REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université El Hadj Lakhdar - BATNA

Faculté des Sciences et des Sciences de l'ingénieur

Département d'informatique

N° d'ordre :…………
Série :…………

Mémoire
Présenté en vue de l'obtention du diplôme

Magister en Informatique

Option: **Informatique Industrielle**

SUJET DU MÉMOIRE :

UML et MODEL CHECKING

Présenté le : 09 /05 /2009

Par : **TIBERMACINE Okba**

Composition du jury:

Mr. BELATTAR Brahim Président (Maître de Conférence à l'Université de Batna)
Mr. CHAOUI Allaoua Rapporteur (Maître de Conférence à l'Université de Constantine)
Mr. BILAMI Azzeddine Examinateur (Maître de Conférence à l'Université de Batna).
Mr. KAZAR Okba Examinateur (Maître de Conférence à l'Université de Biskra).

Remerciements

Acknowledgments

الحمد لله و الصلاة على رسول الله

I think that I have to start from the Quote of Ghandi when he said:

Whatever you do will be insignificant, but it is very important that you do it.

Mahatma Gandhi

And it seems very important to recall what Isaac Newton said:

If I have seen further, it is by standing on the shoulders of giants.

Isaac Newton

So I have to say thanks for those who really respect and work for science.

This work couldn't be released without the help of the others, so I would like to thank Dr. Chaoui Allaoua for accepting giving me this project and for his time spent in reading and evaluating this work.

I would like to express my gratitude to Dr. Bellattar Brahim, Dr.BILAMI Azzedine and Dr. Kazar Okba, who accept to evaluate the present monograph.

Also, thanks to my family and friends who deserve my huge love, gratitude and recognition.

Okba

SOMMAIRE

INTRODUCTION GENERALE..1
CHAPITRE I : 3
UML & GENIE LOGICIEL..3
1. LE GENIE LOGICIEL...4
1.1 DEFINITION 4
1.2 CRITERES DE QUALITE D'UN PRODUIT LOGICIEL...4
1.3 CYCLE DE VIE D'UN LOGICIEL 5
1.4 MODELE DE CYCLE DE VIE D'UN PRODUIT LOGICIEL...6
1.4.1 Modèle de cycle de vie en cascade 6
1.4.2 Modèle de cycle de vie en V..6
1.4.3 Modèle de cycle de vie en spiral...7
1.4.4 Modèle de cycle de vie par Incréments 8
1.5 METHODES D'ANALYSE ET DE CONCEPTION..8
1.5.1 Méthodes fonctionnelle descendante : 8
1.5.2 Méthodes orientées objet 9
2. L'APPROCHE OBJET...9
2.1 L'OBJET..9
2.2 LES CLASSES...10
2.3 L'INSTANCIATION..10
2.4 LES RELATIONS ENTRE ASSOCIATIONS..10
2.5 LES MESSAGES & LA SYNCHRONISATION : 11
2.6 GENERALISATION ET SPECIALISATIONS 11
3. UNIFIED MODELING LANGUAGE...12
3.1 DEFINITION 12
3.2 LES VUES D'UML 13
3.3 LES ELEMENTS DU MODELE 14
3.4 MECANISMES GENERAUX...15
3.5 LES DIAGRAMMES 16
3.5.1 Diagrammes structurels 17
3.5.2 Les diagrammes comportementaux 21
3.5.3 Diagramme d'interaction (interaction diagram) 24
4. LE PROCESSUS UNIFIE 28

4.1 DEFINITION 28
4.2 VIE DU PROCESSUS UNIFIE 29
4.3 LES TACHES D'UN CYCLE DE VIE D'UP 31
4.4 LES PHASES D'UN CYCLE DE VIE D'UP 32
CHAPITRE II : 34
DIAGRAMMES D'ETATS-TRANSITIONS & TRAVAUX DE FORMALISATION 34
1. DIAGRAMME D'ETATS-TRANSITIONS 34
1.1 DEFINITION 34
1.2 ETATS 34
1.3 EVENEMENTS 36
1.4 TRANSITIONS 36
1.5 ETATS ET TRANSITIONS AVANCES 38
1.5.1 Activités d'entrée/sortie 38
1.5.2 Transitions Internes 38
1.5.3 Activités-Do 38
1.5.4 Evénements déférés 39
1.5.5 Sous-machine 39
1.6 SOUS-ETATS 39
1.6.1 Sous-état séquentiel 40
1.6.2 Sous-état Orthogonal 40
1.6.3 Etat Historique 41
1.7 POINT DE CHOIX 41
1.7.1 Point de Jonction 41
1.7.2 Point de Décision 42
1.8 TRANSITION COMPLEXES 42
1.9 SEMANTIQUE DE MACHINE D'ETAT 43
2. TRAVAUX DE FORMALISATION DE DIAGRAMME D'ETATS TRANSITIONS. 44
2.1 LES MODELES MATHEMATIQUES 44
2.2 LES SYSTEMES DE REECRITURE 48
2.3 LES APPROCHES DE TRANSLATIONS 50
CHAPITRE III : 54
LOGIQUE DE REECRITURE, MAUDE ET MODEL-CHECKING 54
1. LES METHODES FORMELLES 54
1.1 PREUVE DE THEOREME 54

1.2 EXPLORATION DE L'ENSEMBLE DES ETATS 55
2. LOGIQUE DE REECRITURE 55
2.1 PRESENTATION 55
2.2 CONCEPTS DE BASES DES THEORIES 57
3. LANGAGE MAUDE 66
3.1 PRESENTATION 66
3.2 MODULES FONCTIONNELLES 67
3.3 MODULES SYSTEMES 68
3.4 SPECIFICATIONS ORIENTEES OBJET EN MAUDE 68
3.5 EXECUTION DU MAUDE 70
3.6 COMMANDES MAUDE 72
4. MODEL-CHECKING ET MAUDE LTL MODEL-CHECKER 74
4.1 MODEL CHECKING 74
4.2 STRUCTURE DE KRIPKE ET LTL 74
4.3 MAUDE LTL MODEL-CHECKER 76
CHAPITRE IV : 80
METHODE DE MODEL-CHECKING DES MODELES UML EN UTILISANT MAUDE 80
1. PRESENTATION DE LA METHODE 80
1.1 ELABORATION DU MODELE UML 82
1.2 TRANSLATION DU MODELE UML 83
1.3 DEFINITION DES PROPRIETES A VERIFIER 90
1.4 VERIFICATION DU MODELE 91
2. EXEMPLE I : PROBLEME DE DINER DES PHILOSOPHES 93
2.1 MODELE UML POUR LE PROBLEME DE DINER DES PHILOSOPHES 93
2.2 TRANSLATION DU MODELE UML 96
2.3 DEFINITION DES PROPRIETES A VERIFIER 99
2.4 VÉRIFICATION DU MODÈLE 102
3. EXEMPLE II : ATM-BANK 106
3.1 MODÈLE UML ATM-BANK 106
3.2 TRANSLATION DU MODELE UML 108
3.3 VERIFICATION DES COLLABORATIONS 110
CONCLUSION 114
REFERENCES 116

LISTE DES FIGURES

FIGURE 1 MODELE EN CASCADE 6
FIGURE 2 MODELE EN V 7
FIGURE 3 DEVELOPPEMENT EN SPIRAL........ 7
FIGURE 4 MODELE PAR INCREMENTS 8
FIGURE 5 REPRESENTATION GRAPHIQUE D'UNE APPROCHE FONCTIONNELLE...... 9
FIGURE 6 LES VUES D'UML 13
FIGURE 7 EXEMPLES D'ELEMENTS DU MODELE........ 14
FIGURE 8 LES DIFFERENTS DIAGRAMMES D'UML 2.0........ 16
FIGURE 9 REPRESENTATION GRAPHIQUE DES CLASSES 17
FIGURE 10 EXEMPLE DE QUELQUES RELATIONS........ 17
FIGURE 11 EXEMPLE D'UN DIAGRAMME DE CLASSE 17
FIGURE 12 REPRESENTATIONS GRAPHIQUES D'UN OBJET 18
FIGURE 13 REPRESENTATION D'UN GROUPE D'OBJETS........ 18
FIGURE 14 EXEMPLE D'UN DIAGRAMME D'OBJET 18
FIGURE 15 REPRESENTATION D'UN COMPOSANT 19
FIGURE 16 EXEMPLE DE DIAGRAMME DE COMPOSANTS........ 19
FIGURE 17 EXEMPLE DE DIAGRAMME DE DEPLOIEMENT........ 20
FIGURE 18 EXEMPLE DE DIAGRAMME DE PACKAGE........ 20
FIGURE 19 VUE INTERNE D'UN COMPOSANT........ 21
FIGURE 20 REPRESENTATION D'UN ACTEUR........ 21
FIGURE 21 REPRESENTATION D'UN CAS D'UTILISATION........ 21
FIGURE 22 EXEMPLE D'UN DIAGRAMME DE USES CASE........ 22
FIGURE 23 EXEMPLE D'UN DIAGRAMME D'ACTIVITE 23
FIGURE 24 EXEMPLE D'UN DIAGRAMME D'ETATS-TRANSITIONS........ 24
FIGURE 25 EXEMPLE D'UN DIAGRAMME DE SEQUENCE........ 24
FIGURE 26 EXEMPLE D'UN DIAGRAMME DE COMMUNICATION........ 26
FIGURE 27 EXEMPLE D'UN DIAGRAMME DE TEMPS........ 26
FIGURE 28 EXEMPLE D'UN DIAGRAMME GLOBAL D'INTERACTION........ 27
FIGURE 29 DEPENDANCES ENTRE LES CAS D'UTILISATION ET LES DIFFERENTS MODELES DU PROCESSUS UNIFIE........ 28
FIGURE 30 UN CYCLE AVEC SES PHASES ET SES ITERATIONS 30
FIGURE 31 LES 4 PHASES ET LES 5 WORKFLOW DANS LE PU........ 30
FIGURE 32 DEUX ETATS SIMPLES DANS UN DIAGRAMME D'ETATS-TRANSITIONS 35
FIGURE 33 LES TRANSITIONS........ 37
FIGURE 34 OPTIONS AVANCES D'ETATS-TRANSITION 38
FIGURE 35 SOUS-ETATS SEQUENTIELS 39
FIGURE 36 SOUS-ETATS CONCURRENTS........ 40
FIGURE 37 ETAT HISTORIQUE........ 41
FIGURE 38 EXEMPLE D'UTILISATION DES POINTS DE JONCTION 42
FIGURE 39 EXEMPLE D'UTILISATION DE POINT DE DECISION........ 42
FIGURE 40 TRANSITION COMPLEXE AVEC JOIN ET FORK 43
FIGURE 41 CATEGORISATION DES APPROCHES SEMANTIQUES DU DIAGRAMME D'ETATS-TRANSITION 44

FIGURE 42 SESSION OUVERT DE MAUDE SOUS MS-DOS 71
FIGURE 43 LES DIFFERENTES ETAPES DE LA METHODE DE VERIFICATION DU MODELE UML 81
FIGURE 44 DIAGRAMMES UML UTILISES PAR LA METHODE DE VERIFICATION 83
FIGURE 45 EXEMPLE DE TRANSLATION D'UNE CLASSE AU MAUDE 85
FIGURE 46 EXEMPLE DE TRANSLATION DES ETATS 86
FIGURE 47 EXEMPLE DE TRANSLATION DES EVENEMENTS 87
FIGURE 48 EXEMPLE DE TRANSLATION DES TRANSITIONS 88
FIGURE 49 PROBLEME DE DINER DES PHILOSOPHES 93
FIGURE 50 DIAGRAMME DE CLASSE DE PROBLEME DE DINER DES PHILOSOPHES 94
FIGURE 51 DIAGRAMME D'ETATS-TRANSITIONS DE LA CLASSE PHILOSOPHER...... 94
FIGURE 52 DIAGRAMME D'ETATS-TRANSITIONS DE LA CLASSE FORK 94
FIGURE 53 DIAGRAMME DE COLLABORATION... 95
FIGURE 54 DIAGRAMME DE COLLABORATION PRESENTANT UN DEADLOCK 95
FIGURE 55 SPECIFICATION MAUDE POUR LA CLASSE PHILOSOPHER ET SON DIAGRAMME D'ETATS-TRANSITIONS... 98
FIGURE 56 SPECIFICATION MAUDE QUI REPRESENTE LA CLASSE FORK ET SON DIAGRAMME D'ETATS-TRANSITIONS... 99
FIGURE 57 SPECIFICATION GLOBALE AVEC LA PROPRIETE A VERIFIER 101
FIGURE 58 RESULTAT DE MODEL-CHECKING DE QUELQUES PROPRIETES TEMPORELLES 103
FIGURE 59 RESULTAT DE MODEL-CHECKING DE PROPRIETE D'EXCLUSION MUTUELLE.. 103
FIGURE 60 RESULTAT DE MODEL-CHECKING DE PROPRIETE DE LA PRESENCE DE BLOCAGE (DEADLOCK).. 103
FIGURE 61 RESULTAT DE MODEL-CHECKING DE PROPRIETE D'ABSENCE DE BLOCAGE (DEADLOCK) : UN CONTRE EXEMPLE....................................... 104
FIGURE 62 RESULTAT DU MODEL-CHECKING DE PROPRIETE N° 8 105
FIGURE 63 RESULTAT DU MODEL-CHECKING DE PROPRIETE N° 9 105
FIGURE 64 DIAGRAMME DE CLASSE DU MODELE ATM / BANK..............................106
FIGURE 65 DIAGRAMME D'ETAT-TRANSITION POUR LA CLASSE ATM..................... 107
FIGURE 66 DIAGRAMME D'ETAT-TRANSITION POUR LA CLASSE BANK…………...107
FIGURE 67 COLLABORATION PREVUE ………………………………………………...108
FIGURE 68 COLLABORATION ERRONEE …………………………………………………108
FIGURE 69 SPECIFICATION MAUDE POUR LA CLASSE ATM ET SA MACHINE D'ETAT………………………………………………………………………....109
FIGURE 70 SPECIFICATION MAUDE POUR LA CLASSE BANK ET SA MACHINE D'ETAT………………………………………………………………………...110
FIGURE 71 SPECIFICATION MAUDE DU MODULE A VERIFIER……………………….111
FIGURE 72 RESULTAT DE LA VERIFICATION ……………………………………...112

Introduction générale

Avec l'évolution numérique et informatique, le produit logiciel est devenu de plus en plus complexe et son évolution n'à qu'accroître. Leur développement fait appel à plusieurs techniques et méthodes dont l'objectif est de respecter les délais de développement et de minimiser les coûts.

En effet, depuis la crise de génie logiciel dans les années 60s, plusieurs efforts ont été inventés pour améliorer ces techniques et méthodes de développement. Toujours, dont le but de répondre aux critères imposés par le marché du logiciel. Les méthodologies Orientées objet, les langages de programmation, les outils formelles, les ateliers logiciels, sont des variantes des ces efforts.

L'unification et la standardisation de ces efforts était toujours une ambition établie dans le monde informatique, UML (Unified Modeling Language), le langage unifié de modélisation, est un excellent exemple d'unification des notations utilisées dans les méthodologies d'analyse et de conception orientés objet. Il est devenu depuis quelques années un standard incontournable dans la modélisation des systèmes.

Malgré qu'UML est un langage riche, doté d'une notation ouverte et largement utilisé, les modèles UML restent toujours en besoin d'être vérifiés pour assurer que le comportement spécifié dans ces modèles est correcte, et que ce comportement répond exactement aux besoins fonctionnels du système. Ce fait est dû que UML est un langage graphique et semi-formelle et sa sémantique n'est pas formellement spécifiée.

Dû à ces critiques, UML à reconnu plusieurs extensions et travaux qui visent à donner au langage une sémantique formelle, et de proposer des techniques permettant à ses utilisateurs de vérifier leurs modèles.

Model-checking, ou vérification par modèle, est une technique bien étudiée de vérification formelle et automatique des modèles. Elle permet d'assurer la satisfaction de certaines propriétés vis-à-vis d'un modèle de système. Généralement, les propriétés sont exprimées en logique temporelle et le modèle d'états fini est spécifié avec un formalisme donné. Cette technique est utilisée par certaines approches pour vérifier automatiquement des modèles UML. Néanmoins, ces approches n'arrivent pas à donnée des modèles sémantiques adéquats aux modèles de vérifications qu'ils utilisent pour le model-checking. Pourtant ils utilisent des bons outils de modèle checking tel que SPIN et SMV.

Dans ce travail, nous proposons une approche de vérification des modèles UML en se basant sur une plateforme formelle qui consiste en la logique de réécriture et son langage Maude.

La logique de réécriture est une logique qui permet de spécifier et de programmer des systèmes concurrents et des langages. En effet, elle englobe plusieurs modèles formels qui expriment la concurrence. Elle est implémentée par des langages tels que OBJ3 et Maude.

Maude est un langage formel de spécification et de programmation déclaratif. C'est un langage simple, expressif et performant. Il est classé parmi les meilleurs langages dans le domaine de spécification algébrique et la modélisation des systèmes concurrents.

L'approche que nous proposons, vise à translater les modèles UML à des spécifications formelles exprimées en Maude. Puis utiliser le LTL model-checker de Maude pour vérifier des propriétés par rapport aux spécifications qui représentent formellement les modèles UML. L'approche nous permet aussi d'analyser le comportement spécifié par les modèles UML en utilisant les commandes de réécriture et de recherche de Maude. Elle nous permet aussi de vérifier les collaborations en UML vis-à-vis des diagrammes d'états-transition du modèle.

Dans le premier chapitre, nous allons présenter quelques concepts sur le génie logiciel, nous présentons brièvement le langage UML et en termine le chapitre par les principes de processus unifié qui constitue un cadre général pour modélisé avec UML.

Le deuxième chapitre est consacré au diagramme d'états-transition. Ce dernier est le diagramme le plus visé par les travaux de formalisation, parce qu'il spécifie le comportement interne des objets d'un système. Nous allons commencer le chapitre par une description syntaxique et sémantique du diagramme d'états-transition. En terminant le chapitre par un survol sur les travaux de formalisation de la machine d'états et de leurs catégorisations.

La logique de réécriture est introduite dans le troisième chapitre après la définition des méthodes formelles. Le langage Maude est présenté dans la troisième partie du chapitre. Le chapitre se termine par les concepts de model-checking et la description du LTL model-checker de Maude.

Dans le quatrième chapitre nous présentons l'approche que nous proposons pour vérifier les modèles UML en utilisant la logique de réécriture et son Langage Maude.

Le présent mémoire se termine par une conclusion générale et des perspectives.

CHAPITRE I :

UML & Génie logiciel

Le phénomène de crise de logiciel a été identifié, depuis les années 60s, dans la communauté de génie logiciel. Aujourd'hui après quarante ans, cette crise n'est pas totalement résolue. Ceci est essentiellement dû à la croissance des demandes de logiciel. En effet, l'immense utilisation de logiciel en pratique, la réalisation et la maintenance des logiciels déjà existants font de ces derniers des produits très coûteux.

La production du logiciel fait appel à beaucoup de technologies et de méthodologies dont la majorité n'ont pas atteint le stade de maturité et de stabilité. Ceci fait de cette production un processus dont l'approche est difficile à unifier.

Afin de répondre aux demandes de logiciel, qui sont croissantes, tout en augmentant leur productivité, certains efforts ont été inventés pour améliorer les techniques de développement de logiciel. Les méthodologies d'analyse et de conception dont l'approche orienté objet est la plus importante, les paradigmes de programmation et leurs langages proposés, les environnements du travail et les outils CASE, tous sont des variantes de ces efforts.

L'unification des efforts et des travaux similaires, est bien sûr une ambition toujours établie dans le monde d'informatique. UML (Unified Modeling Language), comme un excellant exemple, est un grand pas vers l'unification des méthodologies d'analyse et de conception orienté objet. Ce langage est devenu récemment, le standard incontournable de la modélisation. Ainsi, il s'infiltre, jour après jour, dans tous les secteurs de développement d'outils et de produits informatiques.

Dans ce chapitre, nous rappelons quelques définitions et quelques concepts sur ; le génie logiciel, les critères de qualité d'un produit informatique, le cycle de vie d'un logiciel et ses modèles, les méthodes d'analyses et de conceptions fonctionnelles et objets. Puis on fait un survol sur le langage unifié de modélisation UML. À la fin du présent chapitre, on expose brièvement le processus unifié qui constitue un cadre général pour développer des logiciels en utilisant la notation UML.

1. Le Génie logiciel

1.1 Définition

Par définition, un logiciel est un ensemble de programmes qui permet à un système informatique, généralement un ordinateur, d'assurer une tâche ou une fonction particulière (exemple : gestion de stock, gestion de paie, comptabilité, réservation aériennes, logiciels d'aide à la décision …etc.).

Le génie logiciel est un domaine de recherche qui à été défini du 7 au 11 octobre 1968, à Garmisch-Partenkirchen, sous le parrainage de l'OTAN [1]. Le génie logiciel à pour objectif d'optimiser le coût de développement du logiciel. L'importance d'une approche méthodologique est apparue à la suite de la crise de l'industrie du logiciel à la fin des années 1970. Cette crise était due à l'augmentation des coûts, le non respect des spécifications, la non fiabilité, le non respect de délai, les difficultés de maintenance et d'évolution …etc.

Pour résoudre tous ces problèmes, le génie logiciel s'intéresse particulièrement à la manière dont le code source du logiciel est spécifié puis réalisé. Ainsi, le génie logiciel touche au cycle de vie de logiciel. Les projets de réalisation des grands logiciels nécessitent des équipes de développement bien structurées. C'est pourquoi la gestion des projets se trouve naturellement liée au génie logiciel.

1.2 Critères de qualité d'un produit logiciel

Le but du développement de logiciel est de produire des logiciels de qualité. Le terme "qualité" est assez large, en général, il signifie que l'on cherche à développer un logiciel qui correspond aux besoins d'utilisateurs de ce logiciel.

La qualité d'un produit logiciel est déterminée, en génie logiciel, par plusieurs facteurs. On trouve parmi ces derniers [1]:

- **La validité** : l'aptitude du logiciel de répondre aux besoins fonctionnels définis par les cahiers de charge et les spécifications. (à remplir exactement les fonctions attendues de ce produit logiciel).
- **La vérifiabilité** : faciliter de préparation des procédures de tests.
- **Efficacité** : l'exploitation optimale de ressources matérielles.
- **Robustesse ou fiabilité** : l'assurance qu'un produit logiciel peut même fonctionner dans des conditions anormales.
- **Extensibilité ou maintenance** : c'est la facilité avec la quelle un logiciel se prête à sa maintenance (une modification ou une extension des fonctions).
- **Réutilisabilité** : aptitude d'un logiciel à être réutilisé, en tout ou en partie, dans de nouvelles applications.
- **Compatibilité** : facilité avec laquelle un logiciel peut être combiné avec d'autres logiciels.
- **Efficacité** : Utilisation optimales des ressources matérielles.

- **Portabilité** : facilité avec laquelle un logiciel peut être transféré sous différents environnements matériels et logiciels.
- **Intégrité et protection des données**: aptitude d'un logiciel à protéger son code et ses données contre des accès non autorisés.
- **Facilité d'emploi** : facilité d'apprentissage, d'utilisation, de préparation des données, d'interprétation des erreurs et de rattrapage en cas d'erreur d'utilisation.

Parfois, ces facteurs sont contradictoires, le choix d'un compromis doit s'effectué selon le cadre du projet.

1.3 Cycle de vie d'un logiciel

Le cycle de vie d'un logiciel (Software life cycle), désigne toutes les étapes du développement d'un logiciel, de sa conception à sa disparition. Il permet de détecter les erreurs au plus tôt et aussi, de maîtriser la qualité du logiciel, les délais de réalisation et les coûts associés. C'est Royce en 1970 le premier qui a proposé un modèle de cycle de vie.

En général, un cycle de vie comprend les étapes suivantes [1] :

- **Analyse et définition des besoins** : Les fonctionnalités du système et les contraintes sont établis d'une manière compréhensible une fois pour toutes par l'équipe de développement en consultant les utilisateurs.
- **Spécification ou Conception Générale** : il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel en partant de l'analyse des besoins. La conception consiste à représenter les diverses fonctions du système d'une manière qui permettra d'obtenir un ou plusieurs programmes réalisant ces fonctions.
- **Conception détaillé** : cette étape consiste à définir précisément chaque sous-ensemble de logiciel.
- **Implémentation (codage ou programmation)** : il s'agit dans cette étape de traduire dans un langage de programmation les fonctionnalités définies lors de la phase de conception.
- **Tests unitaires** : Lors de l'étape précédente, on réalise un ensemble d'unités de programmes écrites dans un langage de programmation exécutable. Les testes unitaires permettent de vérifier que ces variantes répondent à leurs spécifications.
- **Intégration** : l'objectif de cette étape est de d'assurer de l'interfaçage de différents éléments (modules) du logiciel. Elle fait l'objet de tests d'intégration consignés dans un document.
- **Qualification** : elle vise de vérifier la conformité du logiciel aux spécifications initiales.
- **Documentation** : elle vise à produire les informations nécessaires pour l'utilisation du logiciel et pour les développeurs ultérieurs.
- **Maintenance** : Normalement (Mais pas nécessairement) ceci est la plus longue étape du cycle de vie. L'activité de maintenance consiste à corriger les erreurs qui n'ont pas été découvertes lors des étapes antérieures du cycle de vie, à améliorer la réalisation des

unités du système et à augmenter ses fonctionnalités au fur et à mesure que de nombreux besoins apparaissent.

Les étapes sont distinctes du point de vue gestion. Mais en pratique, ces étapes de développement se recouvrent et provoquent des relations d'informations. Leur présence et séquence dépend du modèle de cycle de vie choisi. Le cycle de vie permet de prendre en compte non seulement les aspects techniques, mais plutôt l'organisation et les aspects humains.

1.4 Modèle de cycle de vie d'un produit logiciel

On trouve généralement des modèles linéaires : en cascade, en V, …etc. et des modèles non linéaires : itératif, en spiral, incrémental, …etc.

1.4.1 Modèle de cycle de vie en cascade

Le modèle en cascade (waterfall) est développé par Royce en 1970. Les phases de développement sont effectuées les unes après les autres (fig.1). Chaque phase doit être approuvée avant de pouvoir commencer l'autre. Le modèle original ne comportait pas de possibilité de retour en arrière. Celle-ci a été rajoutée ultérieurement sur la base qu'une étape ne remet en cause que l'étape précédente, ce qui, dans la pratique, s'avère insuffisant [1].

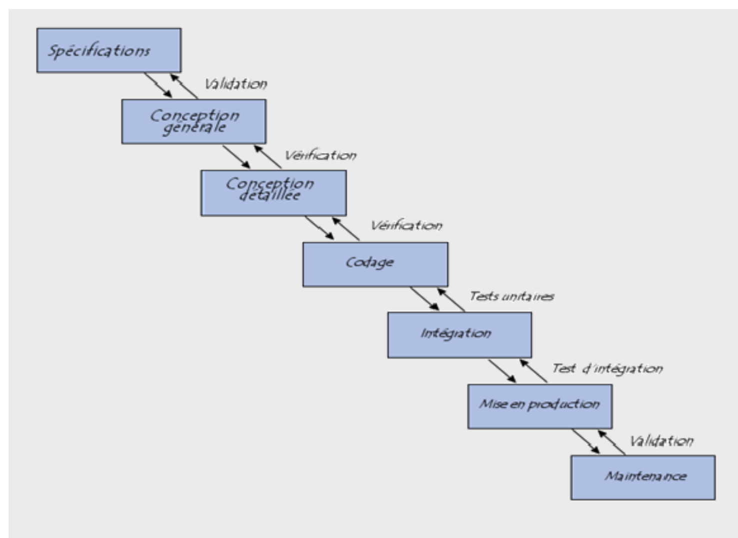


Fig. 1 : Modèle en cascade [1]

1.4.2 Modèle de cycle de vie en V

Ce modèle de cycle de vie est développé par MC Dermid et Ripkin en 1984. Ce modèle est une amélioration du modèle en cascade, chaque phase du projet à une phase de test qui lui est associé. Les phases de la partie montante, doivent renvoyer de l'information sur les phases en vis-à-vis lorsque des défauts sont détectés afin d'améliorer le logiciel (fig. 2).

Le modèle en V met en évidence la nécessité d'anticiper et de préparer dans les étapes descendantes les "attendus " des futures étapes montantes : ainsi les attendues des testes de

validation sont définis lors des spécifications, les attendus des tests unitaires sont définis lors de la conception [1].

