test peuvent être tracées grâce à l'automatisation des tests car les automates préservent l'historique de l'exécution des tests. Les chefs d'équipe peuvent ainsi assurer un suivi fiable de la qualité d'exécution des tests. **[20]**

4.5 Le test automatisé : les points à ne pas négliger

Malgré le fait que les tests automatisés présentent de nombreux avantages pour les entreprises, leur mise en place doit être préparée et mûrement réfléchie afin d'éviter un éventuel échec.

Les gains de temps et d'argent qu'ils promettent ne peuvent se faire que sur la durée. En effet, la mise en place des tests automatisés nécessite un investissement conséquent comparé aux tests manuels, notamment au niveau du temps (l'exécution, l'enregistrement, la variabilisation ainsi que la création de procédures sont chronophages).

De plus, la mise en place de cette méthode de test nécessite non seulement de la recherche, mais aussi du matériel, des logiciels et des personnes ayant des notions en développement.

Tous les cas de tests ne peuvent pas être automatisés, **ou sont difficilement automatisables. Il est important de réfléchir en amont à ce qui va et pourra être automatisé, et travailler main dans la main avec les équipes de développement pour faciliter l'intégration des tests automatisés en voyant s'il est possible de les automatiser.** Il y a moins d'intérêt à tester un cas de test qui a peu de probabilité d'arriver. **Afin de décider si un cas de test peut être automatisé ou non,** plusieurs questions sont à se poser :

- l'interface graphique de la fonctionnalité est-elle stable ? l'application doit-être stable afin de supporter le test automatisé.
- le scénario est-il lourd à reproduire ?
- le test manuel est-il performant ?
- le scénario des campagnes de test est-il répétitif ?

Le test automatisé nécessite une mise à niveau des compétences des testeurs. Ceux-ci devront être formés afin que leurs compétences soient adaptées. Ils devront au préalable disposer de connaissances en programmation afin qu'ils puissent utiliser l'outil d'automation de manière optimale et ainsi mieux résoudre les problèmes liés aux tests de l'application **[20]**

Autre point à ne pas négliger : l'entreprise elle-même. En effet, des changements en termes d'organisation et de culture d'entreprise sont à prévoir. Un tel changement n'étant pas anodin, il pourra susciter une certaine résistance de la part des employés.

Pour conclure, l'intelligence artificielle ne remplacera jamais l'intelligence humaine. Les tests automatisés doivent impérativement aller de pair avec les **tests manuels**, qui devront être effectués à certains niveaux (tests de convivialités) afin de confronter le produit à une utilisation complexe en conditions réelles. Par conséquent, cela permettra de réduire les bugs de manière significative et ainsi livrer une application de meilleure qualité et d'avoir une satisfaction client plus importante.

Dans notre cas d'étude on va générer des cas de test automatique pour le système multi-agent choisi, à partir du diagramme de séquence AUML, en utilisant le test basé model qui est capable de couvrir les interactions qui existent entre les agents du système présentées dans ce diagramme. Les cas de test obtenus par cette étape seront utilisés par la suite pour détecter les erreurs. Notre approche consiste à utiliser le test basé model qui va générer des cas de test d'une manière automatique en utilisant l'outil **ModelJUnit**.

4.6 Définition d'outil ModelJUnit

ModelJUnit est un ensemble de bibliothèques open source composé d'un ensemble de classes Java pour les tests basé modèle. Les bibliothèques ont été conçues par Dr.Mark Utting. Il permet la génération de séquences de test à partir modèles FSM / EFSM écrit en Java et permet de mesurer différentes couvertures de test. ModelJUnit permet en particulier d'automatiser à la fois la génération de tests et leur exécution.

5 La concrétisation

Sur la base de modèle abstrait, des cas de test peuvent être dérivés sous la forme de suites de tests. Ces suites de tests ne sont pas directement exécutables, car elles n'ont pas le même niveau d'abstraction que le code exécutable. Cela demande souvent une intervention manuelle de la partie d'un ingénieur de test qui doit concevoir une couche d'adaptation permettant de passer d'une suite de tests abstraits en suite de tests exécutables.

6 L'exécution des cas de test

Cette étape consiste à exécuter les agents avec les entrées décrites dans les cas de test et de comparer les résultats obtenus avec ceux attendus. La non-conformité entre les résultats obtenus et les résultats attendus signifie qu'il existe une erreur d'interaction et la non-conformité entre la post-condition obtenue et celle attendue signifie qu'il y a une erreur de scénario.

7 Etude de cas

Pour valider notre approche, nous l'appliquons sur une étude de cas concrète ***: Une application multi-agents pour les ventes des livres :*** Ce SMA contient :

- Des agents qui demandent l'achat d'un livre.
- Des autres agents qui vendent des livres.
- L'acheteur devrait choisir l'agent qui offre le meilleur prix à travers un processus de communication entre les agents.

7.1 La modélisation :

7.1.2 Présentation du diagramme de séquence :

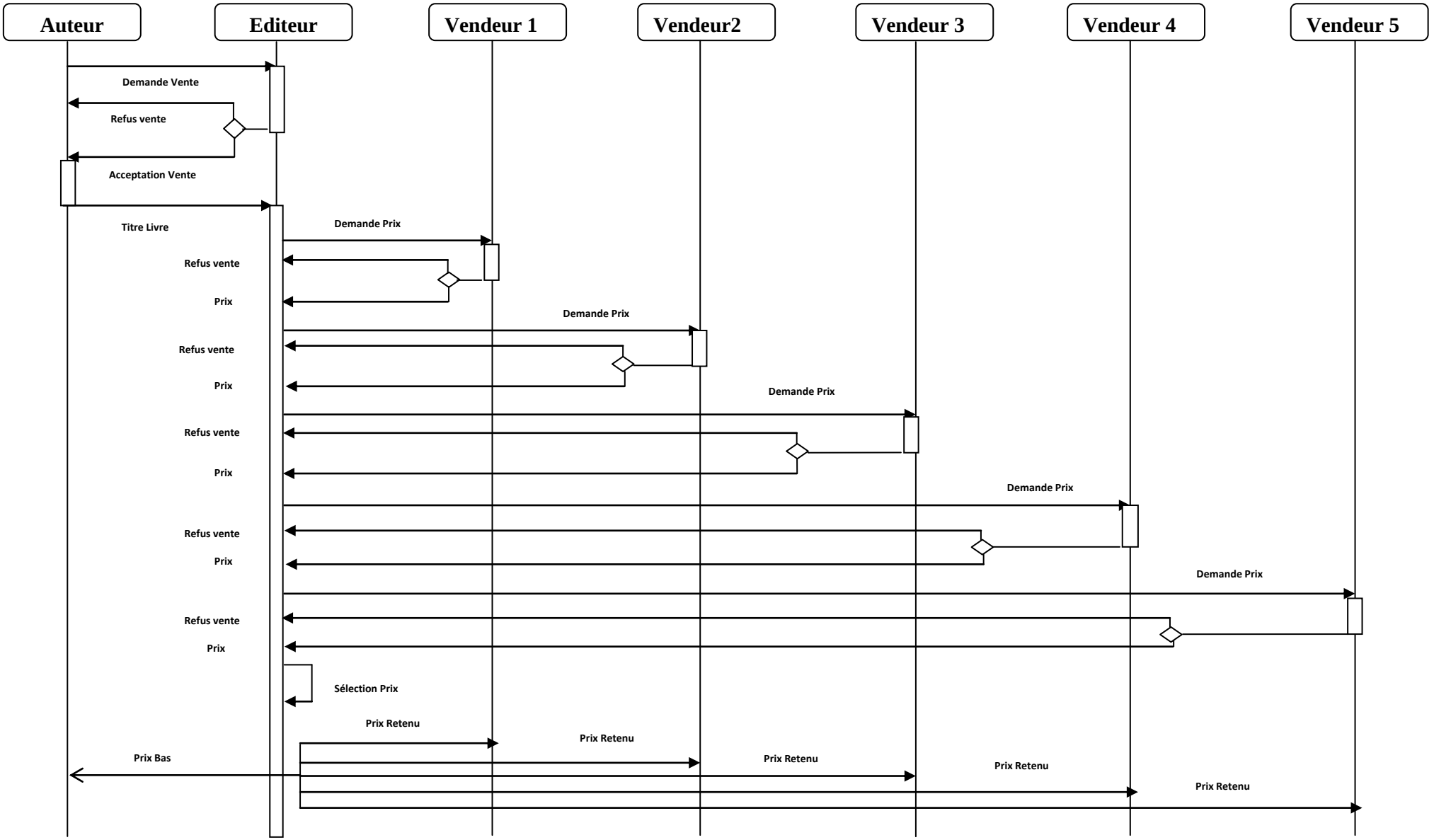

Figure : Diagramme de séquence

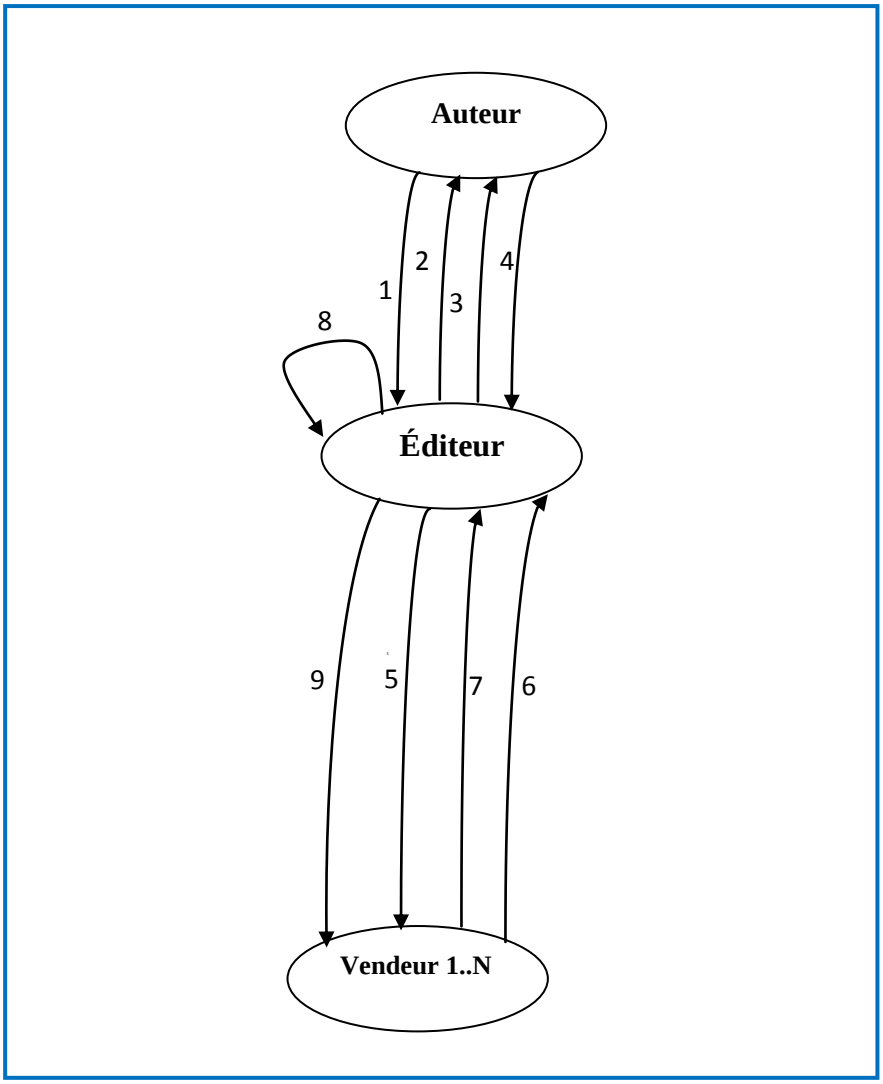

Figure : Diagramme de Flux de données

1 : Demande de vente

2 : acceptation de vente

3 : refus de vente

4 : Titre Livre

5 : Demande prix

6 : Refus de vente

7 : Acceptation de vente

8 : Sélection du Prix

9 : Prix Retenu

8 Conclusion :

Au cours de ce chapitre, nous avons fait la conception de notre projet, a fin de réaliser une analyse profonde de la solution adaptée en précisant les différentes étapes du processus de test basé modèle. Dans le chapitre suivant, nous allons entamer la phase de l'implémentation.