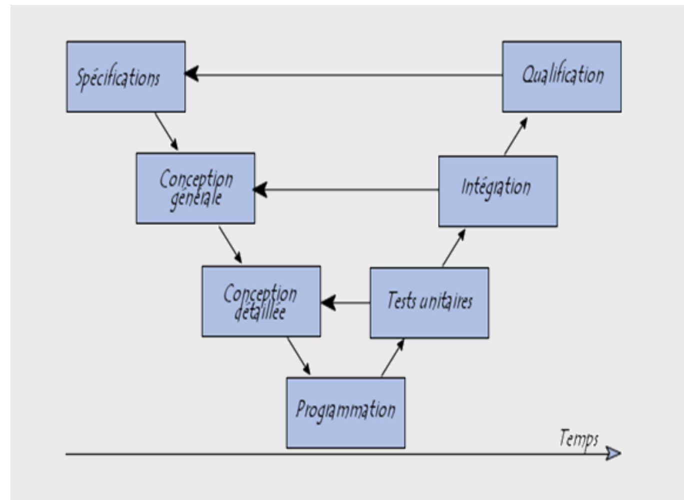


Fig. 2 : Modèle en V [1]

1.4.3 Modèle de cycle de vie en spiral

Proposé par B. Boehm en 1988, ce modèle est beaucoup plus général que le précédent. Il met l'accent sur l'activité d'analyse des risques : chaque cycle de la spirale se déroule en quatre phases (fig.3) [2]:

1. détermination, à partir des résultats des cycles précédents, ou de l'analyse préliminaire des besoins, des objectifs du cycle, des alternatives pour les atteindre et des contraintes ;
2. analyse des risques, évaluation des alternatives et, éventuellement maquettage ;
3. développement et vérification de la solution retenue, un modèle « classique » (cascade ou en V) peut être utilisé ici ;
4. revue des résultats et vérification du cycle suivant.

L'analyse préliminaire est affinée au cours des premiers cycles. Le modèle utilise des maquettes exploratoires pour guider la phase de conception du cycle suivant. Le dernier cycle se termine par un processus de développement classique.

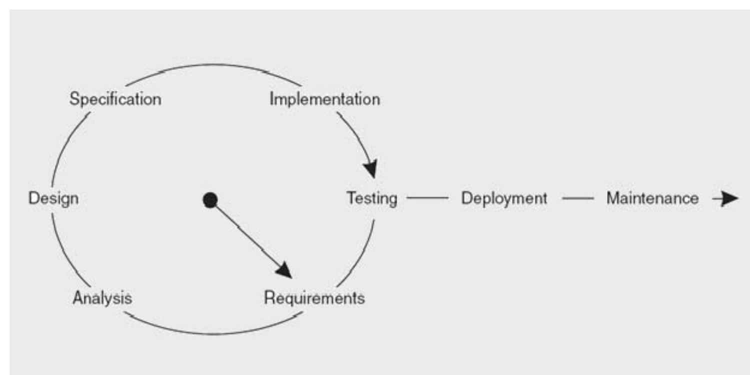


Fig. 3 : Développement en spiral [2]

1.4.4 Modèle de cycle de vie par Incréments

Dans les modèles spirale, V ou cascade, les composants sont développés indépendamment les uns des autres. Dans le modèle par incréments (fig.4), seul un sous ensemble est développé à la fois, d'abord un logiciel noyau, puis successivement, les incréments sont développés et intégrés. On débute par définir les exigences et on les décompose en sous-systèmes. À chaque version du logiciel, de nouvelles fonctionnalités venant comblées les exigences sont ajoutées. On continue de la sorte jusqu'à ce que toutes les fonctionnalités demandées soient comblées par le système. Un incrément est une version du système : une augmentation apportée à la construction en cours d'un système [2].

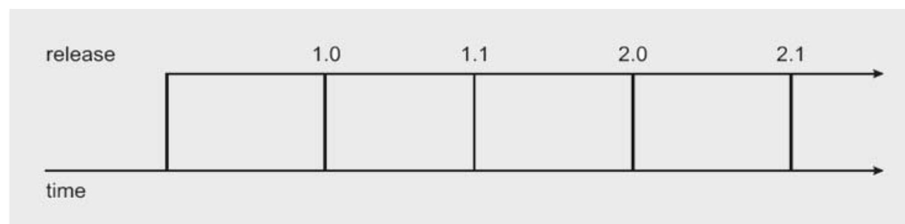


Fig. 4 : Modèle par incréments [2]

1.5 Méthodes d'analyse et de conception

La conception consiste, dans le processus de développement, la phase la plus importante, car un système bien conçu est facile à réaliser et à maintenir, facile à comprendre et fiable. Par contre, un système mal conçu peut fonctionner correctement mais il sera souvent coûteux à maintenir, difficile à tester et peu fiable. Pour accomplir les tâches de cette phase avec une conception efficace et facile à élaborer, il est commode d'utiliser une des méthodes d'analyse et de conception.

1.5.1 Méthodes fonctionnelle descendante :

Le système est conçu d'un point de vue fonctionnel, en commençant au niveau le plus général. Et en descendant progressivement vers la conception détaillée (fig.5). Ces méthodes sont appelées : méthodes de conception structurée ou conception par raffinement successif.

Ces méthodes utilisent intensivement les raffinements successifs pour produire des spécifications sous forme de diagrammes de flots de données. Le plus haut niveau représente l'ensemble du problème (sous forme d'activité, de données ou de processus, selon la méthode). Chaque niveau est ensuite décomposé en respectant les entrées/sorties du niveau supérieur. La décomposition se poursuit jusqu'à arriver à des composants maîtrisables.[1]

La SADT (Structured Analysis Design Technique) est probablement la méthode d'analyse fonctionnelle et de gestion de projets la plus connue. Elle permet non seulement de décrire les tâches du projet et leurs interactions, mais aussi de décrire le système que le projet vise à étudier, créer ou modifier, en mettant notamment en évidence les parties qui constituent le système, la finalité et le fonctionnement de chacune, ainsi que les interfaces entre ces diverses

parties. Le système ainsi modélisé n'est pas une simple collection d'éléments indépendants, mais une organisation structurée de ceux-ci dans une finalité précise.

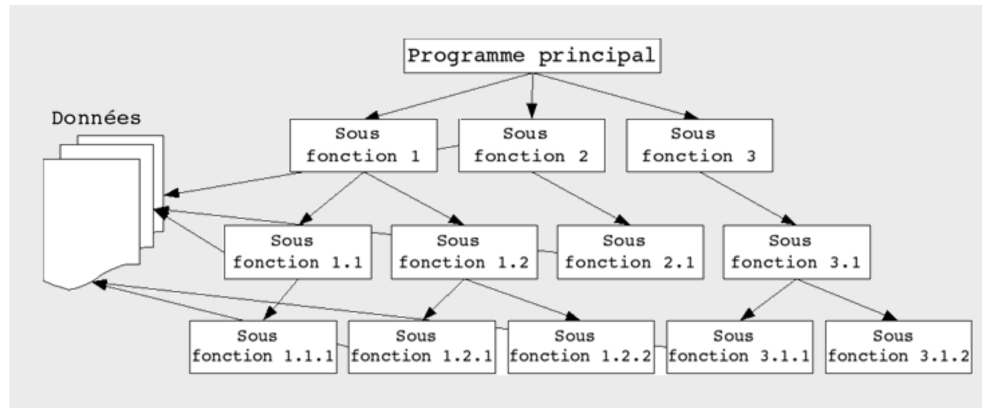


Fig. 5 : Représentation graphique d'une approche fonctionnelle [1]

1.5.2 Méthodes orientées objet

Le système est vu comme une collection d'objets communiquant entre eux par des messages. À chaque objet on associe, un ensemble d'opérations. Elle est basée sur l'idée de masquage de l'information (encapsulation des données).

La difficulté de ces méthodes consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une existence matérielle ou virtuelle.

La Conception Orientée Objet (COO) est la méthode qui conduit à des architectures logicielles fondées sur les objets du système, plutôt que sur la fonction qu'il est censé réaliser. Dans la section suivante on présente l'approche objet avec plus de détailles.

2. L'approche Objet

L'approche objet est basée sur un concept unique unifiant les traitements et les données. Ce concept est l'objet. Elle considère le logiciel comme une collection d'objets dissociés définis par des propriétés, qui sont soit des attributs, soit des opérations [3].

L'approche est simple grâce à son modèle qui fait appel seulement aux cinq concepts fondateurs (les objets, les messages, les classes, la généralisation et la surcharge) [4].

2.1 L'Objet

L'objet est définit comme une entité atomique constituée d'un état, d'un comportement et d'une identité [3];

- ***L'état*** **:** C'est la situation instantanée dans laquelle se trouve l'objet. En d'autre terme, c'est l'ensemble des valeurs d'attributs dans un moment donné.

- ***Le comportement*:** C'est le groupe de toutes les compétences de l'objet. Il décrit les actions et les réactions de cet objet. Chaque atome de comportement est appelé opération, représenté par une méthode. Ces opérations sont déclenchées suit à une réception d'un message envoyé par un autre objet et peut provoquées des envoies de messages vers autres objets.
- ***L'identité* :** C'est la caractéristique fondamentale de l'objet qui le distingue de tous les autres objets.

2.2 Les Classes

Une classe est une description d'une famille d'objets ayant la même structure et le même comportement. Chaque classe possède une double composante [4].

- ***Une Composante Statique*:** C'est les données qui sont des champs nommés possédant des valeurs, caractérisant l'état d'un objet instancié de cette classe.
- ***Une Composante Dynamique***: c'est les procédures appelées méthodes qui spécifient et représentent le comportement commun des objets appartenant à la classe. Les méthodes manipulent les champs (attributs) des objets et caractérisent les actions qui peuvent être effectué par l'objet.

2.3 L'instanciation

La classe est l'entité conceptuelle qui décrit l'objet. Sa définition sert de modèle pour construire ses représentants physiques appelés instances [4].

- ***Une instance*** : c'est donc un objet particulier qui est crée en respectant les plans de constructions données par sa classe. Celle-ci joue le rôle de moule permettent de produire autant d'exemplaires que nécessaire. On parle dans ce cas d'instanciation.

2.4 Les Relations entre associations

Les liens particuliers qui relient les objets peuvent être vus de manière abstraite dans le monde des classes : à chaque famille de lien entre objets correspond une relation entre les classes de ces mêmes objets. Comme les objets sont les instances de classes, les liens entre objets sont des instances des relations [3].

- ***L'association*** : Une relation qui exprime une connexion sémantique bidirectionnelle entre classes, qui décrit un ensemble de lien, et pour l'amélioration de la lisibilité, il est possible que chaque extrémité d'une association puisse porter une information de multiplicité qui précise le nombre d'instances qui participent à la relation.
- ***L'agrégation*** : C'est une relation qui exprime une forme de couplage entre abstraction. La force de ce couplage dépend de la nature de la relation dans le domaine du problème. Ou avec d'autres termes, C'est une forme d'association non symétrique qui exprime un couplage fort et une relation de subordination.

- ***La composition*****:** C'est une Forme d'agrégation avec un couplage plus important, Ce couplage de composition indique que les composants ne sont pas partageables et que la destruction de l'agrégat entraîne la destruction des composants agrégés.

2.5 Les Messages & la Synchronisation :

- ***Le message*****:** C'est l'élément de communication entre objet qui déclenche une activité dans l'objet destinataire. Il permet l'interaction de manière flexible et dynamique entre ces objets.
- ***La synchronisation*** : C'est l'expression de la forme qui décrit la nature des mécanismes de communication qui permettent la transmission de message d'un objet vers un autre. Cette notion précise, ainsi, la nature de la communication et les règles qui régissent le passage des messages. En conséquence, il existe quatre types de messages [3]:
    - *Message Synchrone :* C'est un message dont la forme de communication est bloquante, avec accusé de réception implicite.
    - *Message Asynchrone :* C'est un message dont la forme de communication est non bloquante et sans accusé de réception.
    - *Message Dérobant* : C'est un message qui déclenche une opération seulement si le destinataire s'est préalablement mis en attente du message.
    - *Message Minuté :* Un message minuté bloque l'expéditeur pendant un temps donné, en attendant la prise en compte de l'envoi par le destinataire. L'expéditeur est libéré si la prise en compte n'a pas eu lieu au bout du temps spécifie dans la description de l'envoie du message.

2.6 Généralisation et Spécialisations

La généralisation et la spécialisation sont des points de vue portés sur les hiérarchies des classes. Elles expriment dans quel sens une hiérarchie de classe est exploitée [4].

- ***La généralisatio*****n :** La généralisation est un point de vue ascendant porté sur une classification. Elle consiste à factorisé les éléments communs (attributs, opération et contraintes) d'un ensemble de classe dans une classe plus général appelée superclasse.
  Les classes sont ordonnées selon une hiérarchie; une superclasse est une abstraction de ses sous-classes. La généralisation est une démarche assez difficile car elle demande une bonne capacité d'abstraction. La mise au point d'une hiérarchie optimale est délicate et itérative.
- ***La spécialisation*** **:** La spécialisation est un point de vue descendant porté sur une classification. Elle permet de capturer les particularités d'un ensemble d'objets non discriminés par les classes déjà identifiées. Les nouvelles caractéristiques sont représentées par une nouvelle classe, sous-classe d'une des classes existantes. La spécialisation est une technique efficace pour l'extension cohérente d'un ensemble de classes.
- ***L'héritage*** **:** L'héritage permet un *partage hiérarchique* des propriétés (attributs et opérations). Une sous-classe (ou *classe fille*) peut incorporer, ou *hériter*, des propriétés d'une

superclasse (ou *classe mère*). Généralement une superclasse définit les grands traits d'une abstraction, une sous-classe hérite de cette définition et peut la modifier, raffiner et/ou rajouter ses propres propriétés. Il existe deux types d'héritage simple et multiple, contrairement au premier type le deuxième peut hériter de plusieurs classes mères.

3. Unified Modeling Language

Quand les méthodes d'analyse et de conception orientées objet ont atteints une certaine maturité. UML, le langage unifié de la modélisation, s'est dégagé pour devenir le standard de modélisation objet. En effet, UML n'est pas une méthode mais plutôt une notation qui fusionne les notations d'OOD, OMT, OOSE, et d'autres. La démarche d'unification entreprise en octobre 1995 a commencé par une harmonisation des méthodes OMT et OOD, avec l'élaboration d'un méta-modèle commun. L'apport de la méthode OOSE a eu lieu au cours de l'année 1996, lorsque Ivar Jacobson a lui aussi intégré l'équipe initiale composée de Gardy Booch et James Rumbaugh [4].

Des outils et des ateliers de conception avaient anticipés en proposant des diagrammes de représentation des cas d'utilisation (USE CASES). Désormais, l'emploi des cas d'utilisation est officialisé dans UML. Les versions mises à la disposition du public ont été les versions 0.8, 0.9, 0.91 et finalement la version 1.0 du 13 janvier 1997. C'est cette version qui a été remise à l'association OMG (Object Management Group) en vue d'établir un standard industriel d'un langage de modélisation des modèles Objet [3]. L'OMG adopte en novembre 1997 UML 1.1 comme langage de modélisation des systèmes d'information à objets. La version d'UML en cours à la fin 2006 est UML 2.0 et des transformations continuelles ne cessent d'être effectuées dans ce langage pour supprimer les incohérences, apporter des améliorations et ajouter de nouveaux concepts.

UML est donc non seulement un outil intéressant mais une norme qui s'impose en technologie à objets et à laquelle se sont rangés tous les grands acteurs du domaine, acteurs qui ont d'ailleurs contribué à son élaboration.

3.1 Définition

UML (Unified Modeling Language), peut se traduire par le langage de modélisation unifié, il se définit comme un langage de modélisation graphique et textuel destiné à comprendre et à décrire des besoins, spécifier et documenter des systèmes, esquisser des architectures logicielles, concevoir des solutions et communiquer des points de vues [4].

UML unifie à la fois les notations et les concepts orientés objets. Il ne s'agit pas d'une simple notation, mais les concepts transmis par un diagramme ont une sémantique précise et sont porteurs de sens au même titre que les mots d'un langage. UML unifie également les notations nécessaires aux différentes activités d'un processus de développement et offre, par ce biais, le moyen d'établir le suivi des décisions prises, depuis la spécification jusqu'au codage. [5].

UML a une dimension symbolique et ouvre une nouvelle voie d'échange de visions systématiques précises.

UML se compose des vues, de modèles d'éléments, des mécanismes généraux et des diagrammes. Les sections suivantes décrivent brièvement ces ensembles.

3.2 Les vues d'UML

Les vues présentent les différents aspects d'un système. Une vue n'est pas un élément graphique ou un diagramme, mais une abstraction qui englobe un nombre de diagrammes afin de décrire le système d'un point de vue donné [6].

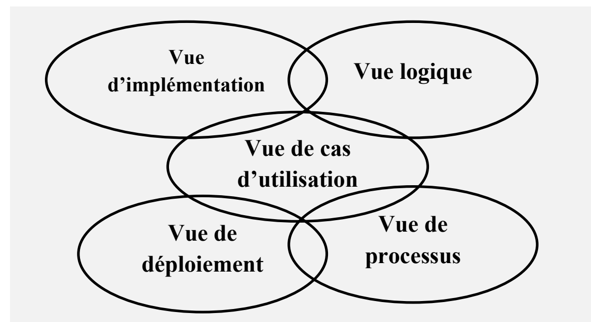


Fig.6 : Les vues d'UML [6]

Parmi ces vues on trouve :

a) ***Vue des cas d'utilisation*** : Cette vue définit les besoins des clients du système et centre la définition de l'architecture du système sur la satisfaction (la réalisation) de ces besoins à l'aide de scénarios et de cas d'utilisation, cette vue conduit à la définition d'un modèle d'architecture pertinent et cohérent [4]. Donc c'est une description du système "vue" par les acteurs du système. Elle correspond, ainsi, aux besoins attendus par chaque acteur (c'est le QUOI et le QUI).

b) ***Vue logique*** **:** cette vue décrit, d'une façon abstraite, comment réaliser les fonctionnalités d'un système. Elle est dédiée aux concepteurs et aux développeurs. Contrairement à la vue des cas d'utilisation, la vue logique est une vue à l'intérieur du système. Elle décrive à la fois, la structure statique (classes, objets, relations …), et les collaborations qui avoir lieu durant l'échange de messages entre objets afin de réaliser des fonctions précises. Des propriétés tel que la persistance, la concurrence, sont aussi définis dans cette vue. La structure statique est décrite dans cette vue par les diagrammes de classes et d'objets. La dynamique est spécifiée par les machines d'états, les diagrammes d'activité et d'interactions [6].

c) ***Vue d'implémentation*** : Cette vue décrite les principaux modules et leurs dépendances dans un système. Elle est propre aux développeurs. Elle montre : L'allocation des éléments de modélisation dans des modules (fichiers sources, bibliothèques dynamiques, bases de

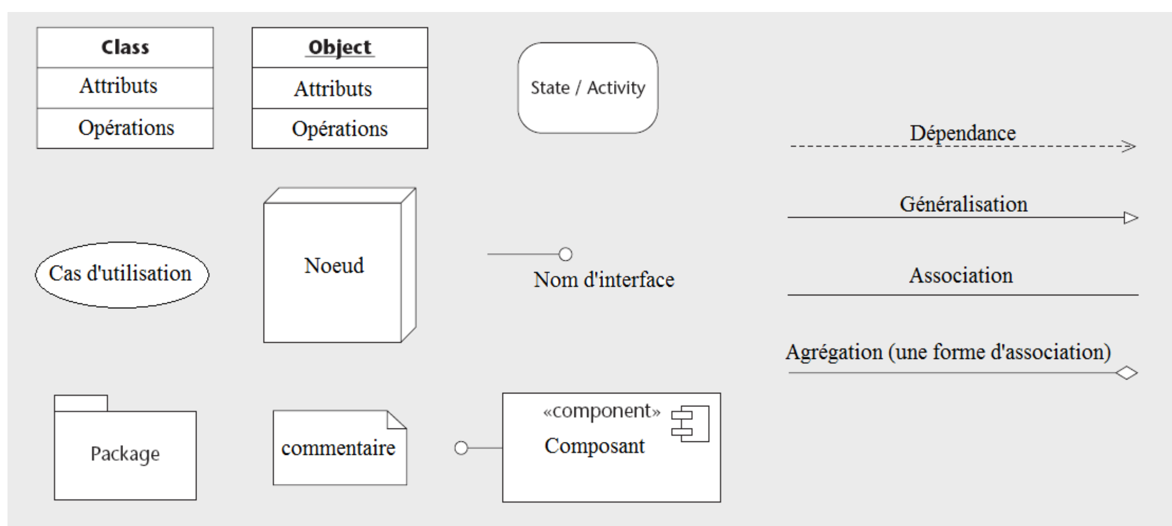
données, exécutables, ….etc.). En d'autres termes, cette vue identifie les modules qui réalisent (physiquement) les classes de la vue logique. L'organisation des composants, c'est-à dire la distribution du code en gestion de configuration, les dépendances entre les composants... Les contraintes de développement (bibliothèques externes...). La vue des composants montre aussi l'organisation des modules en ″sous-systèmes″, les interfaces des sous-systèmes et leurs dépendances (avec d'autres sous-systèmes ou modules) [4].

d) ***Vue des processus*** : C'est la vue temporelle et technique, qui met en œuvre les notions de tâches concurrentes, stimuli, contrôle, synchronisation, etc. elle est très importante dans les environnements multitâches; elle montre : La décomposition du système en terme de processus (tâches); les interactions entre les processus (leur communication); la synchronisation et la communication des activités parallèles (threads) [6].

e) ***Vue de déploiement*** : Cette vue décrit la position géographique et l'architecture physique de chaque élément du système. Elle décrite aussi les ressources matérielles et la répartition du logiciel dans ces ressources : la disposition et nature physique des matériels, ainsi que leurs performances, l'implantation des modules principaux sur les nœuds du réseau, les exigences en terme de performances (temps de réponse, tolérance aux fautes et pannes...). Cette vue est représentée par des diagrammes de déploiement est utilisée par les développeurs, les responsables d'intégration et des testes [4].

3.3 Les éléments du modèle

Les éléments du modèle sont les briques utilisés dans les différents diagrammes. Ce sont des éléments graphiques qui représentent des concepts orientés objets communs, tel que les classes, les objets, les messages, les associations, les généralisations, les cas d'utilisations, les états …etc. un modèle d'élément est utilisé dans différents diagramme, et toujours avec la même sens et le même symbole. La figure (Fig.7) montre quelques exemples d'éléments du modèle.



3.4 Mécanismes généraux

UML définit un petit nombre de mécanismes communs qui assurent l'intégrité conceptuelle de la notation. Ces mécanismes communs comprennent les commentaires, les mécanismes d'extension (stéréotype, contrainte et valeur marquées), etc. [4].

a) ***Les stéréotypes*** **:** Le concept de stéréotype, qui est une nouveauté apparue dans UML, peut être associé à tout élément du modèle, à savoir les classes, les associations, les opérations, les attributs, les packages, les dépendances, etc. Un stéréotype permet de classer les éléments du modèle en grandes familles [7]. Il permet la méta-classification d'un élément d'UML. Il est utilisé aussi pour définir une utilisation particulière d'éléments de modélisation existants ou pour modifier la signification d'un élément. L'élément stéréotype et son parent non stéréotype ont une structure identique mais des sémantiques différentes [3].

b) ***Les valeurs marquées*** **:** Une valeur marquée est une paire **(nom, valeur)** qui ajout une nouvelle propriété à un élément de modélisation. Cette propriété peut représenter une information d'administration (auteur, date de modification, état, version…), de génération de code ou une information sémantique utilisée par un stéréotype. En général, la propriété ajoutée est une information annexe, utilisée par les concepteurs ou les outils de modélisation. La spécification d'une valeur marquée prend la forme : **nom = valeur.** Une valeur marquée est indiquée entre accolades [4].

c) ***Les types primitifs*** **:** Le standard UML définit un certain nombre de types primitifs, utilisés pour la modélisation d'UML dans le méta-modèle.

   Les types suivants sont des types primitifs du méta-modèle UML [4] :

   - **Booléen :** Un booléen est un type énuméré qui comprend les deux valeurs Vrai et faux.
   - **Entier :** Dans le méta-modèle, un entier est un élément compris dans l'ensemble infini des entiers {…,-2,-1, 0, 1, 2,…}.
   - **Expression :** Une expression est une chaîne de caractères qui définit une déclaration à évaluer.
   - **Nom :** un nom est une chaîne de caractères qui permet de désigner un élément.
   - **Chaîne :** Est une suite de caractères, désigner par un nom.
   - **Temps :** Un temps est une valeur qui représente un temps absolu ou relatif. Un temps possède une représentation sous forme de chaîne.

d) ***Type en Extension (PowerType)*** **:** Lorsqu'on modélise une hiérarchie de classes suivant la relation d'héritage, on organise des modèles permettant de construire des instances à partir de ces modèles. Il peut être commode également de considérer une méta-classe dont les instances sont des sous-classes d'une classe particulière. Cette méta-classe est une définition d'une classe en extension, notée par le stéréotype "powertype" [3].

3.5 Les diagrammes

Un diagramme donne à l'utilisateur un moyen de visualiser et de manipuler des éléments de modélisation. UML définit des diagrammes structurels et comportementaux pour représenter respectivement des vues statiques et dynamiques d'un système [4]. Les diagrammes incluent des éléments graphiques qui décrivent le contenu des vues [6].

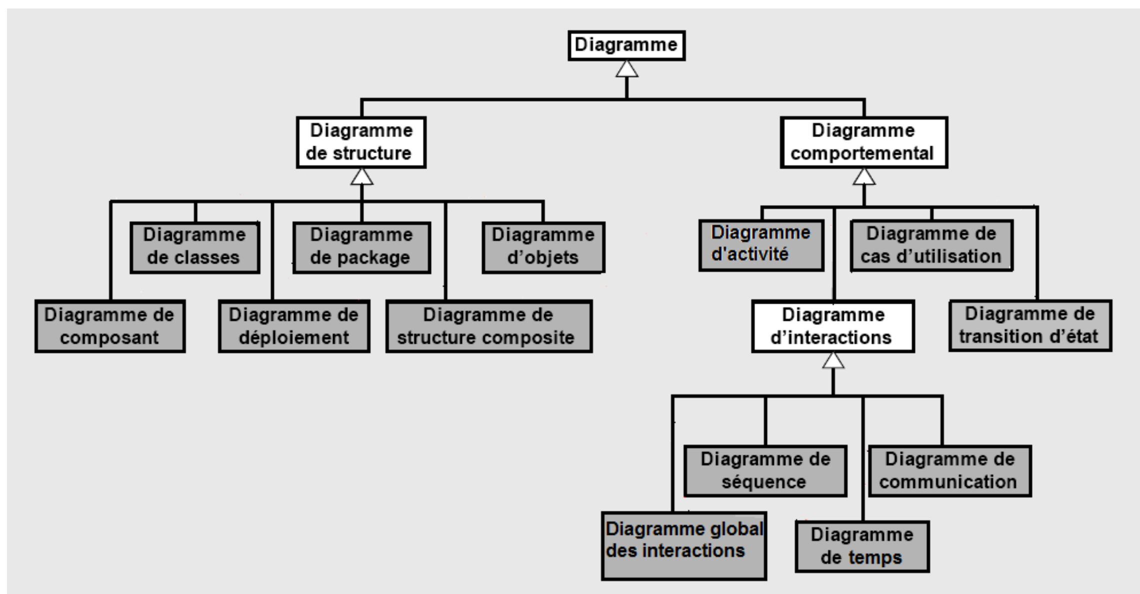


Fig.8 : les différents diagrammes d'UML 2.0 [1]

UML, dans sa version 2.0, comporte plus de treize diagrammes qui peuvent êtres combinés pour définir toutes les vues d'un système. Ils se répartissent en trois grands groupes :

- ***Diagrammes* structurels :** ou diagrammes statiques (UML structure)
  - Diagramme de classes (class diagram)
  - Diagramme d'objets (object diagram)
  - Diagramme de composants (component diagram)
  - Diagramme de déploiement (Deployment diagram)
  - Diagramme de paquetages (Package diagram)
  - Diagramme de structures composites (Composite structure diagram)

- ***Diagrammes* comportementaux** : ou diagrammes dynamiques (UML Behavior)
  - Diagramme de cas d'utilisation (Use case diagram)
  - Diagramme d'activités (Activity diagram)
  - Diagramme d'état-transition (State machine diagram)
- ***Diagrammes* d'interactions (Interaction diagram)**
  - Diagramme de Séquence (Sequence diagram)
  - Diagramme de communication (Communication diagram)
  - Diagramme global d'interaction (Interaction overview diagram)
  - Diagramme de temps (Timing diagram)

Ces diagrammes, d'une utilité variable selon le cas, ne sont pas nécessairement tous produits durant la modélisation du système.

3.5.1 Diagrammes structurels

a) Diagramme de classes (Class diagram)

Les diagrammes de classes expriment de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes. Outre les classes, ils représentent un ensemble d'interactions et de paquetages, ainsi que leurs relations [4].

Les classes sont les descripteurs d'un ensemble d'objets qui ont une structure, un comportement et des relations similaires. Les classes sont représentées par des rectangles compartimentés (Fig.9). Les classes sont reliées l'une à l'autre avec plusieurs formes : les associations (une classe associé une autre classe), la dépendance (une classe dépend ou utilise une autre classe), la spécialisation (une classe est une spécialisation d'un autre classe) …etc [6]. La figure (Fig.10) montre quelques exemples d'éléments qui représentent des relations.