1 Introduction :

Les SMA sont conçus et implantés idéalement comme un ensemble d'agents interagissant selon les modes de coopération, de concurrence ou de coexistence. Ces agents peuvent être des agents réactifs, des agents cognitifs ou bien des agents hybrides, suivant leur rôle. La résolution d'un problème par un SMA permet dans un premier temps, de n'affecter à chaque agent qu'une partie du problème complexe à résoudre et dans un second temps, de transformer des contraintes globales en contraintes locales qui sont résolues par coordination entre les agents.

Pour construire un SMA il est préférable d'utiliser une plate-forme multi-agent qui offre un ensemble d'outils utilisé pour la construction et la mise en service d'agents au sein d'un environnement spécifique. Les outils peuvent être utilisés pour analyser les SMA, créer les SMA ou bien tester les SMA. Ces outils peuvent être sous la forme d'environnement de programmation (API) et d'applications permettant d'aider le développeur à la programmation d'un SMA ainsi que son débogage. Il existe de nombreuses plates-formes pour le développement de SMA : JACK, Jadex, Madkit, JADE…etc.

Nous avons opté dans le cadre de notre travail pour la plateforme JADE (Java Agent DEvelopment framework) pour les avantages qu'elle procure et qui nous allons la présenter dans le reste de ce chapitre.

2 Eclipse :

Eclipse est un projet, décliné et organisé en un ensemble de sous-projets de développements logiciels, de la fondation Eclipse visant à développer un environnement de production de logiciels libre qui soit extensible, universel et polyvalent, en s'appuyant principalement sur Java.
Son objectif est de produire et fournir des outils pour la réalisation de logiciels, englobant les activités de programmation (notamment environnement de développement intégré et frameworks) mais aussi d'AGL recouvrant modélisation, conception, test, gestion de configuration, reporting… Son EDI, partie intégrante du projet, vise notamment à supporter tout langage de programmation à l'instar de Microsoft Visual Studio.
Bien qu'Eclipse ait d'abord été conçu uniquement pour produire des environnements de développement, les utilisateurs et contributeurs se sont rapidement mis à réutiliser ses briques logicielles pour des applications clientes classiques. Cela a conduit à une extension du périmètre initial d'Eclipse à toute production de logiciel : c'est l'apparition du framework Eclipse RCP en 2004.
Figurant parmi les grandes réussites de l'open source, Eclipse est devenu un standard du marché des logiciels de développement, intégré par de grands éditeurs logiciels et sociétés de services. Les logiciels commerciaux *Lotus Notes 8*, *IBM Lotus Symphony* ou *WebSphere Studio Application Developer* sont notamment basés sur Eclipse.

3 JADE (Java Agent Development Framework) :

JADE (Java Agent Development Framework) est une plateforme multi agents répartie (multi-hôtes) développé par F.Bellifemine & A. Poggy, G. Rimassa, P. Turci par la société Telecom Italia Lab « Tilab, anciennement CSELT » en 1999 **[Lot05].** JADE a comme but :

- Simplifier la construction des systèmes multi-agents « SMA » interopérables.
- L'exécution des SMA.
- La réalisation d'applications conformes avec le standard FIPA (FIPA, 1997) pour
- Faciliter la communication des agents JADE avec des agents non JADE.
- Essayer d'optimiser les performances d'un système d'agent distribué.

JADE développée en Java, fonctionne sous tous les systèmes d'exploitation **[5],** inclut tous les composants obligatoires qui contrôlent un SMA **[Oli02],** et possède une architecture très précise permettant la construction dite « normalisée » d'agents **[9].** Pour tout cela la plateforme JADE contient :

- **Un runtime Environment** : l'environnement où les agents peuvent vivre. Il doit être activé pour pouvoir lancer les agents.
- **Une librairie de classes :** que les développeurs utilisent pour écrire leurs agents.
- **Une suite d'outils graphiques:** qui facilitent le débogage, la gestion et la supervision de la plateforme des agents.

JADE fournit aussi des classes qui implémentent JESS pour la définition du comportement des agents **[Fer05]:** JESS (outil de raisonnement à base de règles) est le moteur qui exécute tout le raisonnement nécessaire. On a donc la possibilité de créer des agents JADE intelligents en délégant le raisonnement à d'autres outils**.**

Chaque instance du JADE est appelée conteneur « container en anglais », qui peut contenir plusieurs agents. Ainsi qu'un ensemble de conteneurs constituent une plateforme et chaque plateforme doit contenir un conteneur spécial appelé *main-container* et tous les autres conteneurs s'enregistrent auprès de celui-là dès leur lancement.
Un *main-container* se distingue des autres conteneurs car il contient toujours deux agents spéciaux appelés AMS et DF qui se lancent automatiquement au lancement du **maincontainer**.

Ainsi on a un conteneur par machine et lorsque l'on lance un conteneur :

- Soit il est seul, il devient alors le conteneur principal (main container)
- Soit on lui indique l'adresse du conteneur principal **[10].**