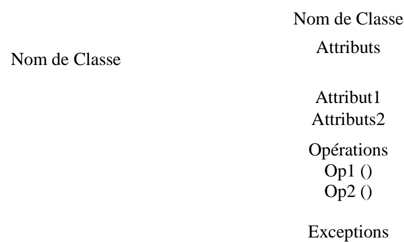


Fig.9 : Représentation graphique des classes. **Fig.10 : exemple de quelques relations**

L'exemple de la figure (Fig.11) présente un diagramme de classe qui montre la classe auteur et les classe ordinateurs que l'auteur peut utilisés.

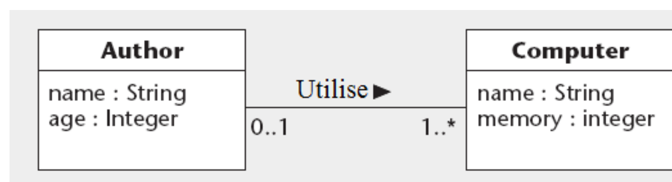


Fig.11 : exemple d'un diagramme de classe [6]

b) Diagramme d'objet (Object diagram)

Les Diagrammes d'objets, ou diagrammes d'instances, montrent des objets et des liens. Ce diagramme est une instance d'un diagramme de classe qui montre l'état du système modélisé à un

instant donné. La notation retenue pour les diagrammes d'objets est ainsi dérivée de celle des diagrammes de classe; les éléments qui sont des instances sont soulignés.

Les diagrammes d'objets s'utilisent principalement pour montrer un contexte, par exemple avant ou après une interaction, mais également pour faciliter la compréhension des structures de données complexes, récursives par exemple. [4]

Les figures (Fig.12, Fig.13) Montrent respectivement la représentation d'un objet et un groupe d'objets.

Nom de l'objet Nom de l'objet : Classe : Classe

Fig.12 : Représentations graphiques d'un objet.

: Personne

Fig.13 : Représentation d'un groupe d'objets

L'exemple présenté dans la figure (Fig.14) montre un diagramme d'objet (instance du diagramme de classe présenté dans la figure (Fig.11)

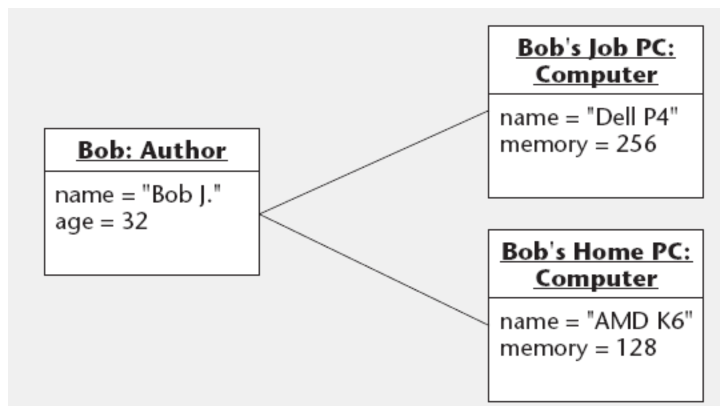


Fig.14 : Exemple d'un diagramme d'objet

c) Diagramme de composant (component diagram)

Les diagrammes de composants décrivent les composants et leurs dépendances dans l'environnement de réalisation. Les diagrammes de composants sont des vues statiques de l'implémentation des systèmes qui montrent les choix de réalisation. En général ils ne sont utilisés que pour des systèmes complexes [4].

Un composant est un élément physique qui représente une partie implémentée d'un système. Un composant peut être du code (source, binaire ou exécutable), un script, un fichier de commande, un fichier de données, une table, etc. il peut réaliser un ensemble d'interfaces qui définissent alors le comportement offert à d'autres composants. Les services sont implémentés

par les éléments du composant. En outre, chaque composant peut posséder des attributs et d'opérations. Un composant est éventuellement connecté à d'autres composants (via des dépendances ou de compositions) [4].

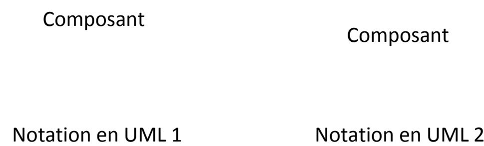


Fig.15 : Représentation d'un composant [8]

La figure (Fig.16) montre un exemple de diagramme de composants.

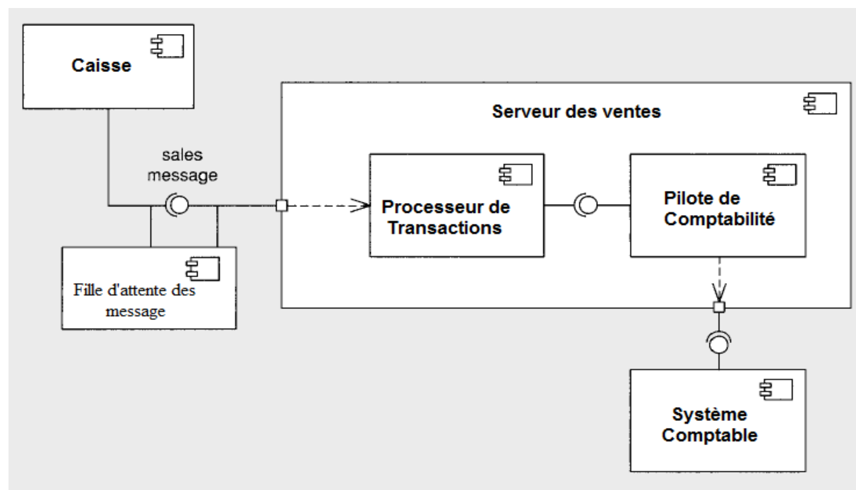


Fig.16 : Exemple de diagramme de composants [8]

d) Diagramme de déploiement (deployement diagram)

Un diagramme de déploiement est un graphe composé de nœuds interconnectés par des liens de communication. Le diagramme de déploiement montre la disposition physique des différents matériels « Les Nœuds » qui entrent dans la composition d'un système et la répartition des instances de composants, processus et objets qui <<vivent>> sur ces matériels [4].

Un diagramme de déploiement permet de donner la structure d'une plate-forme technique, mais aussi de spécifier la localisation des nœuds constitués par des unités distribuées, de préciser où se trouvent les processus et de montrer comment les objets se créent et se déplacent dans une architecture distribué [3]. La figure (Fig.17) illustre un exemple de diagramme de déploiement.

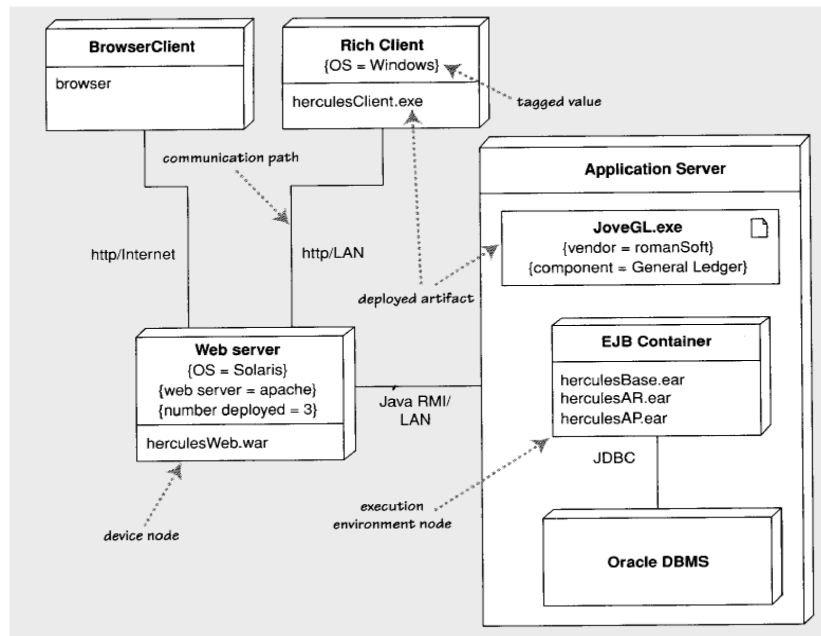


Fig.17 : exemple de diagramme de déploiement [1]

e) Diagramme de Paquetage

En UML, un diagramme de paquetages décrit comment un système est organiser en groupes logique appelé paquetages tout en montrant les dépendances entres ces paquetages. En effet, Les packages (paquetages en français) permettent de regrouper et d'isoler des classes, des associations, et éventuellement d'autre packages. Un package regroupe le plus souvent un ensemble d'entités qui correspondent à une fonctionnalité bien définie. Bien souvent, c'est cette fonctionnalité qui définira le nom du package [7].

Un package n'introduit pas de sémantique particulière. Le système à concevoir est représenté par un package racine. La figure (Fig.18) montre un exemple de diagramme de packages

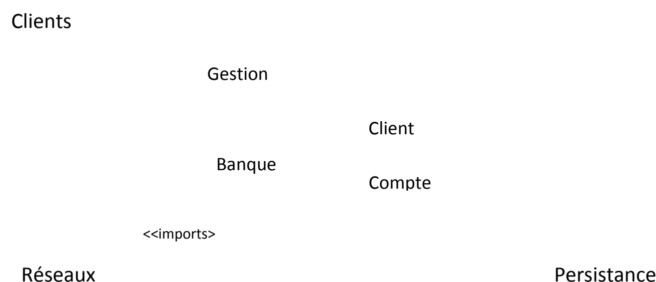


Fig.18 : exemple de diagramme de package [4]

f) Diagramme de structures composites

C'est un nouveau diagramme introduit-en UML2. Il permet de présenter la décomposition hiérarchique d'un objet, un cas d'utilisation, une collaboration, une activité ou une classe, en un ensemble de structures internes. Ces structures internes sont des ensembles d'éléments interconnectés avec leurs relations et leurs modes de communications [8]. Ce diagramme permet de réduire la complexité des objets en donnant une vue détaillée sur l'architecture interne de ces objets durant l'exécution du système [9]. La figure (Fig.19) Montre un exemple sur un diagramme de structures composites qui représente la vue interne d'un composant.

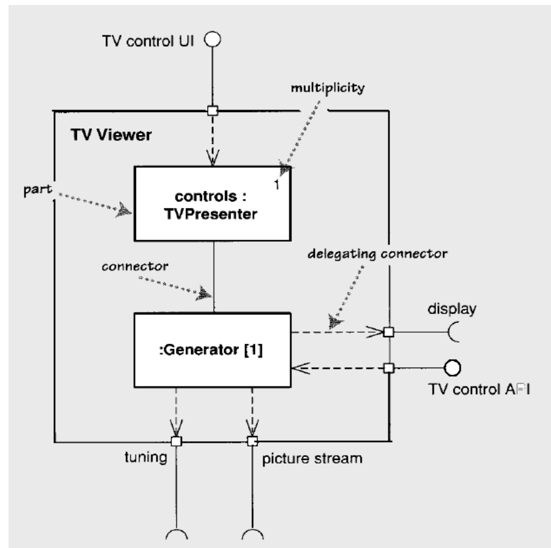


Fig.19 : Vue interne d'un composant [8]

3.5.2 Les diagrammes comportementaux

a) Diagramme de cas d'utilisation (Use case diagram)

Les cas d'utilisations (use cases) constituent le concept principal de la méthode OOSE (Object Oriented Software Engineering) d'Ivar Jakobson, un des pères d'UML. Un diagramme de cas d'utilisation capture le comportement d'un système, d'un sous-système, d'une classe ou d'un composant tel qu'un utilisateur extérieur le voit. Il scinde la fonctionnalité du système en unités cohérentes, les cas d'utilisation, ayant un sens pour les acteurs. Les cas d'utilisation permettent d'exprimer le besoin des utilisateurs d'un système, ils sont donc une vision orientée utilisateur de ce besoin au contraire d'une vision informatique.

Le diagramme de cas d'utilisation se compose d'acteurs (voir Fig.20) un cas d'utilisation (voir Fig.21). Les traits entre les acteurs et les cas d'utilisation représentent les interactions.

<<Actor>>
Client

<<Actor>>

« stéréotype»
Nom du Cas

Le diagramme de la figure (Fig.22) montre un exemple de diagramme de cas d'utilisation.

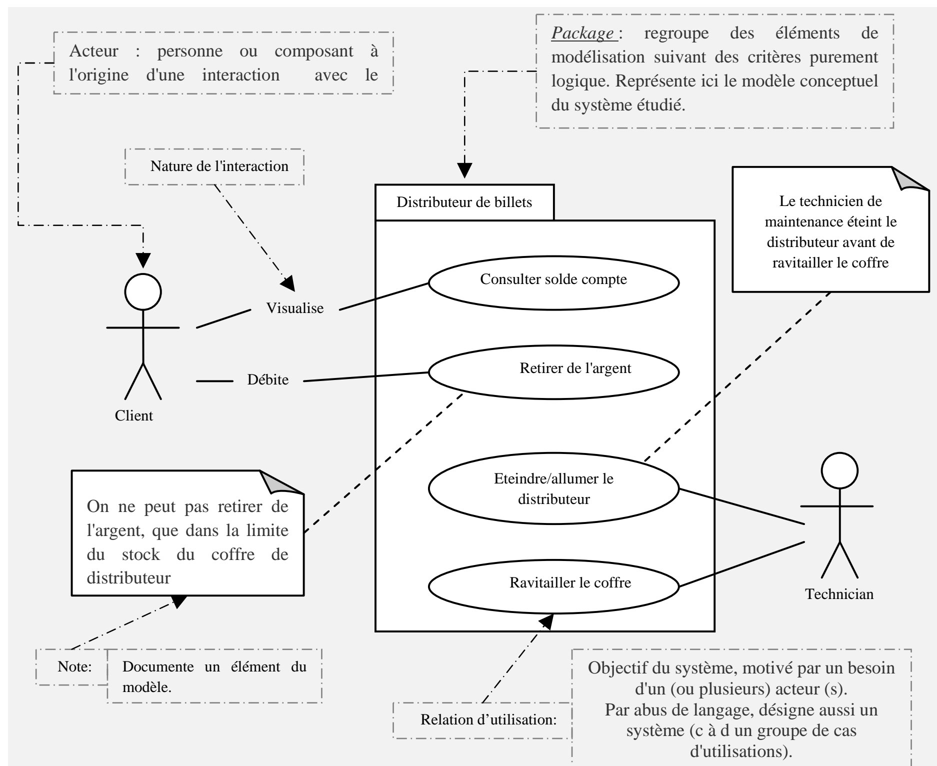


Fig.22 : Exemple d'un diagramme de Uses Case [4]

b) Diagramme d'activité (Activity diagram)

Les diagrammes d'activités permettent de mettre l'accent sur les traitements. Ils sont donc particulièrement adaptés à la modélisation du cheminement de flots de contrôle et de flots de données. Un diagramme d'activité visualise un graphe d'activité qui modélise le comportement interne d'une méthode (la réalisation d'une opération), d'un cas d'utilisation ou plus généralement d'un processus impliquant l'utilisation d'un ou plusieurs classificateurs [4].

Un diagramme d'activités représente l'état de l'exécution d'un mécanisme, sous la forme d'un déroulement d'étapes regroupées séquentiellement dans des branches parallèles de flot de contrôle. Le début et la fin (si elle existe) d'un mécanisme sont définis respectivement par un état initial et un état final [4].

Dans la phase de conception, les diagrammes d'activités sont particulièrement adaptés à la description des cas d'utilisation. Plus précisément, ils viennent illustrer et consolider la description textuelle des cas d'utilisation. De plus, leur représentation sous forme d'organigrammes les rendent facilement intelligibles et beaucoup plus accessibles.

La figure (Fig.23) représente un exemple de diagramme d'activités

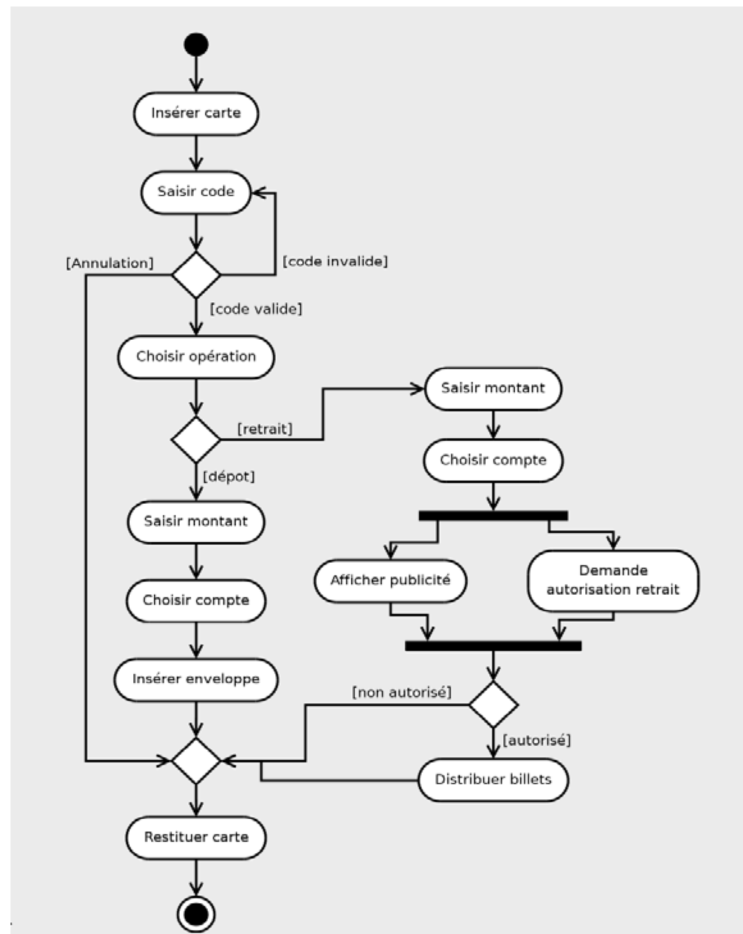


Fig.23 : Exemple d'un diagramme d'activité [1]

c) Diagramme d'états- transitions (State Machine)

Un diagramme d'états-transitions, ou diagramme de machine d'états (State machine diagram), permet de décrire le comportement interne (les changements d'états) d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets/composant ou avec des acteurs [31], à l'aide d'un automate à états finis.

Nous allons présenter le diagramme d'états-transitions, avec plus de détails dans le prochain chapitre, qui sera consacré à la description de la syntaxe et la sémantique des machines d'états, avec un survol sur les approches qui travaillent sur la formalisation de ce diagramme.

La figure (Fig.24) représente un exemple de diagramme d'états-transitions.

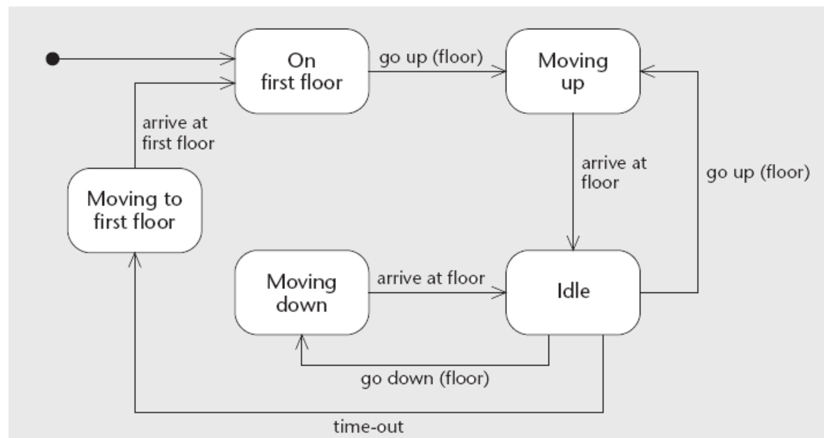


Fig.24 : Exemple d'un diagramme d'états-transitions [6]

3.5.3 Diagramme d'interaction (interaction diagram)

Les diagrammes d'interaction permettent d'établir un lien entre les diagrammes de cas d'utilisation et les diagrammes de classes : ils montrent comment des objets (des instances de classes) communiquent pour réaliser une certaine fonctionnalité. Ils apportent un aspect dynamique à la modélisation du système [1]. On trouve dans cette catégorie les diagrammes de séquence, du global d'interaction, de communication et du temps.

a) Diagramme de séquence (Sequence diagram)

Un diagramme de séquence montre des interactions entre un nombre d'objets en coopération afin d'accomplir une fonction précise. Les principales informations contenues dans ce diagramme sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique. Ainsi, contrairement au diagramme de communication, le temps y est représenté explicitement par une dimension (la dimension verticale). Il s'écoule de haut en bas [1]. La figure (Fig.25) présente un exemple d'un diagramme d'états-transition d'un serveur d'impression [6].

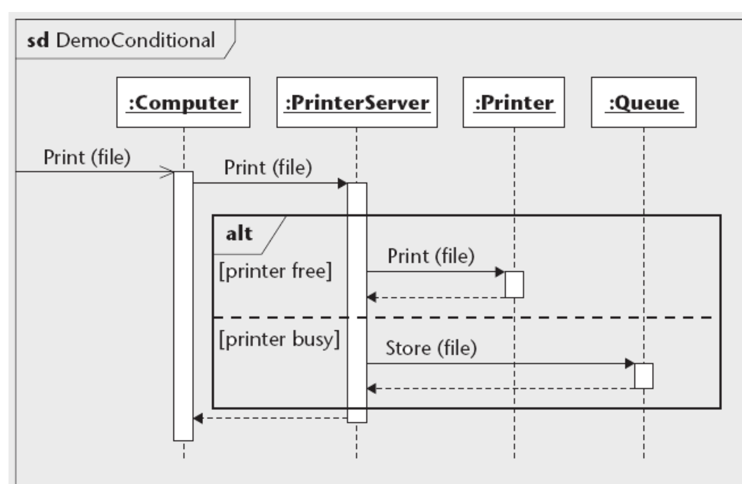


Fig.25 : Exemple d'un diagramme de séquence [6]

b) Diagramme de communication (Communication diagram)

Les diagrammes de communication, diagramme de collaboration en UML1, sont des cas particulier de diagrammes d'interactions qui représentent une vue dynamique du système. Les diagrammes de communication présentent un ensemble de rôles joués par des objets dans un contexte particulier, ainsi que les liens entre ces objets. Ils montrent également des interactions entre ces objets à travers la représentation d'envois de messages. Ils insistent plus particulièrement sur la structure spatiale qui permet la mise en collaboration d'un groupe d'objet [4]. Sous des formes distinctes les diagrammes de collaboration sont équivalents aux diagrammes de séquences. Seul l'aspect chronologique est cependant également présent (les messages sont numérotés).

- ***Définition d'une collaboration*** : Une collaboration définit les éléments qui sont utiles pour l'obtention d'un objectif particulier en spécifiant le rôle de ces éléments dans le contexte de la collaboration. Une collaboration est la réalisation d'une opération ou d'un classificateur (classe, cas d'utilisation, …) dans un contexte donné. Le comportement de cette opération ou de ce classificateur définit le rôle des divers éléments intervenant dans la collaboration.
- ***Représentation des messages*** : Dans un diagramme de communication, les messages sont généralement ordonnés selon un numéro de séquence croissant.
  Un message est, habituellement, spécifié sous la forme suivante :
  ['[**'cond'**]' [**séq**] [*[||] '[**'iter'**]'] :] [**r** :=] **msg**([**par**])
  cond est une condition sous forme d'expression booléenne entre crochets.
  séq est le numéro de séquence du message. On numérote les messages par envoi et sous-envoi désignés par des chiffres séparés par des points
  iter spécifie (en langage naturel, entre crochets) l'envoi séquentiel (ou en parallèle, avec ||). On peut omettre cette spécification et ne garder que le caractère "*" (ou "*||") pour désigner un message récurrent envoyé un certain nombre de fois.
  r est la valeur de retour du message, qui sera par exemple transmise en paramètre à un autre message.
  msg est le nom du message.
  par désigne les paramètres (optionnels) du message.
  Cette syntaxe un peu complexe permet de préciser parfaitement l'ordonnancement et la synchronisation des messages entre les objets du diagramme de communication. La direction d'un message est spécifiée par une flèche pointant vers l'un ou l'autre des objets de l'interaction, reliés par ailleurs avec un trait continu (connecteur).

  La figure (Fig.26) présente un exemple de diagramme de communication.

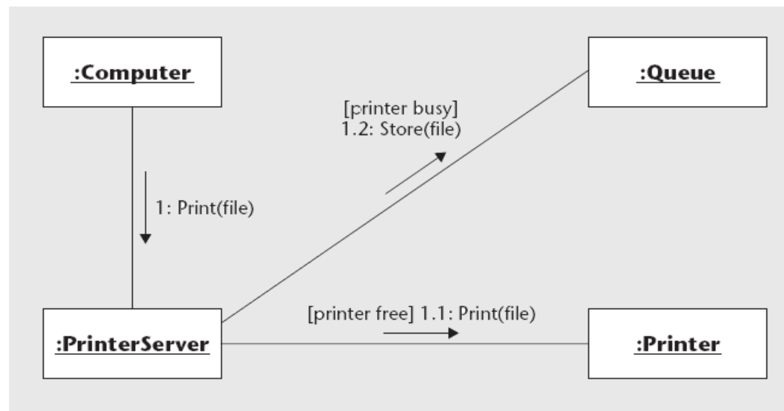


Fig.26 : Exemple d'un diagramme de communication [6]

c) Diagramme de temps (Timing diagram)

Le diagramme de temps (Timing diagram) est une nouveauté apparue dans UML2.0. Il permet de présenter l'interaction entre les objets actifs et leurs changements d'état sur un axe de temps. L'axe X présente le temps et contient des unités temporelles et l'axe Y montre les objets et leurs états [6]. La figure (Fig.27) montre un exemple de diagramme de temps.

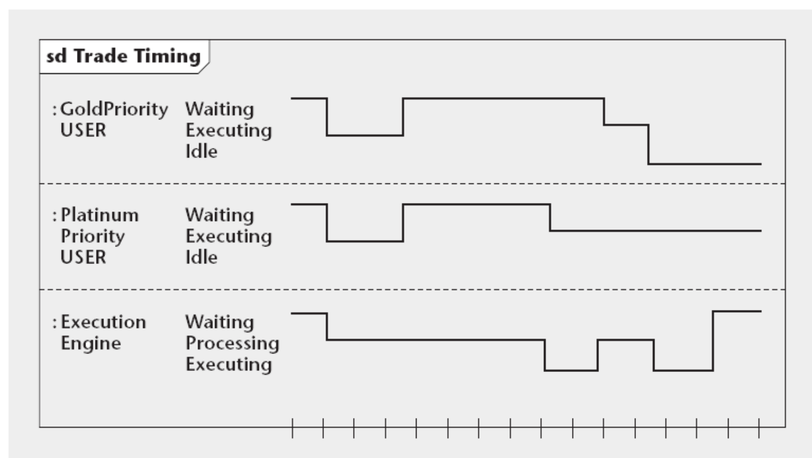


Fig.27 : Exemple d'un diagramme de Temps [6]

d) Diagramme global d'interactions (Interaction overview diagram)

Comme son nom indique, le diagramme global d'interaction donne une vue globale sur les interactions des objets actifs du système. Il regroupe des diagrammes d'activité et des diagrammes de séquence. Il est présenté soit comme un diagramme d'activités dont les activités sont remplacées par des diagrammes de séquence. Ou comme un diagramme de séquence contenant des notations du diagramme d'activité pour montrer le flux des activités [8]. La figure (Fig.28) montre un exemple de diagramme global d'interaction.

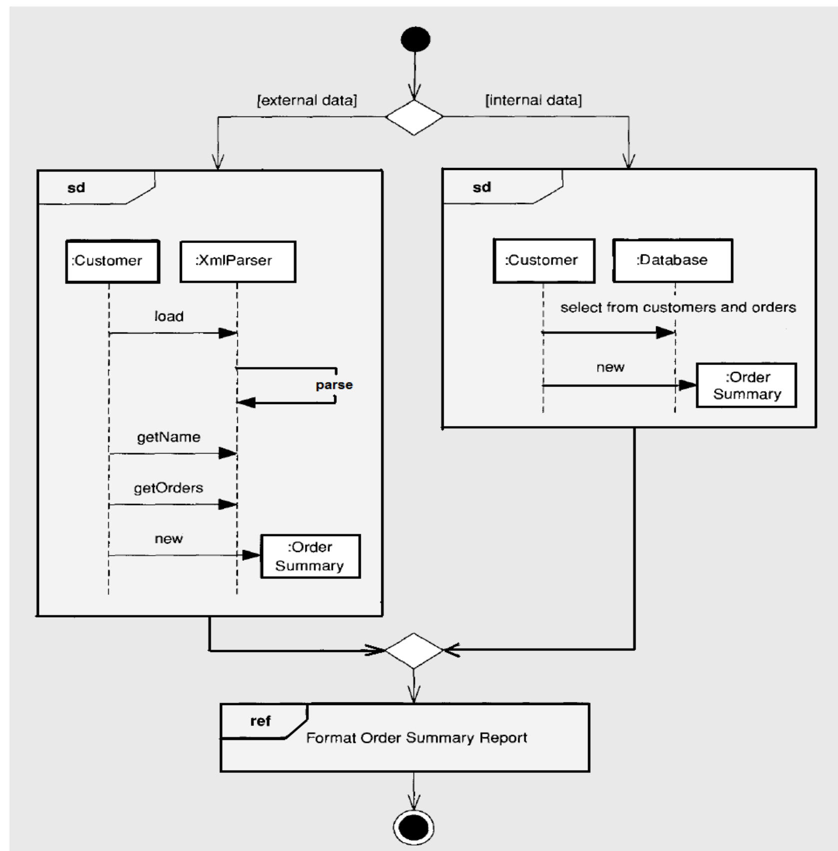


Fig.28 : Exemple d'un diagramme global d'interaction [8]

Avec les différents diagrammes on peut modéliser les différentes parties d'un produit logiciel, mais il reste qu'UML n'est pas une méthode ou un processus. En effet, UML ne propose pas une démarche de modélisation explicitant et encadrant toutes les étapes d'un projet de développement. Une méthode doit définir une séquence d'étapes, partiellement ordonnées, dont l'objectif est de produire un logiciel de qualité qui répond aux besoins des utilisateurs dans un temps opportun et avec un coût prévisible.

Les auteurs d'UML précisent qu'une méthode basée sur l'utilisation d'UML doit être piloté par les cas d'utilisation, centrée sur l'architecture et doit être itérative et incrémentale [10]. Ce qui a donné lieu au processus unifie. Par conséquent la section suivante est consacrée à la présentation de ce processus.

4. Le Processus Unifie

4.1 Définition

Le processus unifie (UP : Unified Process) constitue un cadre général de développement de logiciel. C'est un processus qui utilise UML. Il est caractérisé par le fait qu'il est piloté par les cas d'utilisation, il est itératif et incrémentale, centré sur l'architecture et orienté vers la diminution des risques.

- **Piloté par les cas d'utilisation** : l'objectif principal d'un système informatique est de répondre parfaitement aux besoins de ces utilisateurs. Par ce fait, UP se focalise sur les utilisateurs (utilisateurs humains ou autres systèmes). Les cas d'utilisation font apparaître parfaitement les besoins fonctionnels des utilisateurs et constituent dans leur ensemble le modèle des cas d'utilisation qui décrit les fonctionnalités complètes du système. Par ce fait, les auteurs de ce processus suggèrent que les cas d'utilisation vont complètement guider le processus de développement à travers l'utilisation de modèles basés sur l'utilisation du langage UML [10].

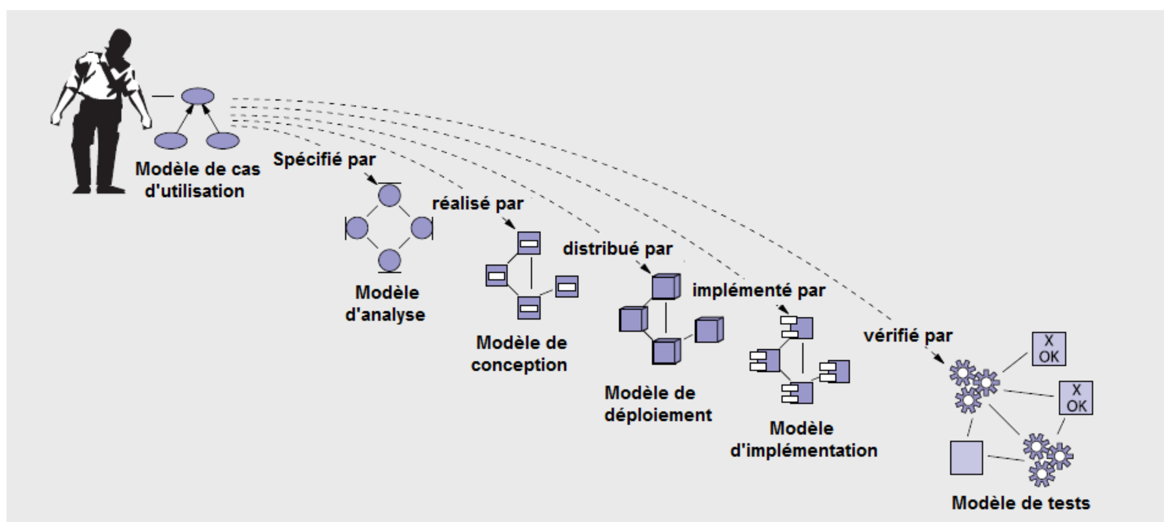


Fig.29 : Dépendances entre les cas d'utilisation et les différents modèles du Processus unifie [10]

Donc, à partir du modèle des cas d'utilisation, les développeurs créent une série de modèles de conception et d'implémentation réalisant les cas d'utilisation. Ensuite, chacun des modèles successifs est révisé pour en contrôler la conformité par rapport au modèle des cas d'utilisation. Enfin, les testeurs testent l'implémentation pour s'assurer que les composants du modèle d'implémentation mettent correctement en œuvre les cas d'utilisation.

- **Centré sur l'architecture** : dans le processus unifie, l'architecture à mettre en place est visé dés le démarrage de processus. L'architecture d'un système logiciel est décrite comme les différentes vues du système qui doit être construit. L'architecture émerge des

besoins de l'entreprise, tels qu'ils sont exprimés par les utilisateurs et les autres intervenants, et tels qu'ils sont reflétés par les cas d'utilisation.

L'architecture subit également l'influence d'autres facteurs :

- la plate-forme sur laquelle devra s'exécuter le système ;
- les briques de bases réutilisables disponibles pour le développement ;
- les considérations de déploiement, les systèmes existants et les besoins non fonctionnels (performance, fiabilité...)

Une forte liaison entre les cas d'utilisation et l'architecture doit avoir lieu durant l'élaboration du processus, où chaque cas d'utilisation doit, une fois réalisé, trouver leur place dans l'architecture. Et L'architecture doit prévoir la réalisation de tout les cas d'utilisations.

- **Itératif et incrémentale** : généralement, le projet de développement d'un produit logiciel est découpé en plusieurs parties qui sont autant des mini-projets. Chacun d'entre eux représente une itération qui donne lieu à un incrément. Une itération désigne la succession des étapes de l'enchaînement d'activités, tandis qu'un incrément correspond à une avancée dans les différents stades de développement.

  Une itération prend en compte un certain nombre de cas d'utilisation qui ensemble, améliorent l'utilisabilité du produit à un certain stade de développement, et traite en priorité les risques majeurs.
  A chaque itération, les développeurs identifient et spécifient les cas d'utilisations pertinents, créent une conception en se laissant guider par l'architecture choisie, implémentent cette conception sous forme de composants et vérifie que ceux ci sont conformes aux cas d'utilisation. Dés qu'une itération répond aux objectifs fixés le développement passe à l'itération suivante.
  Pour rentabiliser le développement il faut sélectionner les itérations nécessaires pour atteindre les objectifs du projet. Ces itérations devront se succéder dans un ordre logique.
  Un projet réussi suivra un déroulement direct, établi dés le début par les développeurs et dont ils ne s'éloigneront que de façon très marginale. L'élimination des problèmes imprévus fait partie des objectifs de réduction des risques.

4.2 Vie du Processus Unifie

Le processus unifie vise à maîtriser la complexité des projets informatiques en diminuant les risques. UP est un ensemble de principes génériques adaptés en fonctions des spécificités des projets. Il répond aux questions suivantes :

- **QUI** participe au projet ?
- **QUOI**, qu'est-ce qui est produit durant le projet ?
- **COMMENT** doit-il être réalisé ?
- **QUAND** est réalisé chaque livrable ?

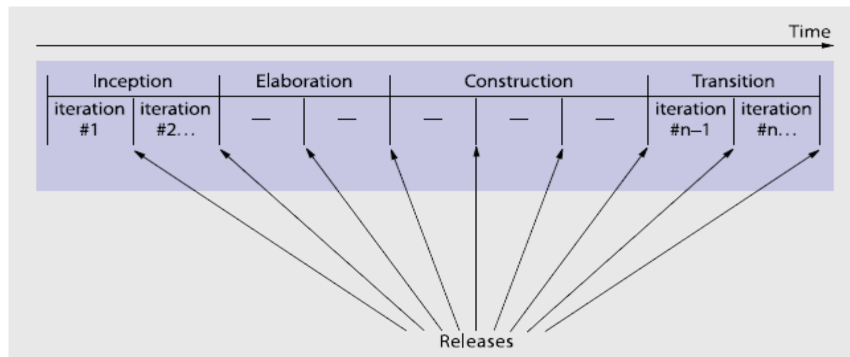


Fig.30 : un cycle avec ses phases et ses itérations [10]

Le processus unifié répète un certain nombre de fois une série de cycles. Tout cycle se conclut par la livraison d'une version du produit aux clients (Fig.30) et s'articule en 4 phases (Fig.31): **Création, Elaboration, Construction et Transition**.

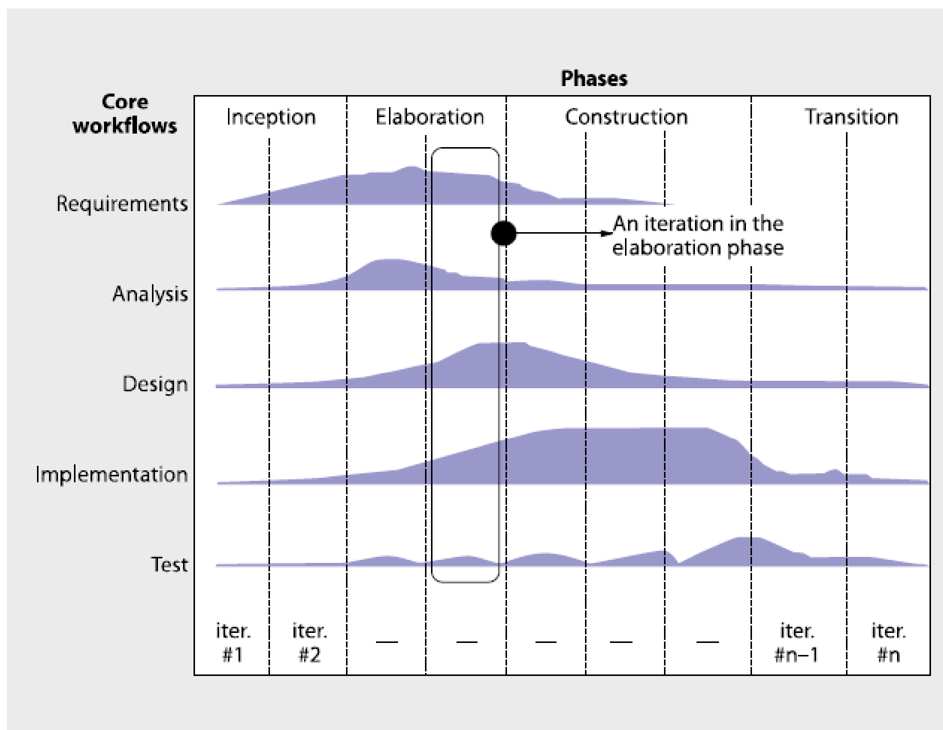


Fig.31 : les 4 phases et les 5 workflow dans le PU[10]

Chaque phase se subdivise à son tour en itérations. Chaque cycle se traduit par une nouvelle version du système. Ce produit se compose d'un corps de code source réparti sur plusieurs composants pouvant être compilés et exécutés et s'accompagne de manuels et de produits associés. Pour mener efficacement le cycle, les développeurs ont besoin de construire toutes les représentations du produit logiciel [10] :

- **Modèle des cas d'utilisation :** représente les cas d'utilisations du système et leurs relations avec les utilisateurs ;
- **Modèle d'analyse :** explore et détaille les cas d'utilisation et procède à une première répartition du comportement du système entre divers objets ;
- **Modèle de conception :** définit la structure statique du système sous forme de sous système, classes et interfaces ; Définit les cas d'utilisation réalisés sous forme de collaborations entre les sous systèmes les classes et les interfaces;
- **Modèle d'implémentation :** intègre les composants (code source) et la correspondance entre les classes et les composants;
- **Modèle de déploiement :** définit les nœuds physiques des ordinateurs et l'affectation de ces composants sur ces nœuds;
- **Modèle de test :** décrit les cas de test vérifiant les cas d'utilisation;
- **Représentation de l'architecture :** il s'agit d'une description de l'architecture**.**

Tous ces modèles sont liés. Ensemble, ils représentent le système comme un tout. Les éléments de chacun des modèles présentent des dépendances de traçabilité ; ce qui facilite la compréhension et les modifications ultérieures.

4.3 Les tâches d'un cycle de vie d'UP

Les activités d'un cycle de vie du processus unifie sont [10]:

- **L'expression des besoins :** il s'agit de définir les différents besoins :
  - inventorier les besoins principaux et fournir une liste de leurs fonctions
  - recenser les besoins fonctionnels (du point de vue de l'utilisateur) qui conduisent à l'élaboration des modèles de cas d'utilisation
  - appréhender les besoins non fonctionnels (technique) et livrer une liste des exigences.

  Le modèle de cas d'utilisation peut exprimer les besoins d'utilisateur. Il présente le système du point de vue de l'utilisateur et se forme de cas d'utilisation et d'acteurs.

- **Analyse :** c'est la tâche de compréhension des besoins et des exigences d'utilisateur. Il s'agit de livrer des spécifications pour permettre de choisir la conception de la solution.
  Le modèle d'analyse livre une spécification complète des besoins issus des cas d'utilisation et les structure sous une forme qui facilite la compréhension (scénarios), la préparation (définition de l'architecture), la modification et la maintenance du futur système.

- **Conception :** La conception permet d'acquérir une compréhension approfondie des contraintes liées au langage de programmation, à l'utilisation des composants et au système d'exploitation.
  Elle détermine les principales interfaces et les transcrits à l'aide d'une notation commune.
  Elle constitue un point de départ à l'implémentation :
  - elle décompose le travail d'implémentation en sous-système.

- elle créée une abstraction transparente de l'implémentation.

- **Implémentation :** L'implémentation est le résultat de la conception, pour implémenter le système sous formes de composants, c'est-à-dire, de code source, de scripts, de binaires, d'exécutables …etc.
  Les objectifs principaux de l'implémentation sont de planifier les intégrations des composants pour chaque itération, et de produire les classes et les sous-systèmes sous formes de codes sources.

- **Test :** Les tests permettent de vérifier des résultats de l'implémentation en testant la construction.
  Pour mener à bien ces tests, il faut les planifier pour chaque itération, les implémenter en créant des cas de tests, effectuer ces tests et prendre en compte le résultat de chacun.

4.4 Les phases d'un cycle de vie d'UP

Les phases d'un cycle de vie du processus unifie sont [10] :

- **Phase de création :** Traduit une idée en vision de produit fini et présente une étude de rentabilité pour ce produit. Elle répond aux questions suivantes :
  - Que va faire le système pour les utilisateurs ?
  - A quoi peut ressembler l'architecture d'un tel système ?
  - Quels sont l'organisation et les coûts du développement de ce produit ?

  On fait apparaître les principaux cas d'utilisation. L'architecture est provisoire, identification des risques majeurs et planification de la phase d'élaboration.

- **Phase d'élaboration :** L'élaboration reprend les éléments de la phase d'analyse des besoins et les précise pour arriver à une spécification détaillée de la solution à mettre en œuvre.
  L'élaboration permet de préciser la plupart des cas d'utilisation, de concevoir l'architecture du système et surtout de déterminer l'architecture de référence.
  Les taches à effectuer dans la phase élaboration sont les suivantes :
  - créer une architecture de référence ;
  - identifier les risques, ceux qui sont de nature à bouleverser le plan, le coût et le calendrier ;
  - définir les niveaux de qualité à atteindre ;
  - formuler les cas d'utilisation pour couvrir les besoins fonctionnels et planifier la phase de construction ;
  - élaborer une offre abordant les questions de calendrier, de personnel et de budget.

- **Phase de construction :** La construction est le moment où l'on construit le produit. L'architecture de référence se métamorphose en produit complet.

Le produit contient tous les cas d'utilisation que les chefs de projet, en accord avec les utilisateurs ont décidé de mettre au point pour cette version.

Phase de transition : Le produit est en version bêta. Un groupe d'utilisateurs essaye le produit et détecte les anomalies et défauts.
Cette phase suppose des activités comme la formation des utilisateurs clients, la mise en œuvre d'un service d'assistance et la correction des anomalies constatées.

En conclusion, et selon [10], Le processus unifié est basé sur des composants. Il utilise UML et est basé sur les cas d'utilisation, l'architecture et le développement sont itératif et incrémental. Afin de mettre en pratique ces idées, il faut recourir à un processus multi-facettes prenant en considération les cycles, les phases, les enchaînements d'activités, la réduction des risques, le contrôle de qualité, la gestion de projet et la gestion de configuration. Le processus unifié a mis en place un cadre général (Framework) intégrant chacune de ces facettes.

Dans ce chapitre nous avons vu quelques définitions et concepts sur le génie logiciel, UML, et le processus unifie. Le prochain chapitre sera consacré au diagramme d'états-transitions dont on va détailler ses éléments.

CHAPITRE II :

Diagrammes d’états-transitions & Travaux de Formalisation

Le langage de modélisation unifié est devenu depuis quelques années un standard incontournable dans l’industrie des logiciels. Néanmoins, plusieurs critiques sont adressés à ce langage, parmi les plus importants, UML n’est pas un langage formel. C’est pourquoi plusieurs efforts ont été fondés pour la formalisation de ce langage.

Le diagramme d’états-transitions ou state machine en anglais, étais le diagramme le plus visé par les travaux de formalisation dû à sa spécificité et sa large utilisation dans la modélisation de comportements internes des objets. L’approche qu’en propose nous même est basée aussi sur ce diagramme, c’est ainsi que le présent chapitre est consacré à la description du diagramme de machine d’états. Dans la première section, nous présentons la syntaxe et la sémantique de ce diagramme. Dans la deuxième partie nous présentons l'état de l'art des approches de formalisation du diagramme états-transitons.

1. Diagramme d’états-transitions

1.1 Définition

Le diagramme d’états-transitions d’UML décrive le comportement interne d’un objet à l’aide d’un automate à états finis. Il spécifie les séquences possibles d’états et d’actions qu’un objet peut traiter au cours de sa vie en réactions à des événements (signaux, invocation de méthodes).

Le diagramme d’états-transitions est le seul diagramme en UML qui offre une vision complète et non ambiguë de l’ensemble des composants de l’élément auquel il est attaché [1].

1.2 Etats

L’état d’un objet est une condition ou une situation durant la vie d’un objet au cours de laquelle il satisfait certaines conditions, accomplit une activité ou attend l’arrivé d’un événement [11]. Un objet reste dans un état pour un temps limité, par exemple un chauffage pourrait être dans un des quatre états suivants : inactif (en attende d’une commande pour démarrer),

démarrage (le gaz circule dans le chauffage mais le dernier attend l'atteinte d'un certain degré de température), actif (le chauffage en action) et l'état fermeture (une fois le chauffage est fermé).

Un état à plusieurs parties à savoir :

a- **Nom** : une chaîne textuelle qui distingue l'état d'autres états, un état peut être anonyme.

b- **Actions d'entrées/sorties** : ce sont les actions à exécutées à l'entrée et à la sortie de l'état.

c- **Transitions internes** : les transitions à effectués sans provoquer un changement d'état de l'objet.

d- **Sous-états :** la structure imbriquée de l'état. L'état peut être simple ou composite.

e- **Evénements reportés (déférés) :** une liste d'événements qui ne sont pas traiter dans cet état mais qui sont sauvegarder dans une fille d'attente pour qu'ils soient traiter par l'objet dans d'autres états.

La figure (Fig.32) représente quelques états simples dans un diagramme d'états-transitions. Un état est représenté graphiquement par un rectangle aux coins arrondi.

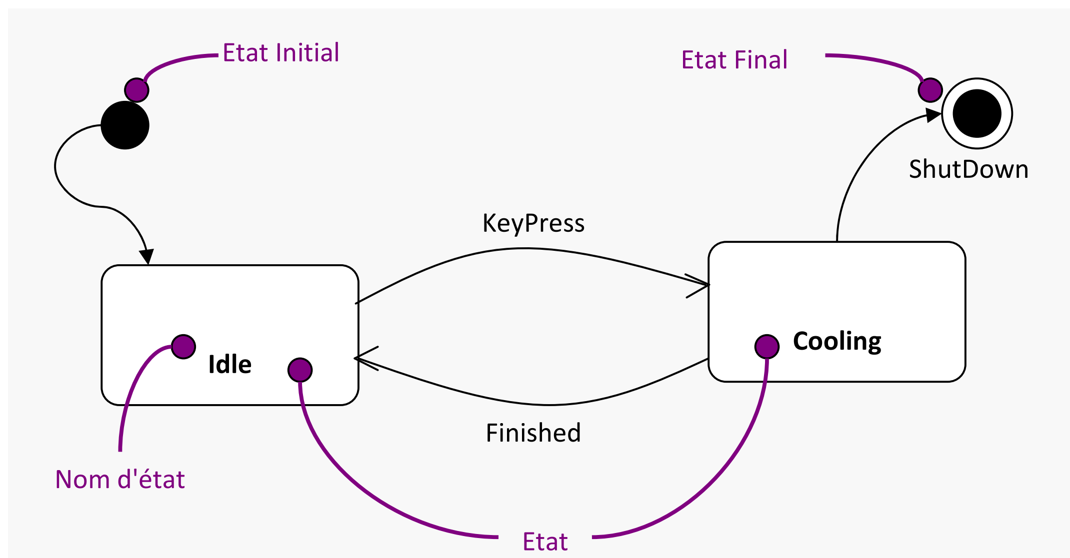


Fig.32 : deux états simples dans un diagramme d'états-transitions [11]

- **Etat initial et final**

Comme la figure (Fig.32) montre, deux états spéciaux sont définis dans un diagramme d'états-transitions ; premièrement, un état initial qui indique la place de départ par défaut du diagramme ou d'un état composite. Deuxièmement, un état final qui indique que l'exécution du diagramme ou l'état composite est terminée.

1.3 Evénements

Un événement est quelque chose qui se produit pendant l'exécution d'un système et qui mérite d'être modélisé [1]. Les diagrammes d'états-transitions permettent de spécifier les réactions d'une partie du système à des événements discrets. Un événement se produit à un instant précis. Lors de la réception d'un événement une transition peut être déclenchée et un changement d'état a lieu pour basculer l'objet d'un état à un nouvel état.

On trouve plusieurs types d'événement à savoir [1]:

a- **événement signal :** Un signal est un type d'événement destiné explicitement à véhiculer une communication asynchrone à sens unique entre deux objets. L'objet expéditeur n'attend pas que le destinataire traite le signal pour poursuivre son déroulement. Un même objet peut être à la fois expéditeur et destinataire.

b- **événement d'appel :** Un événement d'appel représente la réception d'une invocation de méthode par un objet. Les paramètres de l'opération sont ceux de l'événement d'appel. Les événements d'appel sont généralement déclarés au niveau du diagramme de classe.

c- **événement de changement :** Un événement de changement est généré par la satisfaction d'une expression booléenne sur des valeurs d'attributs. Il s'agit d'une manière déclarative d'attendre qu'une condition soit satisfaite. Un événement de changement est évalué continuellement jusqu'à ce qu'il devient vrai, et c'est à ce moment-là que la transition se déclenche.

d- **événement temporelle :** Les événements temporels sont générés par le passage du temps. Ils sont spécifiés soit de manière absolu (date précise), soit de manière relative. Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant. La syntaxe d'un événement temporel spécifié de manière relative est la suivante : **After (<durée>)**. Un événement temporel spécifié de manière absolue est définie en utilisant un événement de changement : **when (date= <date>).**

1.4 Transitions

Une transition est une relation entre deux états. La transition indique que l'objet qui est dans le premier état va accomplir certaines actions et entre au deuxième état une fois qu'un événement précis se produit et une condition spécifique soit vérifiée [11].

Une transition est constituée de cinq partie à savoir ;

a- **Etat source :** C'est l'état dans lequel se trouve l'objet avant le franchissement de la transition.

b- **Evénement déclencheur (event trigger)** : c'est l'événement reconnus par l'état source et avec lequel la transition est franchissable une fois la garde de la transition est vérifiée. Un événement peut être un signal, un appel de méthode, un passage de temps ou un changement d'état [11].

- **c- Condition de garde** : une transition peut avoir une condition de garde (spécifiée par '['<garde> ']' dans la syntaxe). Il s'agit d'une expression logique sur les attributs de l'objet [1]. La transition ne se déclenche qu'une fois la condition de garde est évaluée comme vraie.
- **d- effet** : une fois une transition se déclenche (on parle également de tir (franchissement) d'une transition), son effet (spécifié par '/' <activité> dans la syntaxe) s'exécute. Il s'agit généralement ; des opérations primitives tels que assignations, un envoi d'un signal à l'objet lui-même ou à un autre objet, appel de méthodes, ou une liste d'activités, etc. [1].
- **e- état cible** : c'est l'état actif (le nouvel état) de l'objet après le franchissement de la transition.

La figure (Fig.33) montre un simple diagramme d'états-transitions, la figure se focalise sur la présentation des transitions et ses parties (garde, effets, événements, etc.). La transition dans le diagramme est une flèche orienté de l'état source vers l'état cible.

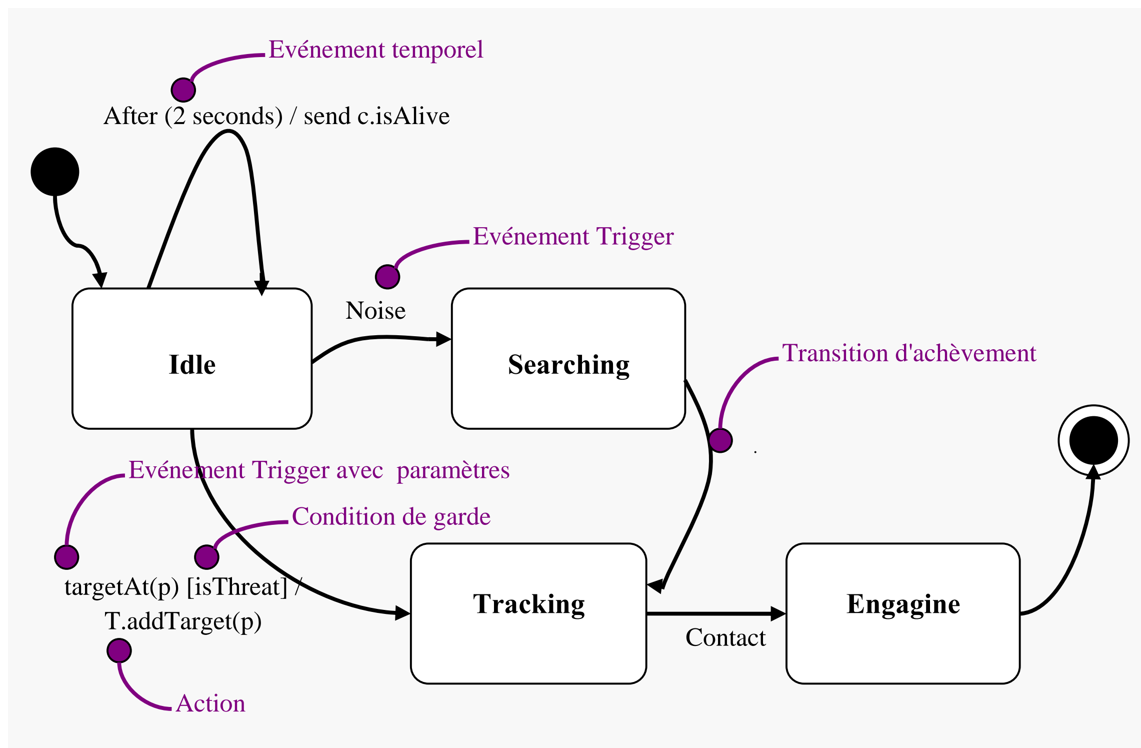


Fig.33 : les Transitions [11]

Une transition peut avoir plusieurs sources (dans ce cas, elle représente une jonction entre plusieurs états concurrents). Elle peut avoir aussi plusieurs états cibles.

1.5 Etats et Transitions avancés

Le diagramme d'états-transitions contient plusieurs options qui nous permettent de modéliser des comportements complexes. Ces options souvent diminuent le nombre d'états et de transitions dans un diagramme. Parmi ces options on trouve les actions d'entrée/sortie, les transitions internes, les activités 'do', les événements déférés, etc. la figure (Fig.34) représente ces options.