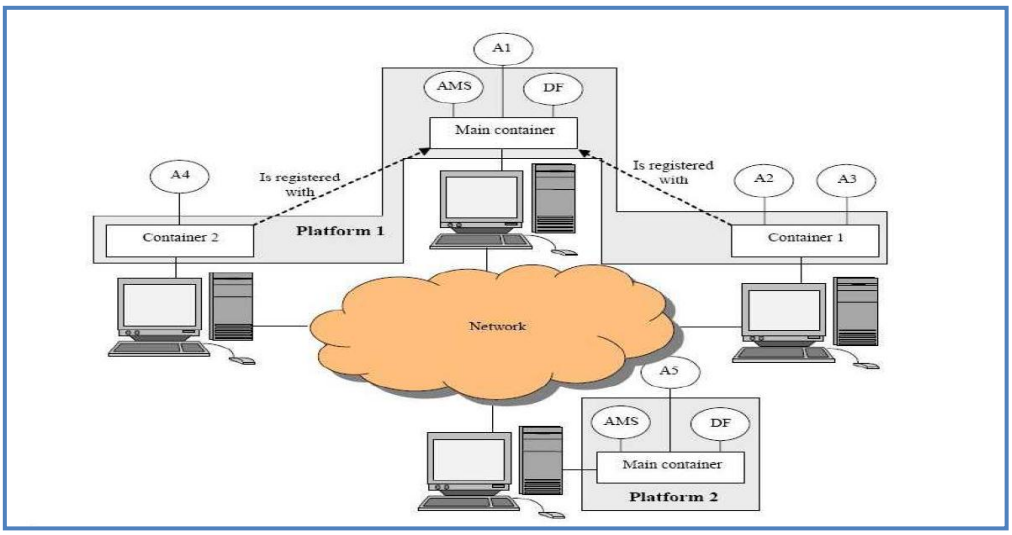

Figure 4.1 : les conteneurs dans JADE

4 Définition ModelJUnit :

ModelJUnit est une bibliothèque Java et interface graphique pour la prise en charge des tests basés sur des modèles. Les modèles sont des machines à états finis étendues (EFSM) écrites en Java.

Dans ModelJUnit, modèle de test (FSM) doit être écrit comme une classe java qui implémente plusieurs interfaces définies dans la bibliothèque tels que :

- **FsmModel**: Il s'agit de l'interface de base que toute classe (E) FSM doit mettre en œuvre pour la génération de tests basé modèle

- **TimedFsmModel:** Il s'agit d'une interface spéciale qui étend les fonctionnalités de l'interface FsmModel et construit le FSM qui utilise le framework temporisé de ModelJUnit

La classe de modèle créé à l'aide des interfaces ci-dessus doit avoir des méthodes suivantes:

- **Object getState() :** Cette méthode renvoie l'état actuel du modèle, qui est généralement un objet. Elle exécute la tâche de mappé l'état interne du modèle EFSM à l'état visible réelle représentée dans le modèle par le concepteur du modèle

- **void reset (boolean) :** Cette méthode effectue la tâche de remise à zéro du SUT à l'état initial ou la création nouvelle instance de la classe SUT. Elle est utilisée généralement dans les tests online, où nous avons besoin de réinitialiser SUT à l'état initial. La valeur par défaut du paramètre booléen est "vrai".

- **@Action void name () :** Ces types de méthodes sont utilisées pour définir les actions dans le modèle. Ces actions modifient l'état d SUT. Il peut y avoir plus d'une

méthoded'action définies dans un modèle. Ces méthodes ne nécessite que les annotations @ action et n'ont pas de paramètres. Chaque méthode d'action ne peut exister avec ou sans guard. Ce gaurd contrôle l'activation de l'action lors de la génération de la séquence de test. Dans le cas où ce gaurd n'est pas définit sa valeur par défaut est vraie

- **boolean nameGuard() :** Il définit le gaurd pour les méthodes d'action définis dans le modèle. Cette méthode renvoie toujours booléen. Le nom de la méthode doit être le même que le nom de l'action (pour qui le gaurd est défini), avec le mot ajouté «gaurd», à son extrémité.