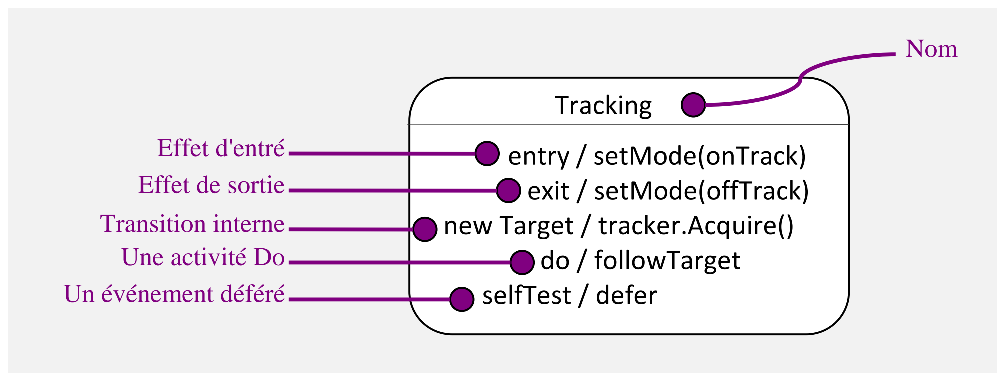


Fig.34 : options avancés d'états-transition [11]

1.5.1 Activités d'entrée/sortie

Dans quelques situations de modélisation, il est important d'exécuter quelques actions en entrée et en sortie de l'état. Comme la figure (Fig.33) le montre, UML propose deux mots clés pour indiquer les activités à exécuter en entrée et en sortie, à savoir [11];

- '**Entry**': entry permet de spécifier une activité qui s'accomplit quand on entre dans l'état.
- **'exit'** : exit permet de spécifier une activité qui s'accomplit quand on sort de l'état.

Les activités d'entrée/sortie généralement ne possèdent pas des arguments ou des conditions de garde. Toutefois, l'activité d'entrée pour le premier état dans le diagramme d'états-transitions peut avoir des paramètres pour les arguments reçus lors de la création de l'objet.

1.5.2 Transitions Internes

Parfois, dans un état donné, le concepteur rencontrera des événements qu'il souhaite traiter sans laisser l'état actuel de l'objet. Ces traitements se conduisent par les transitions internes. Les règles de déclenchement d'une transition interne sont les même que pour une transition externe excepté qu'une transition interne ne possède pas d'état cible et que l'état actif reste le même à la suite de son déclenchement. Le déclenchement d'une transition interne ne provoque pas les activités d'entrée et de sortie de l'état en cour.

1.5.3 Activités-Do

Quand un objet est dans un état donné, il reste généralement en attente de l'arrivé des événements. Mais parfois, on voudra à ce moment accomplir des activités jusqu'à ce que

l'événement apparaisse. Et comme il est présenté dans la figure (Fig.33), UML propose le mot clé **'do'** pour spécifier les activités do qui s'exécutent jusqu'à l'arrivé d'un événement. L'activité-do commence une fois dès que l'activité entry termine.

1.5.4 Evénements déférés

Nous avons mentionné que les événements déférés sont les événements qui ne sont pas traité par l'objet dans cet état. Mais plutôt ces événements seront grader dans une fille d'attente pour les traiter par l'objet dans d'autres états. Ça signifier que n'importe quel événements sera traiter par l'objet dans l'état en cour soit, il est déféré pour qu'il soit traiter ultérieurement par l'objet dans un autre état soit il est ignoré. Pour déclarer les événements déférés en utilise le mot clé '**defer'** la figure (Fig.33) montre un exemple d'un signal déféré.

1.5.5 Sous-machine

Une machine d'état (diagramme d'états-transitions) peut être liée à d'autres machines d'états. Cette machine est appelée sous-machine (submachine en anglais). Ces sous-machine permettent de construire des grand modèle d'états pour plus de détaille voir [12]

1.6 Sous-états

Un autre concept avancé dans les diagrammes d'états-transitions est les sous-états. Un sous-état est un état imbriqué dans un autre état. On dit qu'un état est simple lorsqu'il ne contient pas des sous état. Par contre, un état avec un sous-état ou plus est appelé un état composite.

Un état composite peut contenir des sous-états concurrents (orthogonaux) et/ou des sous-états séquentiels (non orthogonaux).

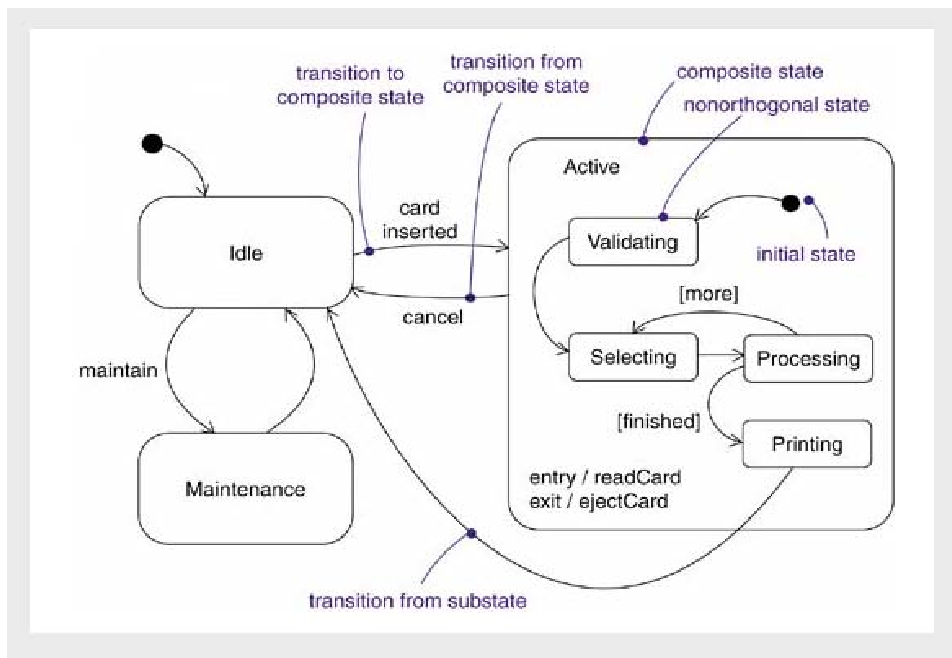


Fig.35 : sous-états séquentiels [11]

1.6.1 Sous-état séquentiel

Un état simple ne possède pas des sous-états. Par contre, un état composite est un état décomposé en une ou plusieurs régions contenant chacune un ou plusieurs sous-états.

Un état composite avec une seule région est qualifié comme état séquentiel où un état non concurrent. Généralement, cette région contient un état initial et un état final. La figure (Fig.35) représente un exemple d'un diagramme d'états-transitions qui contient un état séquentiel.

1.6.2 Sous-état Orthogonal

Le diagramme d'états-transitions permet de présenter efficacement les mécanismes concurrents grâce à l'utilisation d'états orthogonaux. Un état orthogonal est un état composite contenant plus d'une région, chaque région représente un flot d'exécution. Dans le diagramme les différentes régions sont séparées par un trait horizontal en pointillé allant du bord gauche au bord droit de l'état composite.

Chaque région optionnellement peut posséder un état initial et un état final. Une transition qui active l'état composite concurrent atteint en effet touts les états initiaux dans toutes les régions. Toutes les exécutions dont les régions concurrentes doivent atteindre leurs états finaux pour que l'état composite soit considéré comme terminé.

La figure (Fig36) montre un diagramme d'état-transitions qui contient un état concurrent (Maintenance) avec deux régions (Testing et commanding) qui sont exécutées en parallèle.

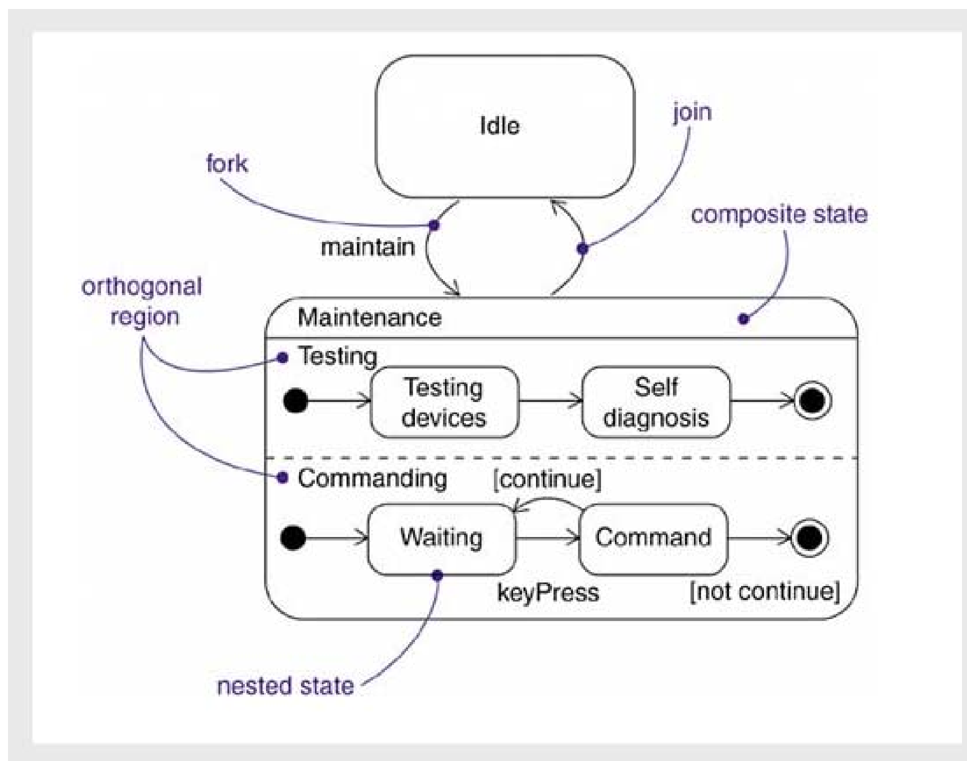


Fig.36 : Sous-états concurrents [11]

1.6.3 Etat Historique

Un état historique est un pseudo-état qui mémorise le dernier sous-état actif d'un état composite. Il est représenté dans le diagramme d'états-transition par un cercle contenant un H.

Une transition ayant comme cible un état historique est équivalente à une transition ayant comme cible le dernier état visité dans un état composite englobant. Un état historique peut avoir une transition sortante non étiquetée indiquant l'état à exécuter si la région n'a pas encore été visitée. La figure (Fig.37) montre un exemple illustrant l'utilisation des états historiques.

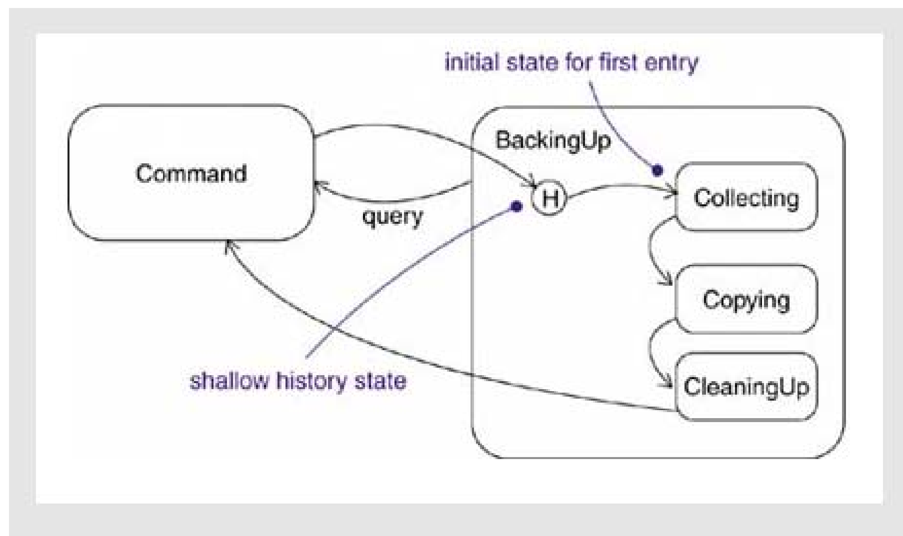


Fig.37 : Etat historique [11]

1.7 Point de choix

UML offre la possibilité de représenter des alternatives pour le franchissement d'une transition par des pseudo-états particuliers :

- les points de jonction
- les points de décision

1.7.1 Point de Jonction

Un point de jonction est un pseudo-état qui permet de partager des segments de transition, dont l'objectif et de donner une notation plus lisible des chemins alternatifs [1]

Un point de jonction peut avoir plusieurs segments de transitions entrantes et sortantes. Chaque segment contient des déclencheurs d'événements.

La figure (Fig.38) illustre l'utilisation des points de jonction dans un diagramme d'état-transition

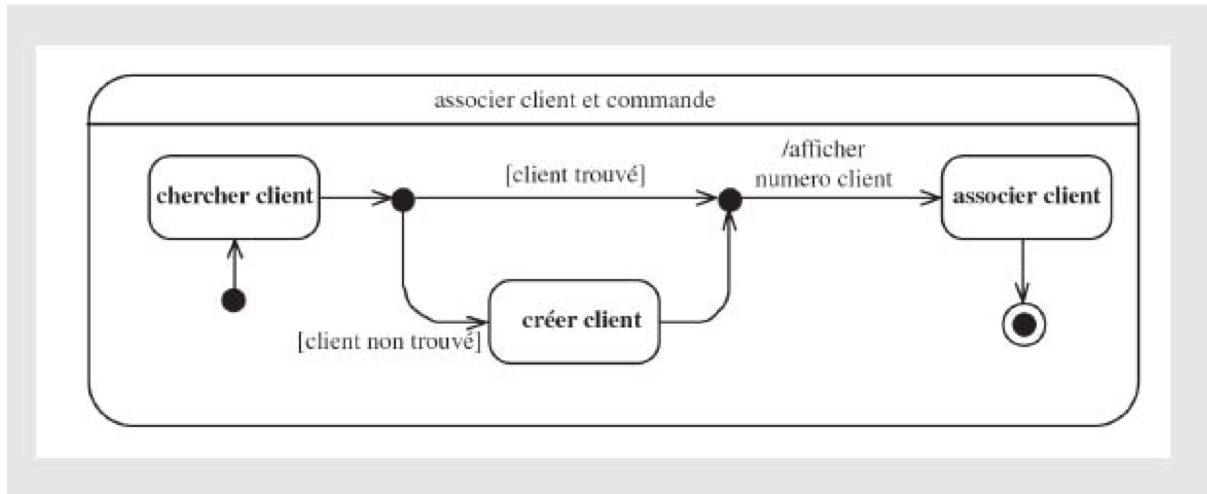


Fig.38 : exemple d'utilisation des points de jonction [1]

1.7.2 Point de Décision

Un point de décision et un pseudo état qui possède une entrée et au moins deux sorties. Les gardes situés après le point de décision sont évalués avant le choix (la prise de décision). Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix.

Une garde particulière, noté [else], peut être utilisé en aval d'un point de décision, ce segment n'est franchissable que si les gardes des autres segments sont évaluées comme fausse.

La figure (Fig.39) montre un exemple d'utilisation des points de décision.

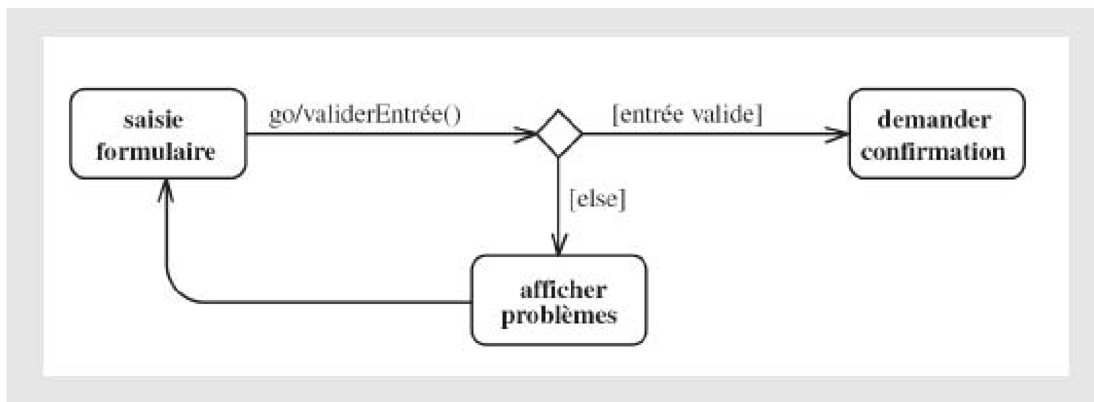


Fig.39 : exemple d'utilisation de point de décision [1]

1.8 Transition complexes

Le diagramme d'états-transitions offre la possibilité d'utiliser des transitions concurrentes. Ces transitions sont des transitions complexes et qui permettent de:

- Passer le contrôle d'un état à plusieurs états concurrents (éventuellement des régions orthogonales) en utilisant un branchement de type **Fork**. Ce branchement est représenté dans les diagrammes par une barre épaisse avec une seule flèche entrantes et plusieurs flèches sortantes.
- Passer le contrôle de plusieurs états concurrent (éventuellement des régions orthogonales) à un seul état en utilisant un branchement de type **Join**. Ce dernier est représenté par une barre épaisse avec plusieurs flèches entrantes et une unique flèche sortante.

La figure (Fig.40) représente

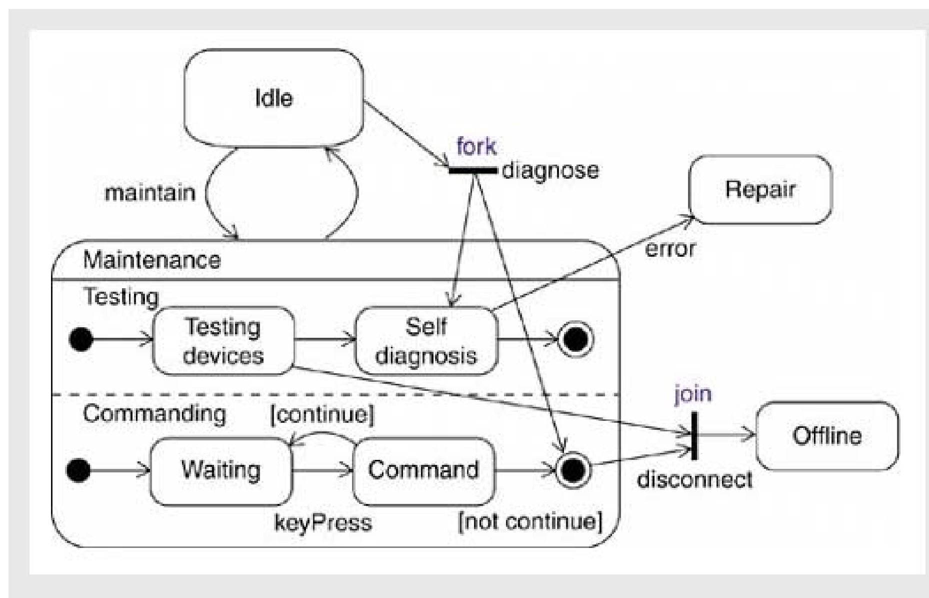


Fig.40 : Transition complexe avec Join et Fork [11]

1.9 Sémantique de Machine d'état

Le diagramme d'états-transition, comme nous avons dis précédemment, permet de spécifier le comportement d'un objet actif où il représente les différents états de cet objet durant sa vie. La sémantique de l'exécution du diagramme est décrite par une machine hypothétique [13]. Cette machine est composé de :

- **Une file d'attente**: elle permet d'enfiler (stocker) les instances d'événements non encore consommées.
- **Un mécanisme pour dispatché les événements** : il a comme rôle de sélectionner et d'enfiler les instances d'événements qui sont stocké dans la file d'attente.
- **Un processeur d'événements** : il traites les événements dispatchés.

Chaque objet va exécuter l'algorithme "pas RTC" (Run-To-Completion step), qui dispatche et exécutes les événements à partir de sa file d'attente jusqu'à ce que la machine d'état se termine. Les standards ne définissent pas la structure exacte de la file d'attente.

La tâche principale du processeur d'événements est de trouver toutes les transitions franchissables avec le premier élément (événement) dans la file. Ainsi, l'ensemble de transition sont groupé dans un seul pas. Les transitions dans ce pas sont exécutées en séquence. Une fois le pas est terminé (toutes les transitions sont exécutées), le processeur d'événement va passer à l'événement suivant dans la file.

Dans les sections précédentes de ce chapitre, nous avons introduit la syntaxe et la sémantique du diagramme d'états-transitions. Dans le reste du chapitre, nous allons présenter brièvement les travaux de formalisation du diagramme de machines d'états.

2. Travaux de formalisation de diagramme d'états transitions.

Dans la littérature, la formalisation de la machine d'états d'UML était le vif de plusieurs approches mathématiques, non-mathématiques, textuelles, graphiques …etc. L'objectif de cette formalisation est de préciser une sémantique non ambiguë permettant de vérifier formellement la dynamique représentée par une machine d'états UML.

Dans cette section nous allons présenter l'état de l'art de ces approches, tout en se basant sur la catégorisation du [14] et [15] qui est représenté par la figure (Fig.41)

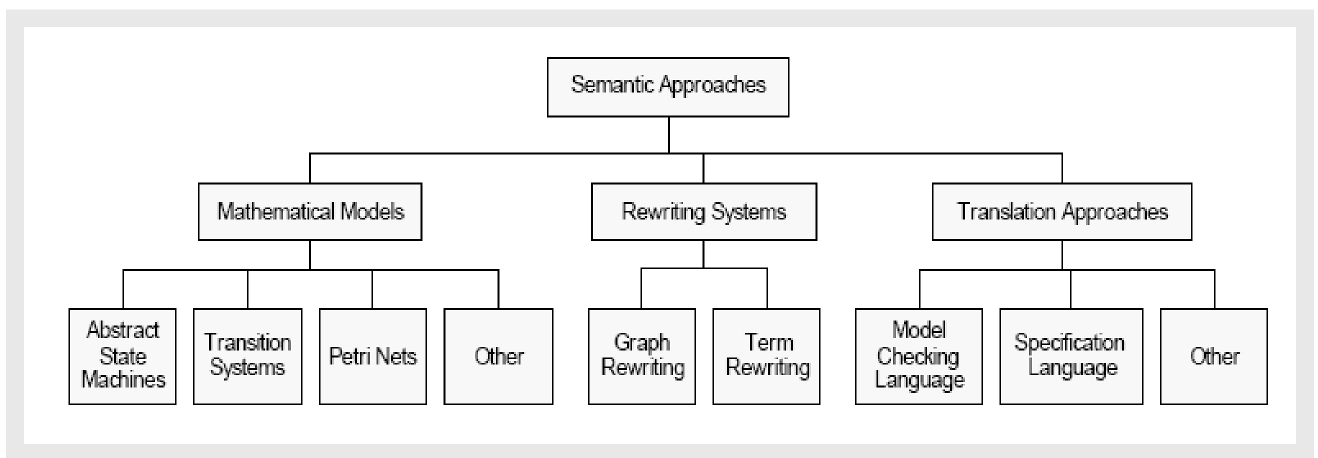


Fig.41 : Catégorisation des approches sémantiques du diagramme d'états-transition [14]

2.1 Les Modèles Mathématiques

Cette catégorie comprend les approches sémantiques qui sont fondées sur des concepts et des notations mathématiques standard [14]. L'avantage d'utiliser une notation mathématique et qu'elle supporte la précision et la prise en charge des détails, en donnant une sémantique complète, précise et non ambiguë.

En principe, les notations utilisées dans ces approches sont simples et accessibles à n'importe quel utilisateur ayant des notions de bases en mathématiques. Néanmoins, ces approches n'arrivent pas à fournir un haut niveau d'abstraction qui permet aux utilisateurs des modèles de les comprendre aisément [16].

Cette catégorie en elle-même est divisée en sous-catégories à savoir :

1. **Les Systèmes de Transitions**: Généralement, un système de transition est un graphe dont les nœuds représentent les états et les arêtes représentent les transitions entre ces états. Il existe plusieurs types de systèmes de transition, y compris les systèmes de transition libellées (LTS : Labled Transition System), la structure de Kripke et les systèmes de transitions symboliques. Dans cette sous-catégorie on trouve :

- L'approche décrite en [17] définie une sémantique pour un sous-ensemble d'UML orientée vers les concepts temps-réel (création et destruction des objets, changement de topologie ...etc.). Cette sémantique associe à chaque modèle un système de transition libellé. L'objectif de l'approche est de donner des fondements formels, des méthodes et des outils pour spécifier et vérifier des systèmes temps-réel modélisés par UML.

- L'approche présentée en [18] utilise les Systèmes de Transitions Libellées et les structures de Kripke afin de donner une sémantique formelle à la machine d'état d'UML. l'apport de cette approche par rapport aux autres selon [14] et qu'elle fait la distinction entre la sémantique du niveau de définition des besoins et celle du niveau d'implémentation.

- Egalement, l'approche qui fait sujet des articles [19][20][24] propose une méthode basée sur les EHA (Extended Hierarchical Automata) comme un module intermédiaire qui se traduit à la fin à une structure de Kripke. Ces EHAs représentent la sémantique d'un sous ensemble de la machine d'état UML. la sémantique est basée sur trois règles ; progression, composition et stuttering qui simplifient la vérification des machines d'états [20],[19]. Le travail est enrichi en [20] par la prise en charge de plusieurs machines d'état. Ces derniers sont modélisées par des systèmes de transition libellé (LTS : Labeled Transition System). [19] introduit également l'utilisation de l'environnement JACK. Ce dernier est utilisé pour éditer des spécifications LTS qui sont écrites en format FC2. Il peut également vérifier automatiquement ces spécifications en utilisant le model-checker AMC inclut en JACK.

2. **Les Machines d'états Abstraites:** La Machine d'états Abstraite (ASM : Abstract State Machine) est constituée principalement d'un ensemble de règles de mise-à-jour applicable itérativement sur un ensemble d'états [22]. Les ASMs sont utilisées actuellement pour la description opérationnelle des algorithmes [14].

La syntaxe des ASMs est semblable à un simple langage de programmation déclaratif qui rend la machine à la portée des utilisateurs ayant un background de base en programmation.

Bien que la machine d'états abstraite puisse être considérée comme un système de transition comme cité en [23], les auteurs de [14] et [16] maintient les deux formalismes disjoints.

Parmi les approches qui se trouvent dans cette sous-catégorie on a [14] :

L'approche [23] utilise la machine d'états abstraite (ASM) pour représenter la machine d'états d'UML. Elle couvre beaucoup de détails tels que la gestion des événements, l'algorithme 'run-to-completion' et les activités internes…etc. L'approche a donnée une meilleure explication sur la façon à représenter les machines d'états UML par les machines d'états abstraites.

- L'article [24] propose un modèle à base de machine d'état abstraite pour formaliser la machine d'état UML. Ce modèle est utilisé par [25] pour conduire des analyses statistiques et dynamiques sur la machine d'états UML. Il présente un prototype d'un outil de vérification construit à base de XASM (extension de la machine abstraite). L'outil permet aussi de convertir une machine d'état UML à une spécification à base d'ASM pour le model-checking en utilisant les Model-checker SMV. L'un des avantage de cette approche est qu'elle utilise le format XMI () qui rend son outil compatible avec les outils case UML commercialisés.

- [26] est aussi une approche à base de machine d'état abstraite. Elle sépare entre la syntaxe et la sémantique de la machine d'état UML. elle exprime la syntaxe par le langage GTDL (Graph Type Definition Language). Le GTDL est un petit langage dédié qui fait partie de l'environnement MOSES suite [27]. Et la sémantique par des OMA (Object Mapping Automata) [28] qui sont des versions spécialisées des machines d'états abstraites. L'environnement 'Moses Tools suite' permet à ses utilisateurs d'éditer, de simuler, et d'analyser automatiquement ce type de systèmes.

- En [22] les machines d'états abstraites sont utilisées pour donner une sémantique formelle aux machines d'états d'UML. la contribution majeure de cette approche est que les machines d'états peuvent communiquer à travers le passage de messages.

3. **Les Réseaux de Petri**: Les réseaux de Petri (Petri Nets) offrent un formalisme intuitif et bien étudié qui est à la fois graphique et mathématique. Un RdP (Réseau de Petri) est un graphe orienté qui comprend deux types de sommets; les places et les transitions. Ils sont reliés par les arcs orientés. Un arc relie soit une place à une transition, soit une transition à une place.

Une place correspond à une variable d’état du système qui va être modélisé et une transition à un événement et/ou actions qui vont entraîner l’évolution des variables d’états du système. A un instant donné, une place contient un certain nombre de jetons qui va évoluer en fonction du temps: il indique la valeur de la variable d’état à cet instant.

L’évolution d’un RdP correspond à l’évolution de ses jetons au cours du temps (évolution de l’état du système) : il se traduit par un déplacement des jetons ce qui s’interprète consommation/production de ressources déclenchées par les événements ou les actions.

De nombreux outils d’édition et d’analyse des RdPs sont disponibles. Comme il existe diverses extensions pour différents domaines tels que les RDP stochastique pour l’analyse des performances. Dans cette sous-catégorie en trouve:

- L’approche présenté en [29][30] transforme la machine d’état UML à un réseau de Petri stochastique général (GSPN : General Stochastic Petri Net), et le diagramme de séquence à un réseau de Petri stochastique général libellé. En [29], les auteures combinent les deux dans un seul GSPN analysable qui inclue à la fois le comportement de la machine d’état et leurs interactions représentées par les diagrammes de séquence. Ce GSPN peut être analysé et vérifié à l’aide des outils suivants :
    1- GreatSPN [31]: c’est un outil pour modéliser, valider et évaluer les performances d’un système distribué en utilisant les GSPN.
    2- Le modèle PROD [32] : c’est un outil d’analyse et de model-checking pour les réseaux prédicats/transition [14].
    3- L’outil GreatSPIN-to-PROD [33] traduit un GPNS à un réseau de Petri de haut niveau qui sera l’entrée du model-checker PROD.