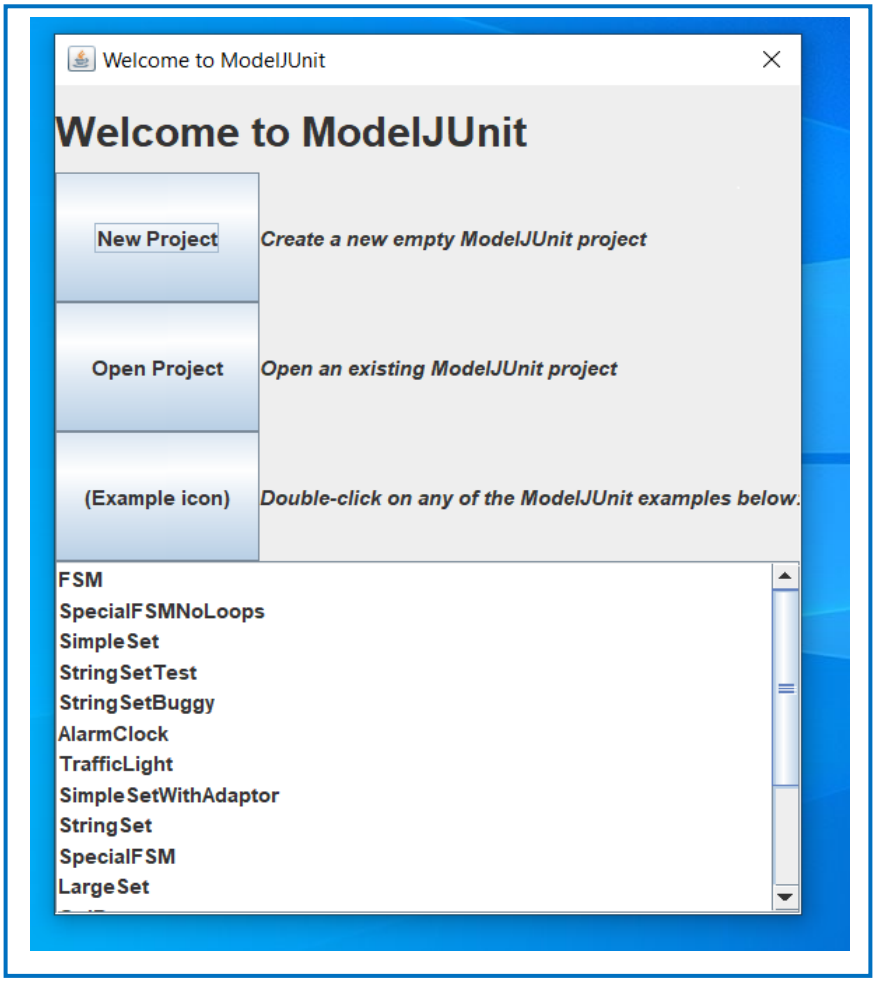

Figure 4.2 : Interface ModelJUnit

4.1 La Modélisation du graphe de flux de données avec l'outil *ModelJUnit* :

4.1.1 Le code source :

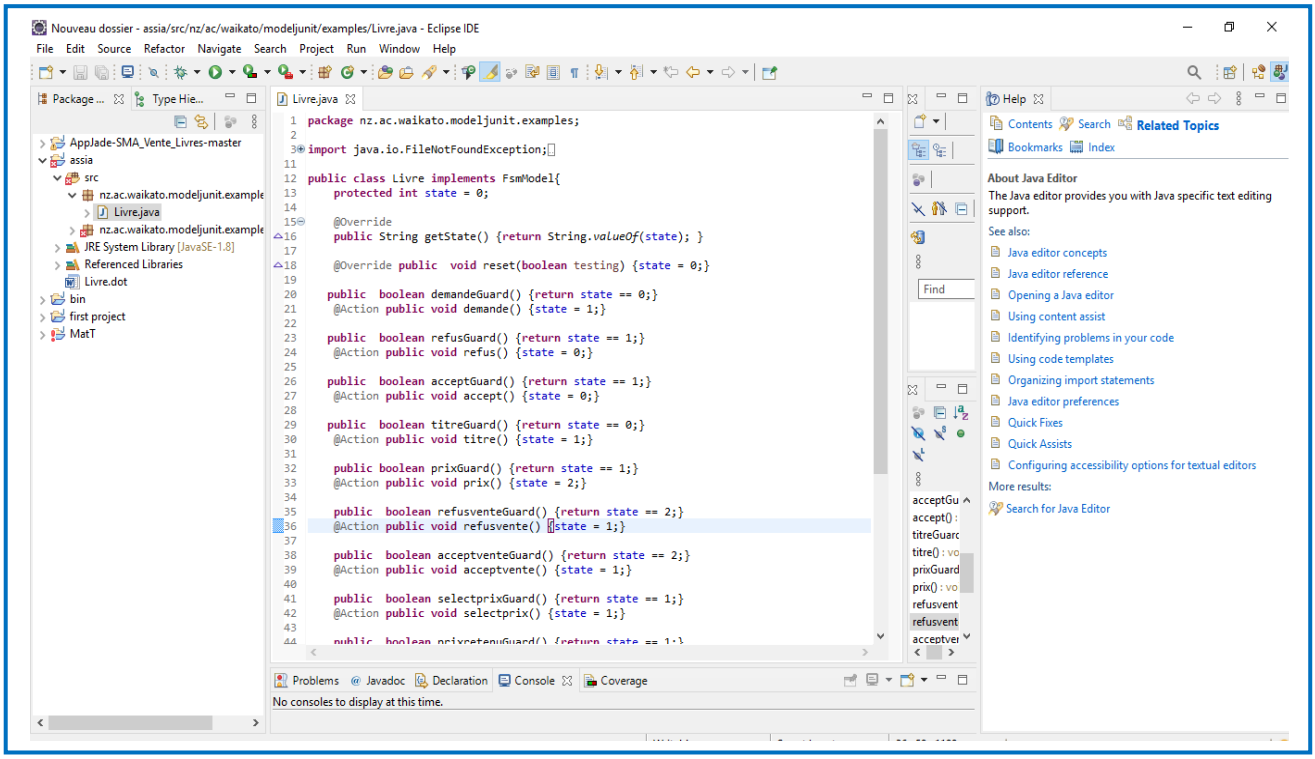

Figure 4.3 : Le code source

4.1.2 La spécification :

```
digraph Livre
{
  "0" -> "1"  [label="titre"];
  "1" -> "0"  [label="accept"];
  "1" -> "0"  [label="refus"];
  "0" -> "1"  [label="demande"];
  "1" -> "1"  [label="selectprix"];
  "1" -> "2"  [label="prixretenu"];
  "2" -> "1"  [label="acceptvente"];
  "2" -> "1"  [label="refusvente"];
  "1" -> "2"  [label="prix"];
}
```

4.1.3 La modélisation

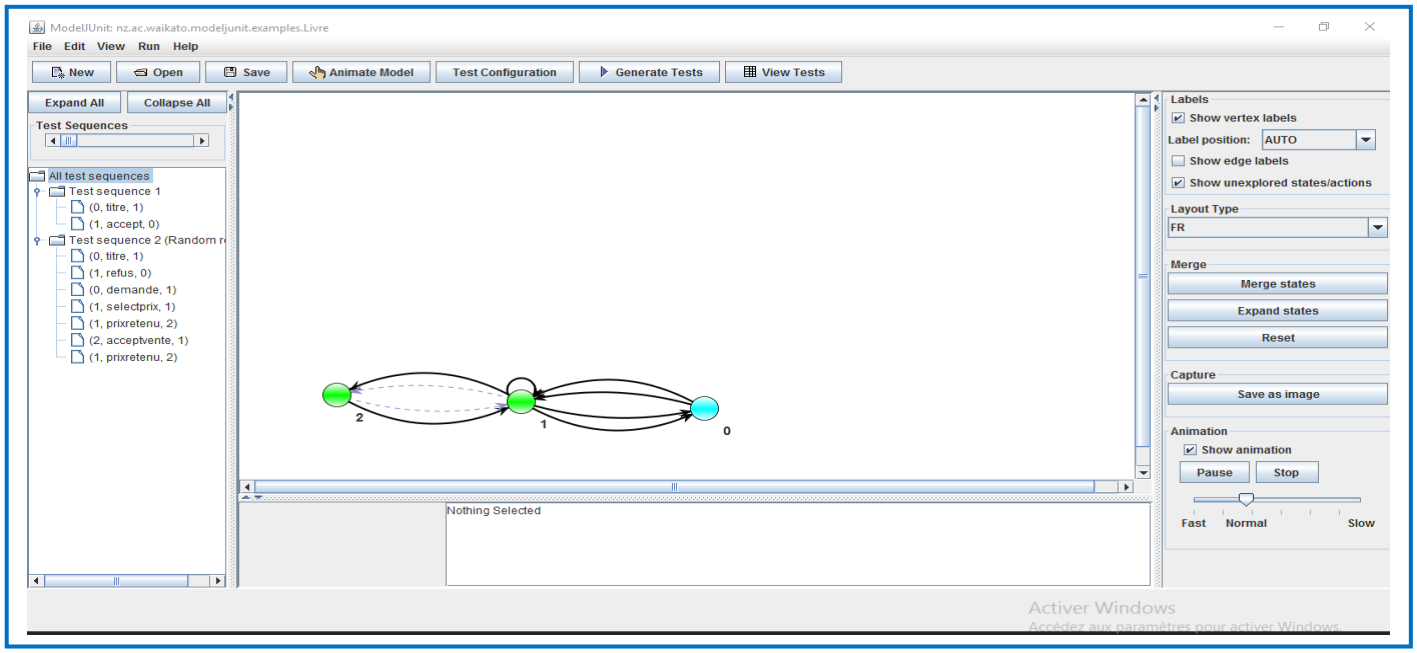

Figure 4.4 : Graphe de Flux de données

5 Présentation de l'outil développé :

Une fois l'application est lancée, l'interface, présentée par la figure 4.5 apparait.