- l’approche décrite en [34] utilise les Réseaux de Petri pour formaliser la machine d’état UML. le papier représente à l’aide des exemples d’illustration la projection d’un modèle UML à un réseau de Petri. Et tant qu’UML est une notation graphique, la grammaire des graphes est utilisée pour définir la syntaxe et la sémantique. Respectivement à une machine d’état, les états sont modélisés par des places dans le réseau de Petri ou les transitions dans une machine d’état correspondent à des transitions dans le Rdp.
  L’article contient un exemple détaillé basé sur le problème des philosophes, le problème est bien modélisé par plusieurs diagrammes UML. Ces derniers sont projetés à un RdP de haut niveau [14]. Le réseau de Petri peut s’exécuter sur plusieurs outils tels que ‘Design/CPN’ [35] et ‘ARTIS s.r.l. Artifex’ [36]. Comme il peut être analysé et vérifier par Design/CPN et Cabernet [37] pour détecter entre autres ; l’absence de deadlock, l’exclusion mutuelle, la ‘reachabilité’…etc. En fin de compte, c’est l’utilisateur qui décide les propriétés à vérifier dans la spécification UML. Les propriétés sont alors définies à travers des réseaux de Petri. Les résultats de l’analyse sont traduits à la notation UML pour les examiner par l’utilisateur.

- Une autre approche utilise les réseaux de Petri stochastique pour appliquer des analyses de performances sur des modèles UML dont la machine d’état fait partie. L’approche, décrite en [38], utilise le package "Stochastique Petri Net Package" de l’outil SPIN pour transformer le modèle UML à un RdP stochastique [14].

4. **Autres Approches Mathématiques**: Cette sous-catégorie contienne d’autres approches mathématiques non citées ci-dessus. Les approches qui ne se basent ni sur les systèmes de transition ni sur les ASMs ni sur les réseaux de Petri pour formaliser les machines d’états UML. Un exemple de cette sous-catégorie est les approches co-algébriques et les formalismes à base d’ensembles-relations.

Dans cette sous-catégorie on trouve :

- L’approche démontrée en [39] et [40] s’intéresse à la formalisation d’UML en sa globalité. Elle utilise un formalisme mathématique basé sur la théorie des flux et les fonctions de traitement des flux [41]. A fin d’adopter ce formalisme dans l’orienté objet, les auteurs augmente le Framework par des modèles systèmes [40]. Selon [14], l’approche ne donne pas des détails, plutôt elle donne une vue générale sur la formalisation d’UML. l’approche fait partie du projet "UML 2 Semantics Project".

- Le document [42] donne un examen très détaillé à la spécification de la machine d’état d’UML 2. Il défini les machines d’état en terme d’ensemble et des relations. La contribution majeure de cette approche et qu’avec cette examinassions détaillée il est possible de répondre à des questions intéressantes telles que : ils proposent diverses machines d’états avec des transitions conflictuelles avec lesquelles il n’est plus claire d’assigner des propriétés aux transitions selon la spécification donnée.

- [43] décrit une approche qui focalise sur la représentation de la machine d’état d’UML par des co-algèbres. Les auteurs se basent sur l’idée que les systèmes de transitions libellés (LTS) peuvent être représentés par des co-algèbres [44]. Ils appliquent ainsi cette idée sur le travail de [21]. Ainsi, ils donnent une représentation co-algébrique à la machine d’état qui offre une sémantique opérationnelle. La contribution majeure de ce travail et qu’il discute l’équivalence entre des machines d’état et des machines d’état raffinées. Il fournit une vue mathématique à l’équivalence et le raffinement en utilisant différentes lois de raffinement [14].

2.2 Les Systèmes de Réécriture

On s’intéresse dans cette catégorie à deux types de réécriture, la réécriture des termes et la réécriture des graphes. En général, un système de réécriture consiste en un ensemble de règles de réécriture où chaque règle est constituée d’une partie gauche et une partie droite. L’exécution d’un tel système est l’application des règles de réécriture à une configuration donnée. Durant

chaque application, la partie gauche dans une configuration est remplacée par la partie droite de la règle appliquée. L'exécution se termine une fois si aucune des règles ne peut être appliquée.

1. **La Réécriture des Graphes**: La réécriture des graphes, appelé aussi transformation des graphes, fournit une technique de spécification mathématique précise et visuelle. En combinant les avantages des graphes et des règles en un seul paradigme de calcul [16].

   Les approches qui se basent sur la réécriture des graphes sont adéquates pour la formalisation des machines d'états d'UML selon [14].

   - Les auteurs de l'article [45] proposent une approche à base d'un méta-modèle dont le méta-modèle statique (exprimé par des diagrammes de classes) est étendue par un modèle dynamique spécifié par des diagrammes de collaborations. Ces diagrammes spécifient la sémantique opérationnelle d'une machine d'état UML. Les diagrammes de collaborations sont formalisés en sorte de règles de transformation de graphe dont le but et de spécifier une sémantique opérationnelles des machines d'états [45].
     L'utilisation de la réécriture des graphes a donnée une sémantique formelle basée sur des concepts mathématiques rigoureux et encore plus simple à utiliser. L'article selon [14] donne une explication détaillée sur l'approche avec des exemples d'illustration. Toutefois, cette approche ne couvre qu'un petit sous ensemble des éléments de la machine d'état d'UML.

   - L'approche décrite en [46] explique comment normaliser la hiérarchie inhérente de la machine d'état par des graphes en utilisant les règles de transformation des graphes. La technique est très intuitive, spécialement parce que l'utilisateur n'a pas besoin de connaître le formalisme mathématique sous-jacent. [47] poursuit l'application des techniques de transformation des graphes pour définir une sémantique formelle à la machine d'état d'UML. la configuration système est présentée par des graphes et le franchissement des transitions dans la machine d'état correspond à l'application des règles de transformation des graphes.

   - L'approche présentée en [16] combine à la fois la méta-modélisation et la transformation des graphes afin de définir formellement le comportement de la machine d'état. L'approche fait appel aux EHA comme une structure intermédiaire pour présenter la machine d'état. La syntaxe à base des EHA n'est pas primordiale; le formalisme qui utilise les attributs dynamiques et les relations manipulées par les règles de production des graphes, peut directement être appliqué sur le méta-modèle de la machine d'état.
     La contribution majeure de cette approche est qu'elle garde la syntaxe avec les concepts de bonne-structuration de la machine d'état dans le méta-modèle, alors que la sémantique dynamique de la machine est spécifiée par des règles de transformation de graphe [16].

Dans un autre travail, les résultats de l'approche sont testés par l'outil VIATRA [48]. En [49], la sémantique est transformée à une spécification SAL [50] qui peut être utilisée pour la vérification et l'analyse des performances.

2. **La Réécriture des Termes**: Elle est similaire à la réécriture des graphes à l'exception que les règles de réécriture sont appliquées sur des termes au lieu des graphes. Dans le contexte d'une machine d'états, un terme représente une configuration (l'exemple d'un ensemble d'états actifs). Les règles de réécriture décrivent la relation entre les termes (les transitions entre les états).

    - L'approche qui fait l'objet de l'article [51], utilise un système de réécriture conditionnelle des termes et la structure de Kripke, afin de translater la machine d'états d'UML aux SMV-SL, le langage d'entrée du model-checker SMV [52]. Les ensembles d'états actifs sont translatés à des termes, et les libellés des transitions sont traduits à des règles de réécriture conditionnelles [51].
    La sémantique des règles de réécritures conditionnelles est basée sur des structures de Kripke. A la fin, la sémantique est translatée au langage d'entrée du model-checker SMV dans le but d'analyser le comportement de la machine d'état.

2.3 Les Approches de translations

Cette catégorie regroupe les approches qui focalisent sur la translation des machines d'états d'UML à des spécifications formelles par l'utilisation des langages de spécification ou des langages d'entrées aux model-checkers ...etc. Ces approches peuvent être classées aussi dans les deux premières catégories (modèles mathématiques et systèmes de réécriture), néanmoins, ces approches s'intéressent de plus à l'analyse et la vérification automatique du comportement représenté par la machine d'état d'UML.

1. **Les langages de Spécification** : Plusieurs approches tentent à formaliser la machines d'états UML par sa translation en une spécification formelle écrite par des langages formels tels que Z, PVS, CASL, Lotus, Object Z, et RSL.

    - L'approche [53] fournit un schéma général pour la traduction de la machine d'état d'UML en PVS-SL, le langage de spécification formelle de l'environnement PVS. La traduction prend en compte la syntaxe abstraite de la machine d'état et les contraintes de 'bonne structuration'.
    L'avantage d'utiliser le Framework PVS est que ce dernier contient des outils d'analyses rigoureuses des spécifications PVS, y compris la vérification de type, preuves de théorème, et même le model-checking [14]. En bref, l'approche propose une spécification PVS-SL pour un sous ensemble de fonctionnalité de la machine d'état et elle introduit les concepts nécessaires pour formaliser la machine d'états.

- [54] travail sur la formalisation des machines d’état, associées à des classes actives, par l’utilisation d’un système de transitions libellées (LTS) et une spécification algébrique conditionnelle écrite par CASL (Common Algebraic Specification language), qui fait partie du CoFi initiative [55]. Relativement, les machines d’états utilisées sont simples, avec quelque pseudo-état et pas d’états concurrents.
  La grande contribution de cette approche est l’examinassions détaillées de la spécification 1.3 d’UML.
  [56] termine le travail avec les classes actives en utilisant des méta-modèles pour les LTS. Cette approche, appelée « Extreme MetaModelling [57]», utilise GML comme une notation visuelle pour présenter des Méta-Modèles.
  À base des travaux [54] et [58], la Méta-Modélisation de la machine d’état correspond tout simplement à la Méta-modélisation des LTS dont le but est de définir une sémantique formelle à la machine d’état.

- L’approche décrite en [59] et [60] utilise le langage Z pour formaliser la machine d’état d’UML. [60] introduise des schémas-Z (Z-schema) qui spécifient les concepts de la machine d’état. [59] enrichi le travail précédent par l’ajout des discussions sur la génération des cas de testes d’une classe à partir d’une machine d’état.
  Les auteurs présentent les techniques de traduction de la machine d’état au FREE [60]. Les modèles FREE sont formellement vérifiés et bien testés.

2. **Langages de Model-Checking** : La vérification par modèle, ou Model-checking, est une technique bien étudiée d’analyse et de vérification automatique des systèmes représentés par des modèles d’états finis. La logique temporelle est utilisée pour exprimer les propriétés à vérifier.

   Les approches listées dans cette sous-catégorie visent à appliquer la technique de model-checking pour vérifier le comportement spécifiée dans une machine d’état par la translation de la dernière à un langage d’entrée d’un model-checker tels que Promela de SPIN, SMV- SL de SMV et UUPAL …etc.

   L’inconvénient majeur de cette sous catégorie est que le modèle sémantique et le modèle de vérification ne sont pas les mêmes ; Parce que les langages d’entrée aux model-checkers ne sont pas vraiment des langages formels [24],[25]. Par exemple, une approche qui utilise un système de transition libellé (LTS) pour présenter le modèle sémantique du système, et puis elle traduit ce modèle à PROMELA (modèle de vérification). C’est pour ça que les deux modèles, sémantique et celui de vérification, sont différent.

   [24][25], [19][20], [51], [65][66] et [61][62] sont des exemples des approches qui adoptent la technique de model-checking pour la vérification de la machine d’état d’UML.

- [24] utilise les machines d’état abstraites pour présenter le model sémantique de la machine d’état UML. Cette technique est utilisée par [25] dont le but d’appliquer une

analyse statique et dynamique. [25] aussi, introduit un prototype basé sur XASM (extension à la machine d'état abstraite). L'outil assure la vérification de bonne-structuration. Il convertit la machine d'état à une spécification ASM pour la vérifiée en utilisant le model-checker SMV [52].
L'une des aspects les plus importants dans cet outil et qu'il utilise le format XMI, qui lui permet d'être en compatibilité avec les outils case commercialisés de la notation unifiée.

- Les auteurs de l'article [61] proposent une approche de model-checking basée sur la relation entre les différents types de diagrammes UML [2], plus précisément, ils tiennent au compte la relation entre les machines d'états et les diagrammes d'interactions d'UML (collaboration et de séquence). Ils ont développés l'outil HUGO/RT qui vérifie si deux modèles sont consistants [62]. Egalement, l'outil Hugo/RT peut vérifier l'absence de Deadlock et des propriétés complexes exprimées en logique temporelle. L'outil fait recours au SPIN model-checker [63] et son langage d'entrée PREMOLA pour réaliser la vérification. L'outil, aussi, a une sortie vers le model-checker temps réel UPPAAL [64]. Une génération de code JAVA est possible pour chaque machine d'état UML [62].
  L'une des choses intéressante dans cette approche, selon [14], est qu'elle n'est pas basée sur un formalisme mathématique ; les états sont modélisé par des processus PROMELA, avec d'autre processus qui sont chargés de dispatcher les événements et de franchir les transitions [61]. L'approche couvre toutes les caractéristiques des machines d'états UML à l'exception des transitions internes, les événements reportés et les choix.

- Une autre approche de model-checking fait l'objet des travaux [65][66]. C'est une approche à deux phases dont la première est dédiée à la formalisation de la structure de la machine d'état UML. la deuxième phase se focalise sur la formalisation de la sémantique opérationnelle de la machine d'état. L'exécution de la machine d'état d'UML correspond à une machine hypothétique. Cette machine est composé par une file d'attente d'évènements, un mécanisme pour dispatcher les événements et un processeur d'évènements [65]. Chaque objet, dans le modèle UML, exécute l'algorithme run-to-completion [13].
  Cette approche couvre non seulement la majorité des caractéristiques de la machine d'état, mais elle est aussi supportée par l'outil vUML [66]. vUML est un outil de model-checking des machines d'états. Il travail directement sur le modèle dédié à l'analyse, la conception, la documentation et l'implémentation du système logiciel traité [65].
  L'outil n'est pas conçu pour vérifier une seule machine d'état, par contre il est destiné pour vérifier l'interaction, décrite par un diagramme de collaboration, entre plusieurs machines d'états.
- Le travail présenté en [67] présente une méthode très semblable à ce que nous allons introduire dans ce travail. Les auteurs proposent de translater les modèles UML à une spécification formelle en Maude. Ils appliquent ensuite la technique de model-checking

sur les spécifications générées. Néanmoins, et comme les auteurs l'avouent dans la conclusion de leur article, leur méthode est limitée sur l'utilisation des diagrammes d'états-transitions et de communication de base. En pratique, les concepts avancés du diagramme d'états-transitions sont très utilisés dans la modélisation du comportement des objets concurrents.

Dans la suite, nous allons introduire une méthode qui couvrent plus de concepts et de détailles sur les diagrammes d'états-transitions et de classes. Notre méthode aussi, en outre que la vérification des propriétés temporelles, elle permet de vérifier les collaborations et leurs bonnes terminaison vis-à-vis les diagrammes d'états-transitions qui modélisent le comportement des ses objets.

3. **Autre approches de translation:** On trouve dans cette sous-catégorie des approches qui traduisent la machine d'états UML à des expressions régulières concurrentes [68]. Deux approches traduisent la machine d'état à un système axiomatique [69] et [70].

- L'approche qui fait l'objet de l'article [68] propose des règles de transformation pour formaliser la machine d'état en sorte d'expressions régulières concurrentes (CREs : Concurrent Regular Expressions [71]). Les CREs sont des expressions régulières avec plusieurs opérations.
  Une fois la machine d'état est exprimée en CREs, une vérification d'inconsistance "inconsistence checking" peut avoir lieu avec des outils adéquats [68].
- L'approche [70] suggère que l'ambiguïté dans les modèles est UML dû au fait que plusieurs modèles (diagrammes) sont construits pour décrire un seul système. Par ce fait, les auteurs proposent une approche axiomatique qui permet de vérifier la consistance entre les diagrammes, par exemple, entre le diagramme de classe et celui de la machine d'état.

Après ce survol sur les méthodes de formalisation d'UML et précisément son diagramme d'états-transition, nous pouvons conclure que le domaine est ouvert et périodiquement enrichi par des nouvelles méthodes qui visent à donner une sémantique précise et no ambiguë pour les le langage UML en général.

Plusieurs travaux que nous avons vue sont des travaux similaires à ce que nous somme entrain de proposer. Le prochain chapitre est consacré à la logique de réécriture, Maude et son LTL modèle-checker qui servent de bases pour notre approche.

CHAPITRE III :

Logique de réécriture, Maude et Model-Checking

Dans ce chapitre, nous présentons quelques concepts sur les modèles formelles, la logique de réécriture, le langage Maude. Nous allons voir aussi quelques notions de modèle checking tout en focalisant sur le LTL model-checker du langage Maude.

1. Les méthodes formelles

Les méthodes formelles sont les langages, les techniques et les outils qui reposent sur la logique mathématique [72]. L'objectif principal derrière l'utilisation de ces méthodes dans le développement de logiciels est de prouver que les programmes (logiciels) sont correctes, Où les testes et la simulation ne permettent pas de garantir l'absence totale des erreurs qui est généralement le cas avec les grands logiciels.

Afin d'atteindre l'objectif des méthodes formelles et de garantir l'exactitude d'un système, nous devons d'abord spécifier son modèle en utilisant un formalisme donné et puis définir l'ensemble de propriétés que le système doit satisfaire. Ces propriétés (exigence d'exactitude) peuvent être données sous forme de formule logique.

Généralement, il existe deux types de méthodes formelles pour prouver qu'un modèle vérifie un ensemble de propriétés [73]:

- la preuve de théorème
- l'exploration de l'espace de recherche

1.1 Preuve de théorème

Pour appliquer la technique de preuve de théorème nous devons exprimer sous forme de formules logiques le modèle et les propriétés à prouver. Ces formules peuvent être combinées entre elles en respectant des règles définies (un système formel) afin de déduire de nouvelles formules [73]. Les preuves de théorèmes se font ainsi par dérivation à partir d'un ensemble initial de propriétés connues sur le modèle.

Toutefois, le problème qui persiste est l'automatisation totale des constructions de preuves. En pratique, les outils de preuve de théorème ne sont pas entièrement automatiques, ils ont besoin de l'intervention interactive de l'utilisateur et leur mode d'emploi nécessite une bonne

expertise. D'une autre part, si ces preuves permettent d'établir la correction d'un modèle, elle se révèle peut pratique en cas d'erreurs ; si une propriété n'est pas vérifier, il peut être difficile de trouver le comportement du système qui en est la cause [73]

L'avantage de cette approche et qu'elle peut être appliqué sur des modèles paramétrés (prouver une propriété par exemple pour tout n, où n représente le nombre de clients dans un modèle d'application client-serveur).

1.2 Exploration de l'ensemble des états

Les méthodes basées sur l'exploration de l'ensemble des états sont désignée sous le nom modèle-checking (vérification du modèle). Ces méthodes basent sur la modélisation de l'ensemble d'états du système (espace d'états) qui décrivent tous les états accessibles du système ainsi que les liens entre eux [73]. Généralement cette représentation est immense, voir infini, mais l'avantage et qu'elle peut être générée automatiquement à partir d'un modèle de plus haut niveau.

Plusieurs propriétés peuvent être vérifiées sur cet espace d'états. Le rôle de l'utilisateur se réduit à modéliser le système, spécifier les propriétés à vérifier, et finalement lancé le calcul. Les méthodes de modèle-checking, parce qu'elle explore tous l'espace d'état, sont en général capables d'exhiber des contre-exemples lorsque la propriété recherchée n'est pas vérifier. Cette caractéristique, simplifie la vérification et permet le retour sur erreur, ce qui est considéré comme un point fort du modèle-checking vis-à-vis la preuve de théorèmes.

Toutefois, et contrairement aux preuves de théorème, le modèle checking ne considère que des systèmes à ensemble d'états fini. [72] cette restriction est due à l'exploration d'un nombre fini d'états.

C'est à cette méthode (modèle checking) que nous allons s'intéresser dans la suite. Mais avant d'appliquer le modèle checking sur les modèles UML. Nous allons d'abord s'intéressé à la présentation de la logique de réécriture et son langage Maude qui sont le formalisme avec lequel le modèle UML sera translaté, et sur lequel le modèle-checking d'UML est appliqué tout en utilisant le LTL modèle-checker du langage Maude.

2. Logique de Réécriture

2.1 Présentation

La logique de réécriture est introduite par José Méseguar [74] [75] comme une séquence de travaux sur des logiques générales. Cette logique est un cadre sémantique pour spécifier des systèmes concurrents et des langages [76]. C'est également un cadre logique pour présenter et exécuter différentes logiques et langages [77].

La logique en général est une méthode de raisonnement correcte pour quelques classes d'entités [78]. La logique de réécriture permet de raisonner d'une manière aussi correcte sur les

systèmes concurrents non-déterministes ayant des états et qui changent d'états à travers des transitions. Cette logique, qui englobe plusieurs modèles formels qui exprime la concurrence, permet d'expliquer n'importe quel comportement concurrent dans un système comportant une sémantique de vraie concurrence.

Formellement, la logique de réécriture est représentée par une théorie de réécriture $\mathfrak{R} = (\Sigma, E, L, R)$ où Σ est un ensemble d'opérateurs (sorte, symboles de fonction), chaqu'un est associé à un nombre Naturel. E est un ensemble de Σ-équations. La signature (Σ, E) est une théorie équationnelle qui décrit la structure algébrique des états du système. L est un ensemble d'étiquettes et R est un ensemble défini par $R \subseteq L \times (T_{\Sigma,E}(X))^2$ qui est un ensemble de pair tel que son premier composant est une étiquette et le deuxième est une paire de classes d'équivalence des termes avec $X = \{x_1, x_2, \dots, x_n\}$ est un ensemble nombrable et fini de variables x_i.

Les éléments de R sont des règles des réécritures conditionnelles et inconditionnelles, une règle de forme $(r, ([t], [t']))$ est une séquence étiquetée notée par $r : [t] \longrightarrow [t']$. Pour dire que $\{x_1, x_2, \dots, x_n\}$ est un ensemble de variables en t et/ou $[t']$. On écrit $r : [t(x_1, \dots, x_n)] \longrightarrow [t'(x_1, \dots, x_n)]$, où en abréviation $r : [t(\bar{x})] \longrightarrow [t'(\bar{x})]$. [79]

Les règles de réécriture $[t] \longrightarrow [t']$ dans R ne sont pas des équations car au niveau du calcul, elles sont interprétées comme des règles de transition locales dans un système concurrent et logiquement elles sont interprétées comme des règles d'inférence. [78]

Pour une théorie $\mathfrak{R}$, nous disons que la séquence $[t] \longrightarrow [t']$ est prouvable dans $\mathfrak{R}$ et nous écrivons $\mathfrak{R} \vdash [t] \longrightarrow [t']$ Si seulement si $[t] \longrightarrow [t']$ peut être obtenu par l'application finie des règles de déduction suivantes [79]:

- **Réflexivité** : pour chaque $[t] \in T_{\Sigma,E}(X)$,

$$\overline{[t] \longrightarrow [t']}$$

- **Congruence** : pour chaque $f \in \Sigma_n,\ n \in \mathrm{N}$, $\dfrac{[t_1]\longrightarrow[t'_1] \ \dots \ [t_n]\longrightarrow[t'_n]}{[f(t_1,\dots,t_n)] \longrightarrow [f(t'_1,\dots,t'_n)]}$

- **Remplacement :** pour chaque règle de réécriturer $r : [t(x_1, \dots, x_n)] \longrightarrow [t'(x_1, \dots, x_n)]$ dans R, $\dfrac{[w_1]\longrightarrow[w'_1] \dots [w_n]\longrightarrow[w'_n]}{[t(\overline{w} / \bar{x})] \longrightarrow [t'(\overline{w'} / \bar{x})]}$ tel que $t(\overline{w} / \bar{x})$ indique la substitution simultanée des w_i pour x_i dans t.

- **Transitivité** : $\dfrac{[t_1]\longrightarrow[t_2] \qquad [t_2]\longrightarrow[t_3]}{[t_1]\longrightarrow[t_3]}$.

La logique équationnelle (modulo un ensemble d'axiome E) est obtenue à partir de la logique de réécriture par l'ajout de la règle suivante [79]:

- **Symétrie** : $\dfrac{[t_1]\longrightarrow[t_2]}{[t_2]\longrightarrow[t_1]}$.

Avec cette règle, les séquences dérivables dans la logique équationnelle sont bidirectionnelles. Ainsi, nous pouvons utiliser cette notation $[t] \leftrightarrow [t']$ et nous appelons cette séquence bidirectionnelle une équation. [79]

Les unités de base de spécification dans la logique de réécriture sont appelées des théories qui sont de deux types : théories fonctionnelles et théories systèmes.

- **Les théories fonctionnelles** sont des théories dans la logique équationnelle d'appartenance appelée MEL(MemberShip Equational Logics) qui est équivalente à la logique de Horn. Les séquences en MEL sont des égalités $t = t'$ ou bien des assertions d'appartenance de la forme $t{:}s$ exprime que t est de sorte s. Une telle logique étend la logique équationnelle des sortes ordonnées (order-sorted) et contient ainsi les sortes, les relations sous-sorte, la surcharge des opérateurs, et la définition des opérations.
- **Les théories systèmes** quand à elles spécifient le modèle initial $\mathfrak{R}0$ d'une théorie de réécriture $\mathfrak{R}$, ce modèle initial est un système de transition dont :
  - les états sont des classes d'équivalence $[t]$ des termes t modulo les équations E définies dans $\mathfrak{R}$,
  - les transitions sont des $\alpha - prouves$ (-preuve) de la forme : $[t] \leftrightarrow [t']$.

Dans la suite nous détaillons les différents concepts utilisés dans les théories équationnelles et systèmes.

2.2 Concepts de bases des théories

Dans cette section, nous allons présenter les concepts de base de la logique de réécriture qui sont implémenté dans des langages de réécriture comme obj3 [80] et Maude. L'exemple qui se trouvent dans cette section sont écris en Maude qui sera détaillé juste après. Les éléments de base de la logique de réécriture sont : [78]

1. **Sortes :** les sortes ou les types sont les premières choses à définir dans une spécification, ce sont partiellement ordonnées via une relation de sous-sorte. Une sorte est déclarée en utilisant le mot **sort** suivi d'un identificateur (le nom de sorte) suivi d'un espace et un point :

   Sort <Sort_Name> .

   Les sortes multiples sont déclarées en utilisant le mot-clé **Sorts :**

   Sorts <Sort_Name1> …. <Sort_Name N> .

2. **Sous-sortes** : la relation de sous-sorte **subsort** sur les sortes est équivalente à une relation de sous ensemble (relation d'inclusion). Les sous-sortes sont déclarées en utilisant le mot-clé **subsort** ou **subsorts.** La déclaration est sous forme :

   Subsorts <Sort1> < <Sort2> .

Cette déclaration signifiée que **<Sort1>** est une sous-sorte de **<Sort2>.** Un exemple de déclaration de sous sorte :

Subsort Zero < Nat .
Subsort NzNat < Nat .

Nous pouvons aussi déclarer plusieurs relations de sous sorte en utilisant **Subsorts :**

Subsorts <Sort1> … <Sort2> < … < <Sortn>

Pour l'exemple précédemment cité on peut le déclaré comme suit :

Subsort NzNat Zero < Nat .

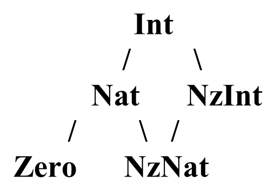
Un autre exemple pour illustrer l'utilisation de subsort :

Sorts NzInt Int .
Subsorts NzNat < NzInt Nat < Int .

Où NzInt et Int sont les ensembles des entiers naturels non nuls et des entiers.

Un ensemble de déclarations de sous-sortes définit un ordre partiel sur l'ensemble de sortes. Et afin que la déclaration soit correcte, l'utilisateur doit éviter les cycles dans les déclarations de sous-sortes.

Nous pouvons visualiser par un graph l'ordre partiel de déclaration de sous-sorte de la façon suivante :



3. **Opérations :** les opérations sont déclarées à l'aide du mot-clé **op** suivi de nom d'opération, suivi par deux points, suivi d'une série de sortes qui constituent ses arguments (arité), suivi de →, suivi des sortes des résultats (co-arité), optionnellement suivis de la déclaration d'attributs, suivi d'un espace et point. Le schéma général a la forme suivante :

Op <OpName> : <sort0> ... <sortk> → <sort> [<operatorattributes>] .

Un exemple de déclaration des oéprations est le suivant : [81]

Op zero : → Zero .
Op s_ : Nat → NzNat .
Op sd : Nat Nat → Nat .
Op _+_ _*_ : Nat Nat → Nat.

En cas ou l'arité de l'argument est vide, l'opération est appelée une constante comme c'est le cas pour **zero**. Le nom d'opération est une chaîne de caractères les espaces soulignés (_) jouent un rôle spécial dans ces chaînes de caractères. En effet, si aucun espace souligné ne se

produit dans la chaîne de caractères de l'opération comme dans le cas de **sd** alors l'opérateur est déclaré sous la forme préfixée. Si les espaces soulignés se produisent dans la chaîne de caractères, alors le nombre doit coïncider avec (le nombre de doit coïncider) avec le nombre de sortes déclarées comme arguments d'opérateur. L'opérateur est alors sous la forme dite mixfix où le nième espace souligné indique la position de l'argument de la nième sorte dans les expressions formées avec cet opérateur. Dans l'exemple cité en dessus les opérateurs s_, _+_, et _*_ sont sous la forme mixfix. [78]

Des opérations ayant les mêmes arité et co-arité déclarées simultanément en employant **ops** comme mot-clé et en donnant les sortes non vides après le mot-clé **ops** et avant deux points, comme pour les déclaration de _+_, _*_ dans l'exemple plus haut.

4. **Surcharge d'opérateurs :** dans la logique de réécriture, les opérateurs peuvent être surchargés, c'est-à-dire que nous pouvons avoir plusieurs déclarations d'opérations pour le même opérateur avec différents arités et co-arités. Voici un exemple sur la surcharge d'opérateur dans la logique de réécriture :

```
Op _+_ : NzNat Nat → NzNat .
Sort Nat3 .
Ops 0 1 2 : →Nat3 .
Op _+_ : Nat3 Nat3 → Nat3 .
```

L'opérateur _+_ est surchargé et à trois déclarations (une dans l'exemple cité auparavant). Cependant, il y a deux types de surcharge dans l'exemple. La déclaration de _+_ avec le premier argument NzNat est un exemple de la surcharge de subsort où les deux opérateurs sont sensés avoir le même comportement [78].

5. **Variables :** dans la logique de réécriture, les variables sont déclarés à tout moments avec une syntaxe comportant un identificateur (le nom de la variable), deux point et un autre identificateur (sa sorte). Par exemple A: Nat déclare une variable nommée **A** de sorte **Nat**. Des variables de la même sorte peuvent être déclarées en utilisant le mot-clé **vars** :

vars M N : Nat .

6. **Théories Fonctionnelles :** à travers les théories fonctionnelles, on définit les sortes de données et les opérateurs sur ces données. Les sortes de données se composent des éléments qui sont utilisés par les termes. Par définition, deux termes dénotent le même élément si et seulement si ils appartiennent à la même classe d'équivalence déterminée par les équations.

Nous supposons que les théories fonctionnelles vérifient la propriété suivante : l'application répétée des équations comme règles de simplification atteint par la suite un terme à laquelle aucune autre équation ne s'applique. Le résultat, appelé forme canonique est toujours le même quelque soit l'ordre de l'application des règles de simplification. Ainsi chaque classe d'équivalence a un représentant normal, sa forme canonique, qui peut être calculé par simplification équationnelle. [78]

La déclaration d'une théorie fonctionnelle suit la syntaxe suivante :

```
fmod  NOM_TF is
. . . . . .
endfm
```

La théorie fonctionnelle **NOM_TF** est déclarée par le mot **fmod** suivi par le nom de la théorie suivi par **is**. Les points représentent la déclaration et les expressions qui peuvent apparaître dans la théorie fonctionnelle. Les expressions incluent des axiomes équationnels et d'appartenance.

7. **Equation inconditionnelles:** La déclaration des équations dans une spécification à base de logique de réécriture se fait à l'aide du mot-clé **eq**. Suivi par deux termes séparé par le signe d'égalité (=). Et optionnellement, après le deuxième terme par des attributs et à la fin un espace et un point. La syntaxe générale de cette déclaration est la suivante :

   eq <Terme1> = <Terme2> [<StatementAttributes>] .

   Les termes **Terme1** et **Terme2** dans l'équation doivent être de même sorte. Chaque variable qui apparaît dans la partie gauche doit apparaître aussi dans la partie droite. [78]

   L'exemple suivant montre l'utilisation d'équations dans une théorie fonctionnelle des nombres:

```
Vars N M : Nat .
eq N + zero = N .
eq N + s M = s ( N + M) .
```

8. **Axiomes d'appartenance inconditionnelle** : les axiomes d'appartenance inconditionnelle (unconditional memberships) sont déclarés avec le mot-clé **mb** suivi par un terme. Suivi par deux points, suivi par une sorte, suivi par un point. Les axiomes d'appartenance peuvent aussi optionnellement avoir des attribues désignés par <statements>. [82]

   mb <Term> : <Sort> [<StatementAttributes>] .

   Comme un exemple d'illustration d'utilisation de ces axiomes, considérons la théorie 3*NAT avec la déclaration des nombres de base " peano". Une nouvelle sorte 3*Nat. En effet, 3*Nat correspond aux multiples de 3 et est exprimée en utilisant déclaration de sous-sorte suivant : **Zero < 3*Nat < Nat** et l'expression d'appartenance **mb(s s s M3) : 3*Nat** pour **M3** une variable de sorte **3*Nat**. [81]

```
fmod 3*NAT is
sort Zero Nat .
subsort Zero < Nat .
op zero : -> Zero .
op s_ : Nat -> Nat .
sort 3*Nat .
```

```
subsorts Zero < 3*Nat < Nat
var M3 : 3*Nat .
mb (s s s M3) : 3*Nat
endfm
```

9. **Equation et Axiomes d'appartenance conditionnelle :** les conditions dans les équations conditionnelles et d'appartenance se composent de différentes équations T=T0 et d'appartenance T : S. une condition peut être une équation simple, une appartenance simple, ou une conjonction d'équation et d'axiomes d'appartenance en utilisant la conjonction binaires de liaison / \ qui est associative.

   Ainsi la forme générale des équations conditionnelles (ceq) et des axiomes d'appartenance (cmb) est la suivante :

   ceq <term-1> = <term-2>
   if <EqCondition-1> / \ … / \ <EqCondition-2> [<StatementAttributes>] .

   cmb <term> : sort >
   if <EqCondition-1> / \ … / \ <EqCondition-2> [<StatementAttributes>] .

   La syntaxe concrète de la condition équationnelle " **EqCondition**" à trois formes:

   - équation ordinaire t=t'
   - équation d'affectation t := t'
   - équation booléennes dites abrégées de la forme t, avec t un terme dans la sorte Bool abrégeant l'équation t= true.

   Ainsi, ces conditions peuvent apparaitre dans une équation :

   (N = = zero) = true
   (M =/= s Zero) = true
   (N > zero or M =/= s zero) = true

10. **Confluence et terminaison des équations :** la logique de réécriture ne spécifie pas dans quel ordre les équations dans une théorie seront appliquées en calculant la forme normale d'un terme. Par conséquent les équations doivent être confluentes et mener à une terminaison pour plus de détails veuillez consulter [81].

11. **Attributs :** la logique de réécriture permet de renforcer la déclaration des opérateurs pas des attributs qui fournissent des informations additionnelles à l'opérateur; des informations sémantique, syntaxique, pragmatique …etc. les attributs sont déclarés dans une seule paire de crochets après la sorte du résultat et avant le point. [82]

    Parmi les variantes d'attributs on trouve :

- **Attributs équationnels (equational attributes):** les attributs équationnels permettent de déclarer certains genres d'axiomes qui permettent à la logique de réécriture d'utiliser les équations d'une manière intégrée [78]. La logique de réécriture supporte les attributs équationnels suivants:

  - **assoc** (associativité),
  - **comm** (commutativité),
  - **idem** (idempotence)
  - **id : <Term>** (identité)

  Un opérateur peut être déclaré avec n'importe quelle combinaison de ces attributs qui peuvent apparaître dans n'importe quel ordre dans la déclaration.

  Du point de vue fonctionnement, l'utilisation des attributs équationnels évite des problèmes d'arrêt et mène à une évaluation beaucoup plus efficace des termes contenant cette opération. En fait, l'effet de déclarer des attributs équationnels permet de calculer les classes d'équivalences modulo les équations. Par exemple, si une équation de commutativité, x + y = y + x, est déclarée comme équation ordinaire, elle produira facilement une boucle infinie de simplification. Si on la déclare avec l'attribut **comm**, ainsi la boucle ne se produit pas. Dans l'exemple du nombre cité avant, nous pouvons ajouter **nil** pour exprimer le vide et raffiner la déclaration de la concaténation des séquences des nombres, de sorte que la concaténation soit associative avec l'identité nil. [78]

  ```
  op nil :  → Natseq .
  op __ : Natseq  Natseq → Natseq [assoc id : nil] .
  ```

- **Constructeurs :** si on suppose que les équations dans une théorie fonctionnelle se terminent. Ainsi chaque terme dans la théorie (un terme sans variables) sera simplifie dans une forme canonique modulo certains attributs fonctionnels. On appel les opérateurs qui apparaissent sous telle forme des constructeurs. Dans la logique de réécriture il est important de dire quand un opérateur est un constructeur. Ceci peut être fait avec l'attribut **ctor.** Un exemple sur la déclaration des constructeurs est le suivant :

  ```
  op zero : → zero [ctor] .
  op s_ : Nat → NzNat [ctor] .
  op __ : Natseq  Natseq → Natseq [ctor assoc] .
  ```

D'autres attribues sont utilisés dans la logique de réécriture et Maude, voir [82] pour plus de détails.

12. **Théories systèmes :** une théorie système spécifie une théorie de réécriture dans la logique de réécriture. Une théorie de réécriture a des sortes, des opérateurs, et peut avoir des équations, des axiomes d'appartenance et des règles de réécriture qui peuvent être conditionnelles [83]. Par conséquent, une théorie de réécriture n'est en effet qu'une théorie équationnelle

fondamentale, contenant des équations et des axiomes d'appartenance, plus les règles de réécriture.

Une théorie système est déclarée comme suit :

mod <modulename> is <declaration and statements> endm

par exemple :

```
mod refiner is
….
endm
```

Dont les points correspondent à toutes les déclarations et expressions dans une théorie :

- Importation des modules
- Sortes et sous sortes
- Opérations
- Variables
- Équations et axiomes d'appartenances (conditionnelles et non conditionnelles).
- Règles de réécriture (conditionnelles et non conditionnelles).

Nous remarquons que les éléments dans une théorie système sont les même que celles de théorie équationnelle à l'exception des règles de réécriture.

13. **Règle inconditionnelles de réécriture:** dans la logique de réécriture, le comportement dynamique est modélisé par des règles de réécriture. Une règle de réécriture peut être vue comme une équation à sens unique, et de ce fait, la logique de réécriture est une logique équationnelle sans symétrie [81].

Mathématiquement, une règle inconditionnelle de réécriture à la forme L : $t \rightarrow t'$ Où t et t' sont des termes de même type et qui peuvent avoir des variables, et L est l'étiquette de la règle. Intuitivement, une règle de ce type décrit une transition concurrente locale dans un système. La déclaration d'une règle inconditionnelle suit la syntaxe suivante :

rl [<label>] : <Term-1> => <Term-2> [<StatementAttributes>] .

Comme un exemple sur les théories système et l'utilisation des règles de réécriture, supposant qu'on veut spécifier une machine de vente (Vending machine) qui dispose des pommes (apples) et des tartes (Cakes) [82]. La théorie fonctionnelle **vending-machine-signatures** est la théorie de base sur laquelle repose la théorie système. Cette théorie est importée par la théorie système **vending-machine** qui ajoute des règles de réécriture qui rend la machine opérationnelle.

Les constants **$** et **q** représentent respectivement une pièce de monnaie d'un dollars et une pièce d'un quart de dollar. Et les constants **a** et **c** représentent respectivement une pomme (**a**pple) et une tarte (**c**ake).

```
fmod VENDING-MACHINE-SIGNATURE is
sorts Coin Item Marking .
subsorts Coin Item < Marking .
op __ : Marking Marking -> Marking [assoc comm id: null] .
op null : -> Marking .
op $ : -> Coin [format (r! o)] .
op q : -> Coin [format (r! o)] .
op a : -> Item [format (b! o)] .
op c : -> Item [format (b! o)] .
endfm
```

L'attribut format permet d'afficher les constants **$, q, a, c** en couleurs.

```
mod VENDING-MACHINE is
including VENDING-MACHINE-SIGNATURE .
var M : Marking .
rl [add-q] : M => M q .
rl [add-$] : M => M $ .
rl [buy-c] : $ => c .
rl [buy-a] : $ => a q .
rl [change] : q q q q => $ .
endm
```

Ces deux théories spécifient une machine concurrente qui vende des pommes et des tartes. Une tarte coûte un dollars et une pomme 3 quarts. On peut insérer des pièces de dollars ou d'un quart. A cause d'un défaut dans la fabrication de la machine, la dernière n'accepte que des pièces d'un dollars, donc l'utilisateur doit déposer un dollar s'il veut acheter une pomme et la machine lui rend un quart. Pour surmonté ce problème, la machine peut changer 4 quarts à un dollar.

La machine est concurrente, parce qu'on peut appuyer sur plusieurs boutons à la fois (c'est pourquoi on peut appliquer plusieurs règle de réécriture en même temps et on dit que la machine est concurrente). Si on a un dollars dans la place $ et 4 quarts dans la place q, on peut simultanément appuyer sur le bottons **buy-a** et **change** et la machine renvoi simultanément aussi un dollar dans la place **$**, une pomme dans la place **a**, et un quart dans la place **q**.

14. **Règle conditionnelles de réécriture**: les règles conditionnelles de réécriture peuvent avoir des conditions très générales impliquant dans des équations, des axiomes d'appartenances et d'autres réécritures. Mathématiquement, les règles conditionnelles de réécriture ont la forme suivante : [82]

$$l : t \rightarrow t' \quad if \quad (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \rightarrow q_k)$$

La représentation des règles de réécriture conditionnelle en logique de réécriture suit la syntaxe suivante:

crl [<etiquette>] : <Term-1> => <Term-2> if <condition-1 /\ .. /\ condition-2> [<StatementAttributes>] .

La condition peut se composer d'une expression simple ou peut être une conjonction avec le connectif associatif /\.

L'exemple suivant modélise la vie d'une personne. Un état dans la vie d'une personne est représenté par le terme **person(name, age, status)** ou **status** (état) peut être **single** (célibataire) **engaged** (engagé), **married** (marié), **separated** (séparé) , **divorced** (divorcé), **widow** (veuve), **widower** (veuf), or **deceased**(décédé). Et l'**age** dénote l'âge de la personne.

Un exemple d'état peut être : **person("Walid", 58, married)** . La théorie système qui modélise cet exemple est la suivante : [72]

```
mod ONE-PERSON is
protecting NAT .
protecting STRING .
sorts Person Status .
op person : String Nat Status -> Person [ctor] .
ops single engaged married separated divorced deceased
widow widower : -> Status [ctor] .
var X : String . var N : Nat . var S : Status .
crl [birthday] :
person(X, N, S) => person(X, N + 1, S) if N <= 1000 /\ S =/= deceased .
crl [successful-proposal] :
person(X, N, S) => person(X, N, engaged)
if N >= 15 /\ (S == single or S == divorced) .
rl [marriage] : person(X, N, engaged) => person(X, N, married) .
...
Endm
```

Les règles [**birthday**] et [**successful-proposal**] sont des exemples de règle conditionnelle de réécriture.

3. Langage Maude

3.1 Présentation

Maude [83] [84] est un langage formel de spécification et de programmation déclarative basé sur une théorie mathématique de la logique de réécriture [81]. La logique de réécriture et le langage Maude sont développés par Jose meseguar et son groupe dans le laboratoire d'informatique en SRI Inetrnational. Maude est un langage simple expressif et performant, il est considéré parmi les meilleurs langages dans le domaine de spécification algébrique et la modélisation des systèmes concurrents [72].

Maude spécifie des théories de la logique de réécriture, les types de données sont définies algébriquement par des équations et le comportement dynamique du système est définie par des règles de réécritures qui décrivent comment une partie d'un état est changé dans un pas. Maude supporte aussi la programmation orientée objet avec l'inclusion de l'héritage multiple et la communication asynchrone par le passage des messages. [81]

L'interpréteur Maude exécute les programmes équationnels écrit en Maude en commençant par une expression initiale. Ensuite on appliquant les équations " de gauche à droite " jusqu'à ce qu'aucune équation ne puisse être appliquée. L'interpréteur exécute des programmes de réécriture par l'application " arbitraire" des règles de réécriture (aussi de gauche à droite) sur l'expression ou l'état initial jusqu'à ce qu'aucune règle ne soit applicable. Ou à un nombre de réécriture donné par l'utilisateur. Dans le cas des réécritures, les équations sont appliquées pour réduire chaque état intermédiaire à sa forme normale avant d'appliquer les règles de réécriture.

Le groupe de Maude ont focalisé leurs efforts aussi sur la performance, la version actuelle de l'interpréteur peut atteindre des millions de réécriture par seconde. Par ce fait, Maude est en concurrence avec les langages de haut niveau en termes d'efficience [72].

Une théorie de réécriture est généralement non déterministe et peut exhiber différent comportements. La commande **rewrite** du Maude peut être utilisée pour exécuter un seule comportement à partir d'un état initial. Pour analyser tous les comportements possibles à partir d'un état initial, on peut utiliser la commande **search** de haut performance et/ou le modèle checker de Maude qui est comparable en termes d'efficience avec d'autres modèle-checker [85]. Les spécifications Maude peuvent être aussi analysées par le calcul méta-niveau du Maude [81] [86].

Maude contient aussi un support d'utilisation des sockets pour la programmation réseau, ce qui permet non seulement de modéliser, de simuler et d'analyser des systèmes concurrents, mais aussi de les programmés [72].

Pour spécifier un système concurrent, Maude offre trois types de modules :

- Modules fonctionnels
- Modules systèmes
- Modules orienté-objet, ce sont définis uniquement dans full Maude.

3.2 Modules Fonctionnelles

Les modules fonctionnels sont des théories fonctionnelles [83] Que nous avons présentées précédemment. Ces modules définissent les types de données et les opérations qui sont utilisés par les équations.

L'algèbre initiale sous jacente est un modèle mathématique dénotationel pour les sortes et les opérations. Les éléments de cette algèbre sont des classes d'équivalence des termes sans variables (ground terms [75]) modulo les équations. Si deux termes sans variables sont égaux par une équation, on dit qu'ils appartiennent à la même classe d'équivalence [79]. Les équations sont utilisées comme des règles de réductions. À la fin du calcul, chaque règles est évaluée à sa forme réduite dite représentation canonique [79]. La représentation canonique est unique et elle représente tous les termes de la même classe d'équivalence.

Les équations dans un module fonctionnel sont orientées, elles sont utilisées de gauche à droite, le résultat de réduction est unique comme nous avons dit quelques soit l'ordre dans lequel les équations sont appliquées.

Les modules fonctionnels supportent aussi les axiomes d'appartenance (membership axioms) [83] ces axiomes précisent l'appartenance d'un terme à un type. Les axiomes peuvent être conditionnels ou inconditionnels. En cas des axiomes conditionnelles, les conditions sont des jonctions des équations et des testes d'appartenance inconditionnels. [72]

En Maude, une spécification équationnelle est un module fonctionnel qui est représenté par la syntaxe suivante [82]

```
fmod MODULENAME is
BODY
endfm.
```

Où MODULENAME est le nom de module introduit, et BODY est l'ensemble de déclarations des sortes, d'opérations, des variables, des équations, des axiomes d'appartenance et des commentaires. Les commentaire commencent par *** ou ---- et se termine par la fin de la ligne courante ou elles commencent par ***(ou ---(et se termine par l'occurrence de ")".

Un exemple d'un module fonctionnel des nombres naturels est le suivant : [81]

```
fmod BASIC-NAT is
sort Nat .                                   *** une sorte
op 0 : -> Nat [ctor] .                       *** une opération
op s_ : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat .
op max : Nat Nat -> Nat .
vars N M : Nat .                             *** deux variables
eq 0 + N = N .                               *** une équation
eq s M + N = s (M + N) .
eq max(0, M) = M .
eq max(N, 0) = N .
eq max(s N, s M) = s max(N, M) .
endfm
```

3.3 Modules Systèmes

Les modules systèmes sont des théories de réécriture [83]. Ils permettent de spécifier le comportement d'un système concurrent. Les modules systèmes ajoutent à la définition des modules fonctionnels les règles de réécriture qui peuvent être conditionnels et inconditionnels. L'introduction des règles de réécriture permet d'exprimer la concurrence dans les systèmes.

Un module système décrive une théorie de réécriture qui inclue des sortes, des opérations, des variables, des équations, des axiomes d'appartenances (conditionnelles et inconditionnelles) et des règles de réécriture conditionnelles et inconditionnelles.

Une règle de réécriture s'exécute quand sa partie gauche correspond (match) une portion dans l'état global du système et avec la satisfaction de la condition en cas d'une règle conditionnelle [79].

En Maude, une spécification est un module fonctionnel qui est représenté par la syntaxe suivante [82] :

```
mod MODULENAME is
BODY
endfm.
```

Où **MODULENAME** est le nom de module introduit, et **BODY** est l'ensemble de déclarations des sortes, d'opérations, des variables, des équations, des axiomes d'appartenance, et des règles de réécriture. Un exemple qui montre l'utilisation de ces modules est le suivant: [72]

```
mod CHOICE-INT is
including INT .
op _?_ : Int Int -> Int .
vars I J : Int .
rl [choose_first] : I ? J => I .
rl [choose_second] : I ? J => J .
endm
```

3.4 Spécifications Orientées Objet en Maude

En effet, la plupart des spécification réelles dans Maude sont des spécifications orientées objets, [87][88][89][90] sont des exemples concrets de spécification des systèmes.

Les objets concurrents peuvent être représentés naturellement et directement sur le Core-Maude par le bais des modules systèmes et fonctionnels. Néanmoins, le groupe qui à développé Maude ont étendu le Core-Maude au full Maude qui est dédié à la spécification orientée objet [82].

Le full Maude est un prototype Maude qui supporte la spécification orientée objets. Il est spécifié est programmé par Francisco Durân [91]. Le Full Maude donne un support pour la spécification orientée objets se forme des modules orientées objets. Ces derniers offrent la syntaxe nécessaire pour déclarer les classes, les sous classes et les messages. [81]

Afin, d'augmenter la vitesse des réécritures, le Full Maude permet de cacher les attributs de classes qui n'influent pas dans l'exécution d'une règle de réécriture, et qui ne sont pas effectués par l'exécution de la règle [72]. L'exécution du full Maude est faite par la translation des modules orientés objet aux modules Maude ordinaires (Core Maude).

1. **Modules Orientés Objets :** les modules orientés objets en Maude sont déclarés avec la syntaxe suivante :

```
(omod  ModuleName    is
…..
endom)
```

Les modules systèmes et fonctionnels peuvent être utilisés en Full Maude. Il faut juste les mètre entre deux parenthèses ().

- **Configuration :** l'état concurrent d'un système orienté objet est appelé une configuration. Une configuration en logique de réécriture est vue comme un multi-ensemble (multi-sets) qui est composé d'objets et de messages. La configuration à la forme suivante :

_ _ : Configuration Configuration -> Configuration.

Les opérateurs _ _ est déclarés pour satisfaire les lois structurelles d'associativité et de commutativité et d'élément d'identité. Les objets et les messages dans un multi-ensemble sont des sous sorte de la configuration.

Object Message < Configuration.

Les configurations les plus complexes sont générées de l'union de ces multi-ensembles.

Les sortes **Oid, Object**, **Msg** et **Configuration** sont définies dans le module Configuration qui se trouve dans le module prelude.maude qui est automatiquement importé dans les modules orienté objet [82].

```
mod CONFIGURATION is
sorts Attribute AttributeSet .
subsort Attribute < AttributeSet .
op none : -> AttributeSet .
op _,_ : AttributeSet AttributeSet -> AttributeSet
[format (o m so o) ctor assoc comm id: none] .
sorts Oid Cid Object Msg Portal Configuration .
subsort Object Msg Portal < Configuration .
op <_:_|_> : Oid Cid AttributeSet -> Object
[ctor object format (b r b g b o b o)] .
op none : -> Configuration .
op __ : Configuration Configuration -> Configuration
[format (o n o) ctor config assoc comm id: none] .
op <> : -> Portal [ctor] .
endm
```

La sorte **Cid** dénote l'identificateur d'une classe et la sorte **AttributeSet** représente un multi-ensemble d'attributs.

- **Règles de réécriture pour l'objet :** l'associativité et la commutativité d'une configuration multi-ensembles rend la dernière plus flexible. Elle est vue comme une soupe dans laquelle les objets et les messages se flottent. Ces derniers peuvent être ensemble à chaque instant pour participer à une transaction concurrente [81].

  En général, une règle de réécriture dans R qui décrit le dynamique d'un système orienté objet peut avoir la forme :

$$
\begin{aligned}
& \boldsymbol{r} : \boldsymbol{M_1} \ldots \boldsymbol{M_n} \ < \boldsymbol{O_1}\!: \boldsymbol{F_1} \mid \boldsymbol{atts_1} > \cdots \ < \boldsymbol{O_m}\!: \boldsymbol{F_m} \mid \boldsymbol{atts_m} > \\
& \rightarrow < \boldsymbol{O_{i1}}\!: \boldsymbol{F'_{i1}} \mid \boldsymbol{atts'_{i1}} > \cdots \ < \boldsymbol{O_{ik}}\!: \boldsymbol{F'_{ik}} \mid \boldsymbol{atts'_{ik}} > \\
& \quad < \boldsymbol{Q_1}\!: \boldsymbol{D_1} \mid \boldsymbol{atts''_1} > \cdots \ < \boldsymbol{Q_p}\!: \boldsymbol{D_p} \mid \boldsymbol{atts''_p} > \\
& \quad \boldsymbol{M'_1} \ldots \boldsymbol{M'_q} \\
& \quad \Leftarrow \boldsymbol{C}
\end{aligned}
$$

  Où : r est l'étiquette de la règle, M_s sont des messages, $i_1, \ldots, i_k$, sont les différent numéros des objets, et C c'est la condition.

  Un exemple qui illustre l'utilisation des modules orientés objet est le suivant : [72]

```
load full-maude
(omod POPULATION is
protecting NAT .
protecting STRING .
sort Status .
op single : -> Status [ctor] .
ops engaged married separated : Oid -> Status [ctor] .
subsort String < Oid .
class Person | age : Nat, status : Status .
vars N N' : Nat . vars X X' : String .
crl [birthday] : < X : Person | age : N > => < X : Person | age : N
+ 1 > if N < 999 .
crl [engagement] : < X : Person | age : N, status : single > < X'
: Person | age : N', status : single > => < X : Person | status :
engaged(X') > < X' : Person | status : engaged(X) > if N > 15 or
N' > 15 .
op initState : -> Configuration .          *** etat initial
eq initState = < "Peter" : Person | age : 37, status : single > <
"Lizzie" : Person | age : 34, status : single > < "Sam the Snake"
: Person | age : 40, status : single > .
endom)
```

Dans le prochain chapitre, nous allons voir comment on utilise ces concepts dans la traduction des modèles UML vers des spécifications Maude.

3.5 Exécution du Maude

L'interpréteur Maude est gratuit. Il est sous la licence de GNU et téléchargeable à partir de Site de l'équipe Maude : http://maude.cs.uiuc.edu.

Une session Maude peut être démarré par l'invocation du fichier binaire maude.linux dans le répertoire maude-linux/bin sur le shell linux. (C'est presque la même chose avec Ms-Dos). Nous avons choisi linux mandriva pour le reste du travail et la figure (Fig.42) représente une session ouverte de Maude.

Durant une session Maude, l'utilisateur interagir avec l'environnement par la saisie des commandes Maude (Maude prompts). L'utilisateur peut saisir les modules directement en prompt. Mais il est très pratique d'écrire ces modules avec un éditeur de texte et les sauvegarder dans des fichiers. Ces derniers sont appelés du prompt Maude par la commande **In** ou **laod**.