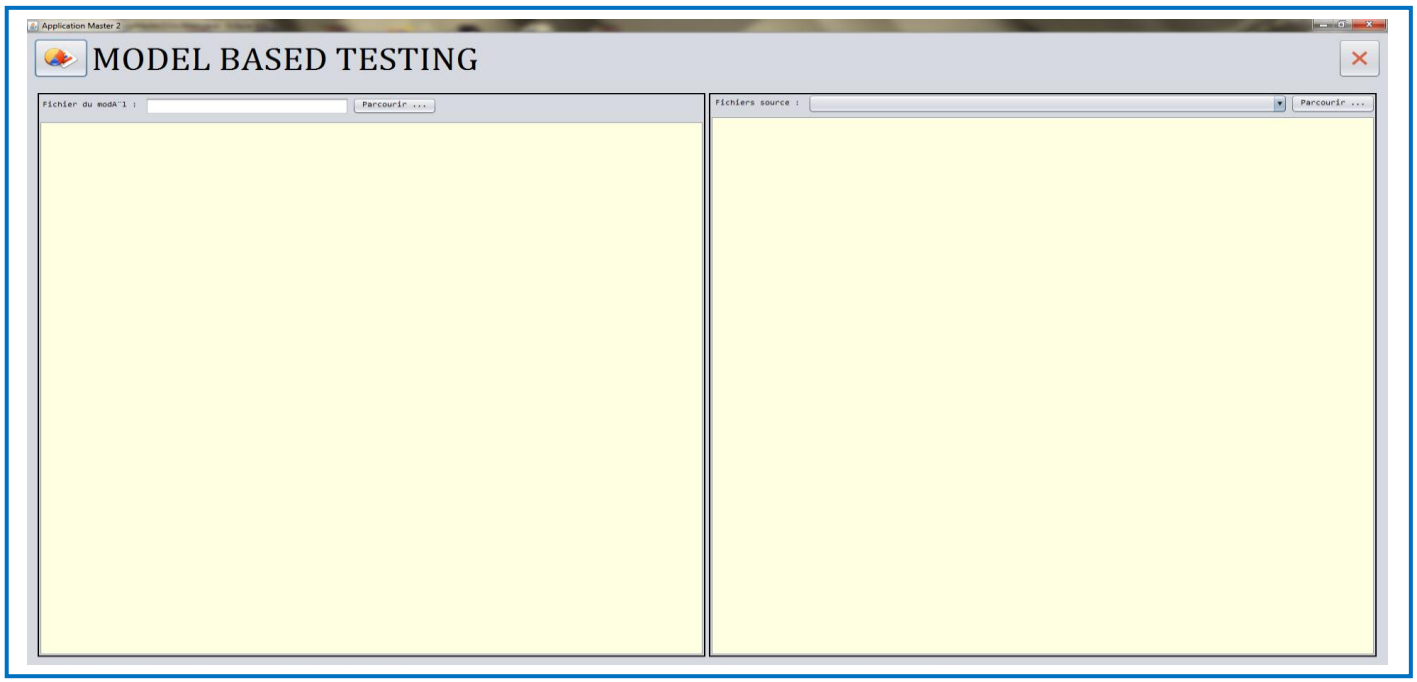

Figure 4.5 : Interface initiale

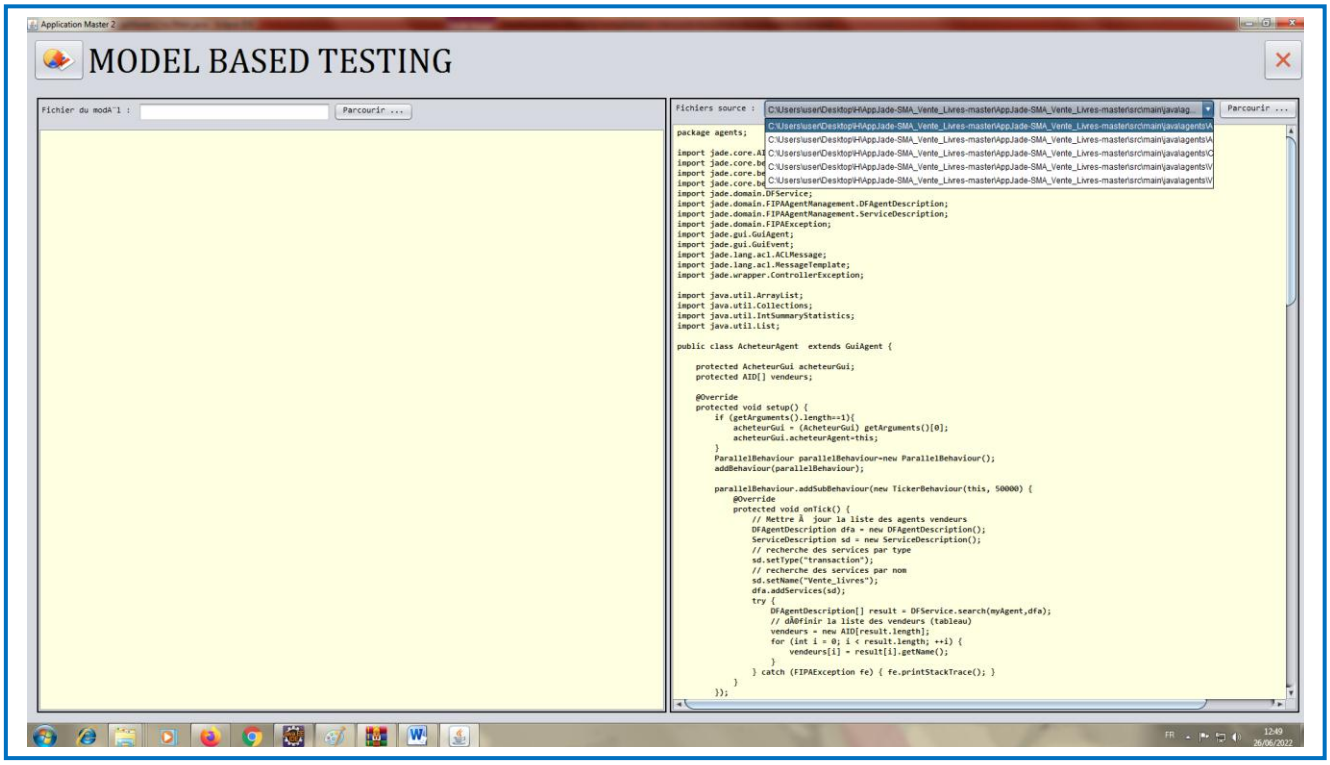

Figure 4.6 : le choix de l'exemple à tester

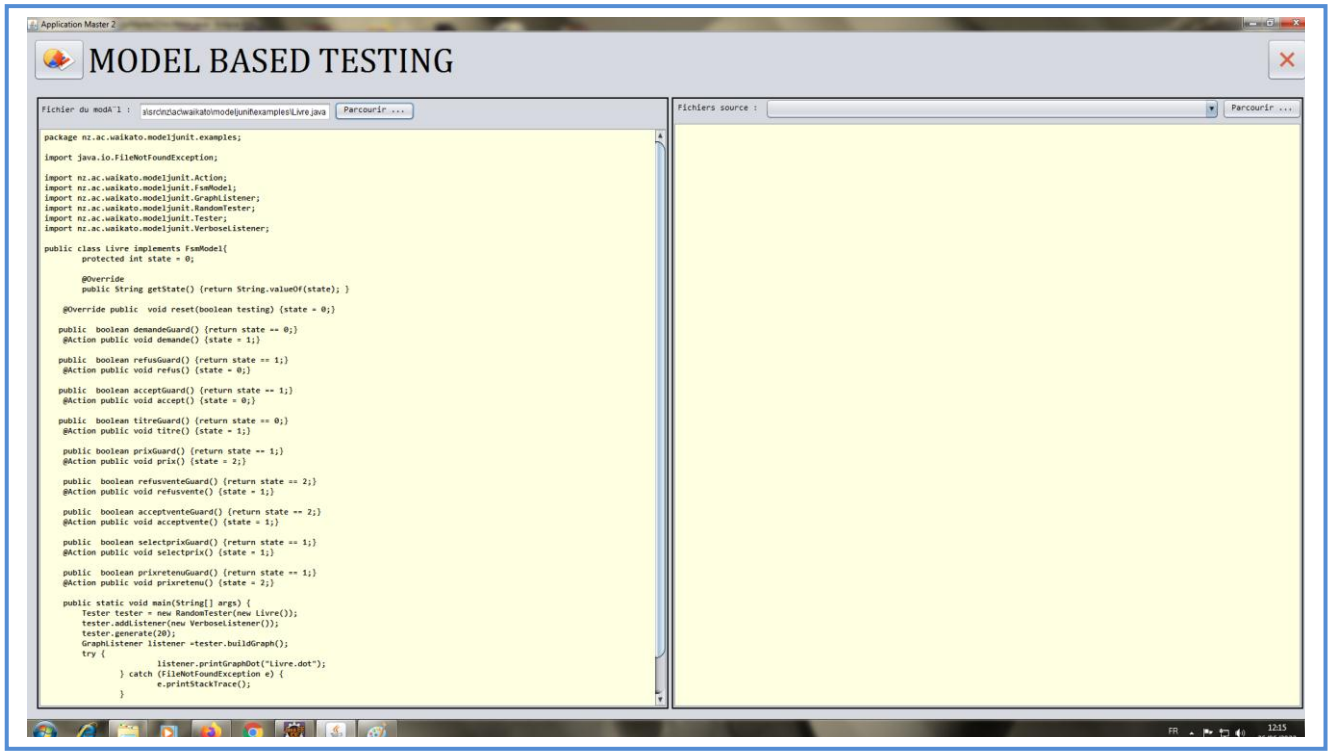

Figure 4.7 : le choix du Model

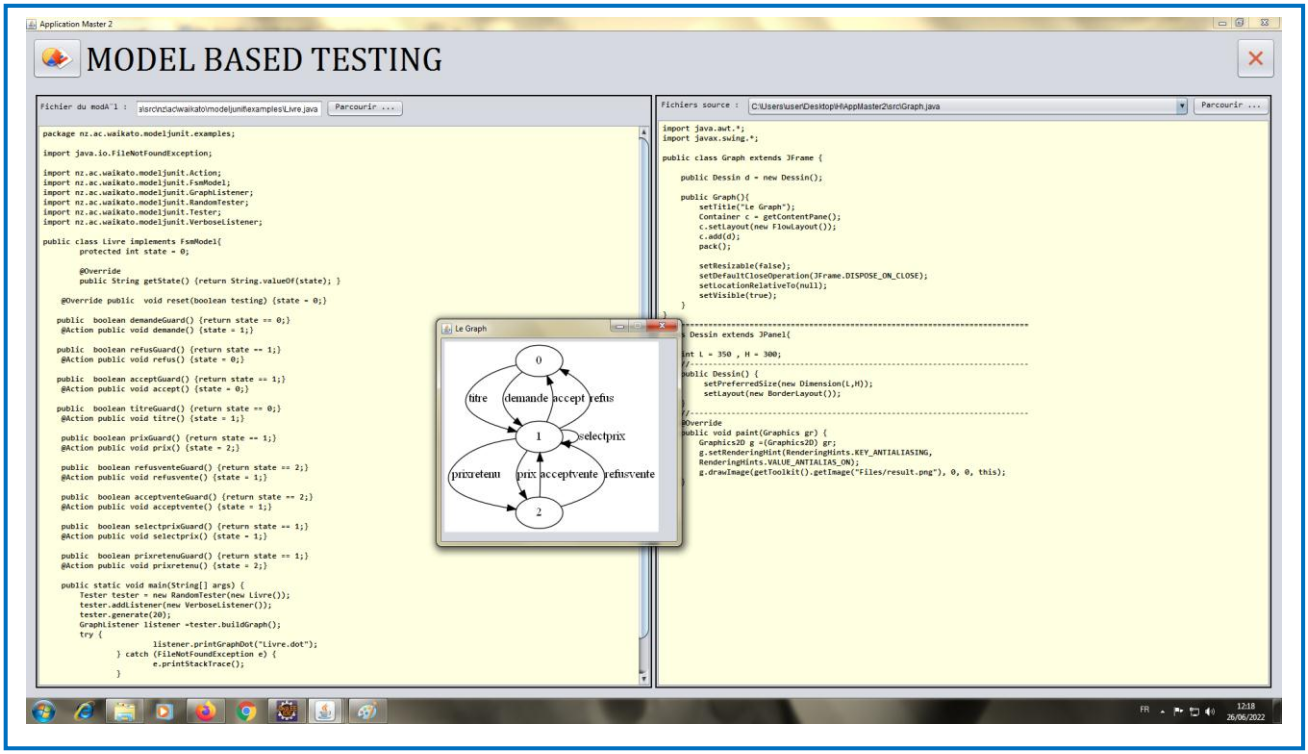

Figure 4.8 : présentation du graphe

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.

Dans ce travail, nous avons présenté une approche de test basé modèle pour les SMA. Cette approche capable de concevoir et de dériver (de manière automatique ou non) des cas de tests à partir d'un modèle abstrait et haut niveau du *système sous test* (SUT). Le modèle est dit abstrait car il offre bien souvent une vue partielle et discrète des comportements attendus d'un logiciel ou d'un système.

Sur la base de modèles abstraits, des cas de test peuvent être dérivées sous la forme de *suites de tests*. Ces suites de tests ne sont pas directement exécutables, car elles n'ont pas le même niveau d'abstraction que le code exécutable. Cela demande souvent une intervention manuelle de la part d'un ingénieur de test qui doit concevoir une *couche d'adaptation* permettant de passer d'une suite de tests abstraites en suite de tests exécutables. Cette étape est généralement appelée *étape de concrétisation*.

Une fois les cas de tests exécutés, une comparaison est possible entre le comportement réel du logiciel (le logiciel développé) et le comportement attendu (décrit dans le modèle). La comparaison entre ce qui est attendue et ce qui se passe réellement permet d'assigner un *verdict* de test. Un test est dit non-passant lorsque le comportement réel du logiciel, ou du système, diffère du comportement attendu.

Cependant, certains utilisateurs affirment que l'utilisation des MBT peut être un réel retour sur investissement avec un gain de productivité et une qualité augmentée.

En effet, l'automatisation des tests a des avantages directs pour les équipes responsables des tests :

- Si le modèle est bien fait, évite des cas de test mal conçus, défectueux ou manquants, du même coup, accroît la couverture de test
- Réduit les coûts pour les tests (tests de non régression)
- Améliore la qualité du processus de test
- Réduction des délais d'exécution des tests

Et également des avantages indirects pour les utilisateurs du système d'information :

- Diminue les efforts de maintenance des jeux de tests
- Renforce la qualité de la documentation des exigences
- Crée une plateforme commune pour les designers et les testeurs

L'automatisation permet d'exécuter des tests à un coût marginal très faible, après un investissement initial significatif et en conception ou maintenance.

Dans notre travail on a testé **« *l'application Multi-agents pour les ventes des livres* »,** on a générer des cas de test basée sur le diagramme de séquence AUML**.**

A la fin nous estimons que le test de logiciel est un domaine très difficile à cause de ses ambigüités que nous avons tentées de lever par notre modeste travail.

Bibliographie :

[1] Agent intelligent Cours web. Technologie d'information et de la communication et appropriation de savoir. http://turing.cs.pub.ro/auf2/html/chapters/chapter2/

[2] I. Chadès, "Planification distribuée dans les systèmes multi-agents à l'aide de processus décisionnels de Markov ", Thèse de doctorat, Université Henri Poincaré-Nancy 1, 2003.

[3] Austin J, Quand dire c'est faire, 1962, trad. fr. 1970, rééd. Seuil, coll. « Points essais »,1991.

[4] https://www.techno-science.net/definition/813.html

[5] Wikipedia

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

[7] Paulson, L. 1993. The Isabelle Reference Manual. Technical Report 283 pour générer des cas de test à partir de spécifications Z.

[8] 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

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

[10] 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

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

[12] Dans le niveau d'agent on trouve le travail de **Nguyen et al** 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.

[13] Serrano et Botia 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. Serrano et al. 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.

[14] r.wikipedia.org/wiki/Méthode_formelle_(informatique)#:~:text=En%20informatique %2C%20les%20méthodes%20formelles, rapport%20à%20une%20certaine%20spécification.

[15] http://cotechnoe.com/cartographier-les-connaissances-et-les-ontologies_articles/GeCSO2012_OntoCASE_une_approche_d_elicitation_semi_formelle_graphique_et_son_outil.pdf

[16] Mémoire Génération des diagrammes AUML à partir de programmes JADE

[17] https://fr.wikipedia.org/wiki/Diagramme_de_flux_de_donn%C3%A9es#:~:text= Le%20Data%20Flow%20Diagram%20(DFD,de%20ce%20syst%C3%A8me%20 d'information.

[18] PRACTICAL MODEL-BASED TESTING _ A TOOLS APPROACH_MARK UTTING AND BRUNO LEGEARD.

[19] PRACTICAL MODEL-BASED TESTING _ A TOOLS APPROACH_MARK UTTING AND BRUNO LEGEARD.
[20] https://www2.stardust-testing.com/blog-fr/les-enjeux-du-test-automatis%C3%A9
[21] https://www2.stardust-testing.com/blog-fr/les-enjeux-du-test-automatis%C3%A9
[22] https://www2.stardust-testing.com/blog-fr/les-enjeux-du-test-automatis%C3%A9.