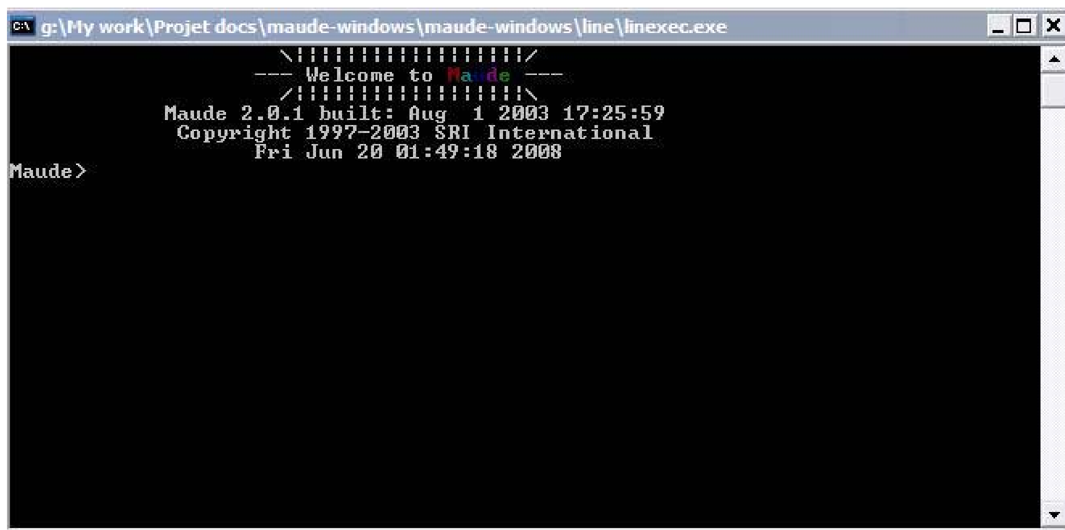


Fig.42 : session ouvert de Maude sous Ms-DOS

Supposons qu'on veut exécuter l'exemple de VENDING-MACHINE que nous avons vue précédemment. Voici quelques commandes et leur exécution: [82]

```
Maude> show sorts VENDING-MACHINE .
sort Bool .
sort Coin .                subsort Coin < Marking .
sort Item .                subsort Item < Marking .
sort Marking .             subsorts Item Coin < Marking .

Maude> show rls VENDING-MACHINE .
rl M => q M [label add-q] .
rl M => $ M [label add-$] .
rl $ => c [label buy-c] .
rl $ => q a [label buy-a] .
rl q q q q => $ [label change] .

Maude> rew [1] in VENDING-MACHINE: $ $ q q .

rewrite [1] in VENDING-MACHINE : $ $ q q .
rewrites: 1 in 0ms cpu (9ms real) (~ rews/sec)
result Marking: $ $ q q q
```

```
Maude> rew [2] $ $ q q .

rewrite [2] in VENDING-MACHINE : $ $ q q .
rewrites: 2 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ q q q
Maude> rew [3] $ $ q q .

rewrite [3] in VENDING-MACHINE : $ $ q q .
rewrites: 3 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ q q q q
Maude> rew [4] $ $ q q .

rewrite [4] in VENDING-MACHINE : $ $ q q .
rewrites: 4 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ $ q q q q
Maude> rew [5] $ $ q q .

rewrite [5] in VENDING-MACHINE : $ $ q q .
rewrites: 5 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ $ $

Maude> rew [6] $ $ q q .
rewrite [6] in VENDING-MACHINE : $ $ q q .
rewrites: 6 in 0ms cpu (0ms real) (~ rews/sec)
result Marking: $ $ $ $ $ q

Maude> rew [200] $ $ q q .
rewrite [200] in VENDING-MACHINE : $ $ q q .
rewrites: 200 in 10ms cpu (10ms real) (20000 rews/sec)
result Marking: $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
$ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
$ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $
$ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ q q q.
```

3.6 Commandes Maude

La liste qui suit présente quelques commandes Maude. En effet Maude à plusieurs autres commandes qui sont décrit en [82].

1. **red** Term.

   La commande demande de réduire le terme "Term" en utilisant les équations dans le module courant.

2. **red** in mod : Term .

   la commande réduire le terme "Term" dans le module "mod"

3. **rew** Term .

   Cette commande permet de réécrire le term "Term" en utilisant les équations les axiomes d'appartenance et les règle de réécriture jusqu'à ce qu'aucune règle ne peut être appliquée.

4. **rew** [n] Term .

Réécrire le terme Term pour un nombre n de réécriture. (Le nombre des réductions équationnelles n'influent pas sur le nombre n).

5. **frew** Term .

   Pour "fair rewriting", c'est la même que la commande rew à l'exception que les règles sont applicable d'une façon uniforme. Cette commande est souhaitable pour les spécifications orienté objets parc qu'elle assure que chaque objet avoir à chance à se réécrire.

6. **search** {[bound]} {in module :} subject searchtype pattern {such that condition}.

   La commande search performe une recherche dans le calcul de la commande "rewrite". Cette recherche débute du terme 'subject' jusqu'à un état final qui correspond à "pattern" et qui satisfait une condition optionnelle.

   Les valeurs possible de "search tapes' sont :

   - =>1, un pas de calcul
   - =>+, un ou plusieurs pas de calcul
   - =>*, zero ou plusieurs pas de calcul
   - =>!, seulement les états finaux sous forme canonique sont permis.

   Le nombre maximal des chemins (solutions) trouvées est borné mais peut être s'étendre à l'infini. Pour afficher le graphe généré par la commande search il faut utiliser la commande **Show search path.**

7. **select** mod.

   La commande permet de sélectionner le module mod pour qu'il soit le module courant.

8. **set trace on**.

   Demande à Maude d'afficher comment il applique les équations.

9. **show module.**

   Cette commande, permet à Maude d'afficher le module courant.

10. **show sorts**

    Permet d'afficher les sorts du module courant.

Plusieurs d'autres commandes se trouvent dans le manuel du langage Maude [82].

Après la présentation du langage Maude. Nous s'intéressons dans la section suivante à la technique de model-checking avec Maude et au Maude LTL Model-checker.

4. Model-Checking et Maude LTL model-checker

4.1 Model checking

Le Model-checking [92] est une technique de vérification automatique des systèmes dynamiques. Il s'agit de construire un modèle du système à vérifier avec un formalisme donnée, généralement représenté par un automate à états fini. Ce modèle est vérifié s'il satisfait un ensemble de propriétés (une spécification) souvent formulées en logique temporelle [93]. En autres termes et de point de vue logique, le système est décrit par un modèle sémantique (structure de Kripke), et les propriétés sont décrites par des formules logiques [97].

Les modèles checker qui sont des outils de vérification automatique des programmes sont appliqués avec succès dans la vérification des semi-conducteurs, la conception des processus et dans la télécommunication. Aujourd'hui plusieurs firmes de fabrication de processeurs ont leurs propre model-checker [72] dû aux résultats obtenus par ces outils.

4.2 Structure de kripke et LTL

Les structures de kripke sont des modèles naturels pour la logique temporelle propositionnelle. Une structure de kripke est un système total de transition non étiqueté auquelle il est ajouté des prédicats dans l'ensemble de ses états.

Une relation binaire $R \subseteq A \times A$ sur l'ensemble A est appelée totale si et seulement si pour chaque $a \in A$, il existe au moins un $a' \in A$ tel que $(a, a') \in R$. si R n'est pas total, il peut être total par la définition $R^* = R \cup \{(a, a) \in A^2 \mid \nexists a' \in A(a, a') \in R\}$.

Une structure de Kripke est un triplet $\mathcal{A} = (A, \rightarrow_A, R)$ tel que A est un ensemble d'états, $\rightarrow_A$ est une relation binaire totale sur l'ensemble A et appelé la relation de transition, et $L : A \rightarrow \rho(AP)$ est une fonction appelée fonction d'étiqueté associé à chaque état $\in A$. l'ensemble $L(a)$ de ces propositions atomiques de AP et qui sont inclus à chaque état.

La fonction d'étiquette $L : A \rightarrow \rho(AP)$ spécifie quelles sont les propositions associées à chaque état.

Supposons un ensemble AP de propositions atomiques, on définie la logique temporelle linaire propositionnelle $LTL(AP)$ comme suit :

- **True** (vrai). $T \in LTL(AP)$.
- **Atomic Propositions** (propositions atomiques). Si $p \in AP$, donc $p \in LTL(AP)$.
- **Next Operator** (opérateur suivant) $si\ \varphi \in LTL(AP), donc$ Oφ ∈ LTL(AP).
- **Until operateur** (opérateur jusqu'à) $si\ \varphi, \psi \in LTL(AP), donc\ \varphi \mathrm{U} \psi \in LTL(AP)$.

pour une structure de Kripke $\mathcal{A} = (A, \rightarrow_A, R)$ l'ensemble $path(A)$ de ces chemins est un ensemble de fonctions ayant la forme $\pi :\ \mathrm{N} \rightarrow A$. Ainsi, pour chaque $n \in\ \mathrm{N}$, on a :

$\pi(n) \rightarrow_A \pi(n + 1)$.

Les modèles de la logique $LTL(AP)$ sont les différentes structures de Kripke $\mathcal{A} = (A, \to_A, R)$ qui ont AP comme un ensemble de propositions, en autres termes $L : A \to \rho(AP)$.

La relation binaire de satisfaction $\mathcal{A} \vDash LTL$ φ est vérifiée, par définition, si et seulement si pour tout $a \in A$, la relation de satisfaction $\mathcal{A}, a \vDash LTL$ φ est vérifiée si et seulement si pour chaque $a \in A$ et pour tous les chemins $\pi \in Path(A)$ tel que $\pi(0) = a$, la relation quaternaire de satisfaction $\mathcal{A}, a, \pi \vDash LTL$ φ est vérifiée [85].

- On a toujours, $\mathcal{A}, a, \pi \vDash LTL$ ⊺.
- Pour $p \in AP$,

$$\mathcal{A}, a, \pi \vDash LTL\ \mathrm{p} \quad \Leftrightarrow \quad \mathrm{p} \in \mathrm{L(a)}.$$

- For 0φ ∈ LTL(A),

$$\mathcal{A}, a, \pi \vDash LTL\ 0\varphi \quad \Leftrightarrow \quad \mathcal{A}, \pi(1), s; \pi \ \vDash \ LTL\ \varphi$$

Where s : N → N est la fonction successeur.

- For φ U ψ $\in LTL(\mathcal{A})$,

$$\mathcal{A}, a, \pi \vDash LTL\ \varphi\ \mathrm{U}\ \psi \quad \Leftrightarrow$$

$$(\exists n \ \in \mathrm{N})\ ((\mathcal{A}, \pi(n), s^n, \pi \ \vDash LTL\ \psi) \wedge ((\forall m \in N) m < n \ \Rightarrow \ \mathcal{A}, \pi(m), s^m;\ \pi \ \vDash LTL\ \varphi$$

- For ¬φ ∈ LTL(AP),

$$\mathcal{A}, a, \pi \vDash LTL \neg\varphi \quad \Leftrightarrow \quad \mathcal{A}, a, \pi \not\Vdash LTL \neg\varphi\ .$$

- For φ ∨ ψ $\in LTL(AP)$,

$$\mathcal{A}, a, \pi \vDash LTL\ \varphi \vee \psi \quad \Leftrightarrow \quad \mathcal{A}, a, \pi \vDash LTL\ \varphi \quad \text{or} \quad \mathcal{A}, a, \pi \vDash LTL\ \varphi.$$

D'autres connecteurs LTL peuvent être définis avec l'ensemble des connecteurs définis au dessus.

Connecteurs booléennes :

$$\bot = \neg \top$$

$$\varphi \wedge \psi = \neg\big((\neg\varphi) \vee (\neg\psi)\big)$$

$$\varphi \longrightarrow \psi = (\neg\varphi) \vee \psi$$

D'autres opérateurs temporels

- **Eventually** :◊ φ = ⊺ U φ
- **Henceforth :** □φ = ¬ ◊ ¬ φ
- **Release :** φ R ψ = ¬((¬φ)U (¬ψ))
- **Unless:** φ W ψ = (φ U ψ) ∨ (□φ)

- **Leads-to:** $\varphi \rightsquigarrow \psi = \Box(\varphi \rightarrow \Diamond \psi))$

Les propriétés à vérifier sont exprimés en cette logique temporelle. Dans la section suivante, nous présentons le LTL Model-checker du Maude.

4.3 Maude LTL Model-checker

La plate-forme Maude est équipée par un Model-checker appelé Maude LTL Model-checker [85] [86]. Il est utilisé pour vérifier si tous les comportements à partir d'un état initial satisfirent une propriété donnée exprimée en logique temporelle linaire. Le model-checker du Maude est comparable en termes de puissance avec les meilleurs Model-checkers tels que SPIN et SMV [81].

Selon [85], pour appliquer le model-checking deux niveaux de spécification sont définis;

- niveau de spécification système,
- niveau de spécification de propriétés.

1. **Niveau de spécification système :** dans ce niveau, nous s'intéressons à la formalisation du système à vérifier. Ainsi, nous représentons le système par une théorie de réécriture. Généralement, nous utilisons pour ce fait un module système qui décrit le système à analysé. Nous avons introduit dans les sections précédentes les différents modules du langage Maude. La spécification de ce niveau est un module système contenant des sortes (types de données du système), des opérations pour manipuler ces sortes, des équations, des axiomes d'appartenance et des règles de réécriture qui définissent le comportement du système.

2. **Niveau de spécification des propriétés** : dans ce niveau, on définit les propriétés à vérifier dans le système. Le système est décrit par un module système, comme nous l'avons mentionné dans le niveau de spécification système.
En évaluant l'ensemble des états accessibles à partir d'un état initial, la technique de model-checking permet de vérifier une propriété donnée dans un état ou dans un ensemble d'états [85]. Les propriétés sont exprimées en logique temporelle linéaire (LTL). Maude intègre un LTL model-checker, qui est spécifié aussi en Maude, suite à la simplicité de cette logique et aux algorithmes de décision qui sont bien définis [82].
Le model-checker contient un module appelé **LTL** qui définit toutes les opérateurs qui permet de construire les formules en logique temporelle linéaire. Le module LTL est spécifié comme suit :

```
fmod LTL is
protecting BOOL .
sorts Formula .
*** primitive LTL operators
ops True False : -> Formula [ctor format (g o)] .
op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
op _/\_ : Formula Formula -> Formula
[comm ctor gather (E e) prec 55 format (d r o d)] .
op _\/_ : Formula Formula -> Formula
```

```
[comm ctor gather (E e) prec 59 format (d r o d)] .
op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
op _U_ : Formula Formula -> Formula
[ctor prec 63 format (d r o d)] .
op _R_ : Formula Formula -> Formula
[ctor prec 63 format (d r o d)] .
*** defined LTL operators
op _->_ : Formula Formula -> Formula
[gather (e E) prec 65 format (d r o d)] .
op _<->_ : Formula Formula -> Formula [prec 65 format (d r o d)].
op <>_ : Formula -> Formula [prec 53 format (r o d)] .
op []_ : Formula -> Formula [prec 53 format (r d o d)] .
op _W_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
op _|->_ : Formula Formula -> Formula [prec 63 format (d r o d)].
op _=>_ : Formula Formula -> Formula
[gather (e E) prec 65 format (d r o d)] .
op _<=>_ : Formula Formula -> Formula [prec 65 format (d r o d).
vars f g : Formula .
eq f -> g = ~ f \/ g .
eq f <-> g = (f -> g) /\ (g -> f) .
eq <> f = True U f .
eq [] f = False R f .
eq f W g = (f U g) \/ [] f .
eq f |-> g = [](f -> (<> g)) .
eq f => g = [] (f -> g) .
eq f <=> g = [] (f <-> g) .
*** negative normal form
eq ~ True = False .
eq ~ False = True .
eq ~ ~ f = f .
eq ~ (f \/ g) = ~ f /\ ~ g .
eq ~ (f /\ g) = ~ f \/ ~ g .
eq ~ O f = O ~ f .
eq ~ (f U g) = (~ f) R (~ g) .
eq ~ (f R g) = (~ f) U (~ g) .
endfm
```

Maude offre aussi un module fonctionnel 'SATISFACTION' qui défini l'opérateur (|=) qui indique si une formule donnée est satisfaite dans le système à partir d'un état donnée. La définition du module SATISFACTION est spécifié comme suit :

```
fmod SATISFACTION is
protecting BOOL .
sorts State Prop .
op  _|=_  :  State  Prop  ->  Bool  [frozen] .
endfm
```

Maude défini la sorte State qui représente un état générique pour n'importe quel système. Le comportement du système à vérifié est spécifié dans un module système supposant qu'il est noté M. l'utilisateur peut définir pour ce système plusieurs prédicats qui sont liés à son comportement. Ces prédicats sont décrits dans un nouveau module qui importe les deux modules; le module système M (qui spécifie le comportement), et le module

SATISFACTION. Supposons que ce nouveau module (où les prédicats sont définis) est appelé M-PREDS. La spécification de ce dernier suit la syntaxe suivante :

```
mod M-PREDS is
protecting M .
including SATISFACTION .
subsort Configuration < State .
...
endm
```

Maude LTL Model-checker propose le module MODEL-CHECKER qui offre la fonction Model-check. L'utilisateur peut invoquer cette fonction en la donnant un état initial et la formule qu'on souhaite vérifiée. Le model-checker va vérifier si cette formule est satisfaite dans l'espace d'états de système ou non. La fonction Model-check retourne True si la propriété est satisfaite et un contre exemple (counterexemple) dans le cas échéant (propriété non vérifiée). La syntaxe du module MODEL-CHECKER est la suivante : [32]

```
fmod MODEL-CHECKER is
protecting QID .
including SATISFACTION .
including LTL .
subsort Prop < Formula .
*** transitions and results
sorts RuleName Transition TransitionList ModelCheckResult .
subsort Qid < RuleName .
subsort Transition < TransitionList .
subsort Bool < ModelCheckResult .
ops unlabeled deadlock : -> RuleName .
op {_,_} : State RuleName -> Transition [ctor] .
op nil : -> TransitionList [ctor] .
op __ : TransitionList TransitionList -> TransitionList
[ctor assoc id: nil] .
op counterexample :
TransitionList TransitionList -> ModelCheckResult [ctor] .
op modelCheck : State Formula ~> ModelCheckResult
[special (...)] .
endfm
```

Dans la section suivante nous introduisons l'ensemble d'étapes à suivre pour utilise le LTL model-checker du Maude.

3. **Model-checking avec Maude LTL model-checker :** selon [85], les étapes à suivre pour vérifier des propriétés dans un système donné sont :
   a. Spécifier le comportement du système dans un module Système M.
   b. Définir un nouveau modèle, appelé par exemple CHECK-M qui inclue le modèle M (qui décrit le comportement du système) et le module prédéfini MODEL-CHECKER comme des sous modules (éventuellement nous ce module peut inclure d'autres modules).
   c. Faire une déclaration de sous-sorte (Subsort) $state_M$ **< State**, où State est la sorte clé dans le module MODEL-CHECKER.

d. Définir la syntaxe des états prédicats que l'utilisateur désire utiliser comme des constants et des opérations de sorte **Prop**. Prop est une sous-sorte de la sorte **Formula** qui est définie dans le module MODEL-CHECKER.

e. Définir la sémantique des états prédicats par la baie de l'équation :

Op _ |= _: State Prop -> Result [special …] .

Dans le module MODEL-CHECKER, la sorte **Result** est une sous-sorte de **Bool.** On définie la sémantique de chaque état prédicat (appelé état prédicat paramétré **p**), par la déclaration d'un ensemble d'équations (voir conditionnelles) sous forme :

ceq $\exp_{11}$ **|= p (**U_{11}**, …,**U_{1n}**) = True if** C_1**.**

….

ceq expk |= p (U_{1k}**, …,**u_{nk}**) = True if** C_k **.**

Où : $\exp_i, 1 \leq i \leq k$ sont des paternes de sorte $state_M$ qui peuvent être des termes avec des variables.

p ($U_{i1}, \ldots, U_{in}$), $1 \leq i \leq k$ sont des patterns de sorte **Prop**.

Chaque condition C_i, $1 \leq i \leq k$, est une conjonction d'égalité et d'appartenance et peut inclure des fonctions auxiliaires qui sont importé d'autre module ou défini dans le module M-CHECK

f. Une fois les états prédicat sont définis, l'utilisateur peut donner l'état initial **Init** du système à partir duquel le model-checker commence la vérification. Pour vérifier une formule LTL, disant **form,** qui contienne des prédicats, on vérifier cette formule par la commande **reduce init |= form .**

Si l'ensemble des états atteignables est fini, deux types de résultats peuvent avoir lieu; la formule est satisfaite et Maude return **True**; Ou, dans le cas échéant (la formule n'est pas vérifiée) Maude affiche un contre exemple qui exprimé comme suit :

op counterExample : TransitionList TransitionList -> Result [ctor] .

Nous n'allons pas couvrir toutes les détailles sur Maude LTL model-checker dans ce chapitre. Veuillez consulter [82],[81],[85] pour plus de détailles sur cet outil de model-checking .

Dans ce chapitre, nous avons vu quelques concepts sur les méthodes formelles et la logique de réécriture qui constituent un cadre sémantique pour spécifier des Langages et des systèmes concurrent. Nous avons vu aussi, le langage formel Maude qui est basé sur la logique de réécriture. Nous avons terminé par la présentation de la technique de model-checking tout en focalisant sur le LTL model-checker du langage Maude. Toutes ces notions servent de base pour l'approche que nous proposons et qui vise à vérifier des modèles UML en utilisant le langage Maude et son LTL model-checker. Cette approche est le vif du chapitre suivant.

CHAPITRE IV :

Méthode de Model-Checking des Modèles UML en utilisant Maude

Nous avons vu dans les précédents chapitres qu'UML est un standard largement adopté dans la modélisation des systèmes informatiques. Toutefois, les modèles UML sont toujours besoin d'être vérifiés pour assurer que le comportement modélisé est correcte et qu'il correspond aux besoins fonctionnels du système. Nous avons cité plusieurs travaux qui visent à renforcer les modèles UML par une sémantique formelle permettant aux utilisateurs du langage de vérifier les modèles qu'ils élaborent.

Dans le même cadre, Nous proposons une méthode de vérification des modèles UML en utilisant le langage Maude et son LTL model-checker. Cette méthode vise à translater les diagrammes de classes, de collaborations et d'états-transitions à une spécification Maude. Les propriétés à vérifier sont définis par le concepteur du modèle et elles sont exprimées en logique temporelle linéaire. La méthode présente aussi une technique de vérification de l'exécution des collaborations en UML vis-à-vis les diagrammes d'états-transition.

Le chapitre est organisé en deux grandes parties, la première partie est consacrée à la présentation de la méthode. La deuxième partie contient des exemples d'illustration de l'application de cette méthode sur des modèles UML.

1. Présentation de la Méthode

Pour vérifier les modèles UML en utilisant le langage formel Maude et son LTL model-checker, on propose une méthode qui se repose sur quatre étapes (figure (Fig.43)), à savoir ;

- **Etape 1** - **Elaboration du Modèle UML**; il s'agit dans cette première étape d'élaborer le modèle UML en sélectionnant uniquement pour la translation le digramme de classe qui modélise la structure statique du système, les diagrammes d'états-transitions qui spécifient le comportement interne de chaque objet dans le système, et le diagramme de collaboration qui représente l'interactions entre objets.
- **Etape 2 - Translation du Modèle UML à une spécification Maude**: dans cette étape, les diagrammes élaborés dans la première étape se traduisent à une spécification Maude, où chaque classe et son diagramme d'états-transitions associé sont spécifiés en un module orienté objet du Maude. ces modules orientés objets sont intégrés dans un seul module représentant la globalité du système. sur ce module que le model-checking est appliqué.

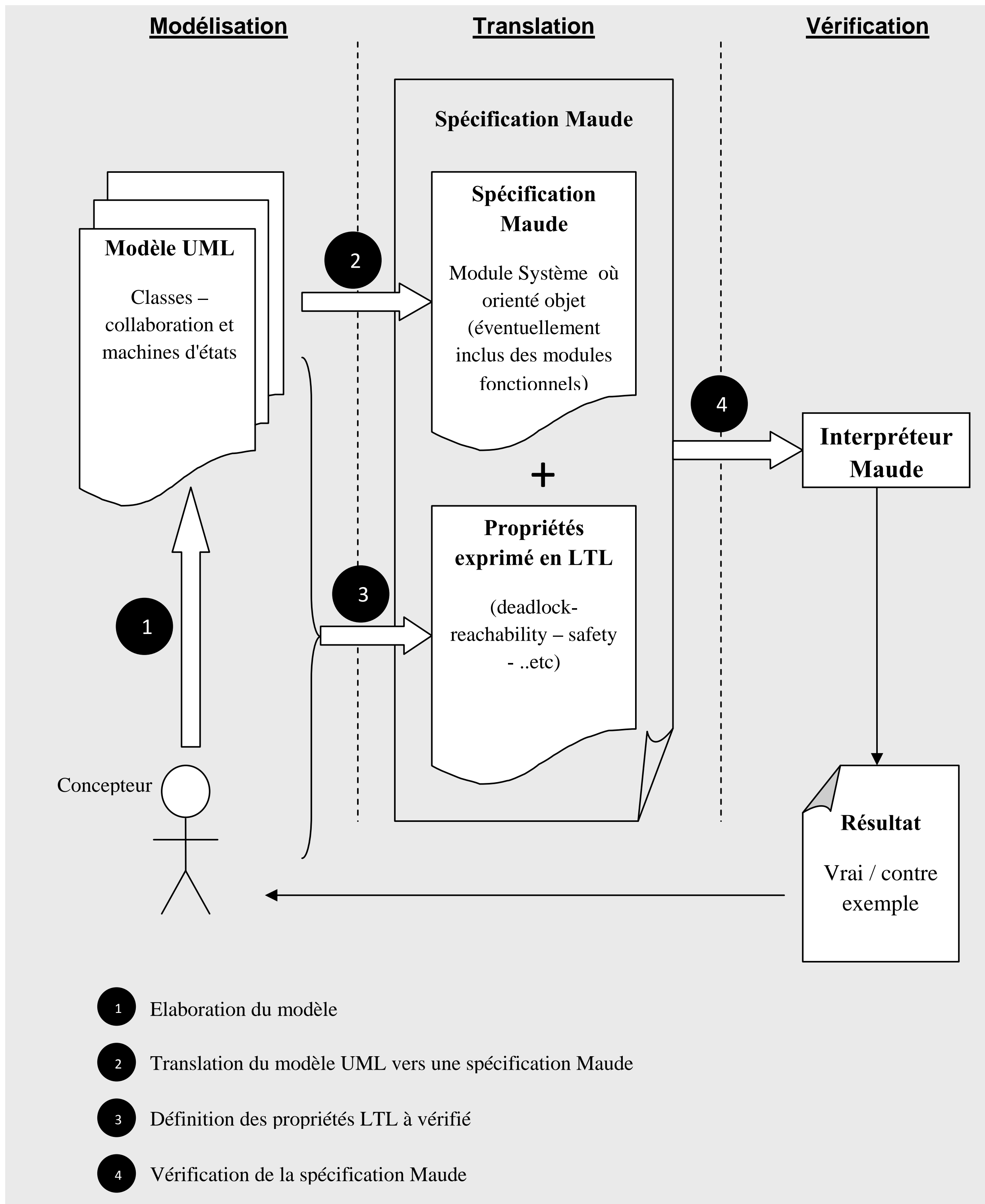


Fig.43 : Les différentes étapes de la méthode de vérification du Modèle UML

- **Etape 3 - Définition des propriétés**: l'utilisateur doit définir les propriétés qu'il veule vérifiées. Ces propriétés doivent être exprimées en logique temporelle. Elles sont vérifiées vis-à-vis la spécification Maude qui représente le modèle UML**.**
- **Etape 4** - **Vérification de la spécification / vérification du modèle)** : c'est l'étape d'analyse et de vérification de la spécification élaboré dans les étapes précédentes. Le concepteur utilise l'interpréteur et les commandes du langage Maude. Le résultat de vérification (model-checking) pour chaque propriété est soit Vrai si les propriétés sont satisfaites ; soit un contre exemple est affiché. L'utilisateur peut raisonner sur ce contre exemple pour détecter les erreurs dans le modèle UML. Il peut ainsi porter les modifications nécessaires sur le modèle afin de le corriger.

Dans les sections suivantes, nous expliquons ces étapes avec plus de détaille en donnant les informations nécessaire pour conduire la vérification des modèles UML avec le langage Maude et son LTL model-checker.

1.1 Elaboration du Modèle UML

La méthode que proposons tient compte à trois diagrammes UML; le diagramme de classe, le diagramme de collaboration et les diagrammes d'états-transitions. Ainsi, nous proposons d'utiliser ces trois diagrammes pour vérifier des spécifications des systèmes. Pratiquement, un système n'est qu'un ensemble d'objets actifs qui interagissent entre eux par l'envoi de messages en exhibant un comportement spécifique.

Le diagramme de classe permet de modéliser la structure statique du système. Il représente l'ensemble des classes d'objets et les associations entre ces classes. Chaque classe contient une partie statique (attributs) et une partie dynamique (méthodes).

Le diagramme d'états-transition (machine d'état), nous permet de spécifier le comportement dynamique de chaque objet dans le système. Il spécifie l'ensemble d'états qu'un objet peut avoir durant sa vie. Il spécifie également les différentes réponses de l'objet vis-à-vis les stimuli provenant de l'extérieur.

Le diagramme de collaboration représente une collection d'objets en interactions. Il représente également l'ordre d'apparition des messages échangés entre les objets dont le but est de réaliser une tache précise.

Par conséquent, notre méthode est applicable sur un modèle UML contenant :

- un diagramme de classe
- des diagrammes d'états-tranistions
- un diagramme de collaboration ou plus.

Eventuellement, on peut vérifier un diagramme d'état transition avec un ensemble d'événements pour vérifier que le diagramme est bien spécifie.

La figure (Fig.44) représente les diagrammes utilisés par la méthode de vérification d'UML avec Maude. C'est à l'utilisateur d'élaborer son modèle en basant sur ces diagrammes

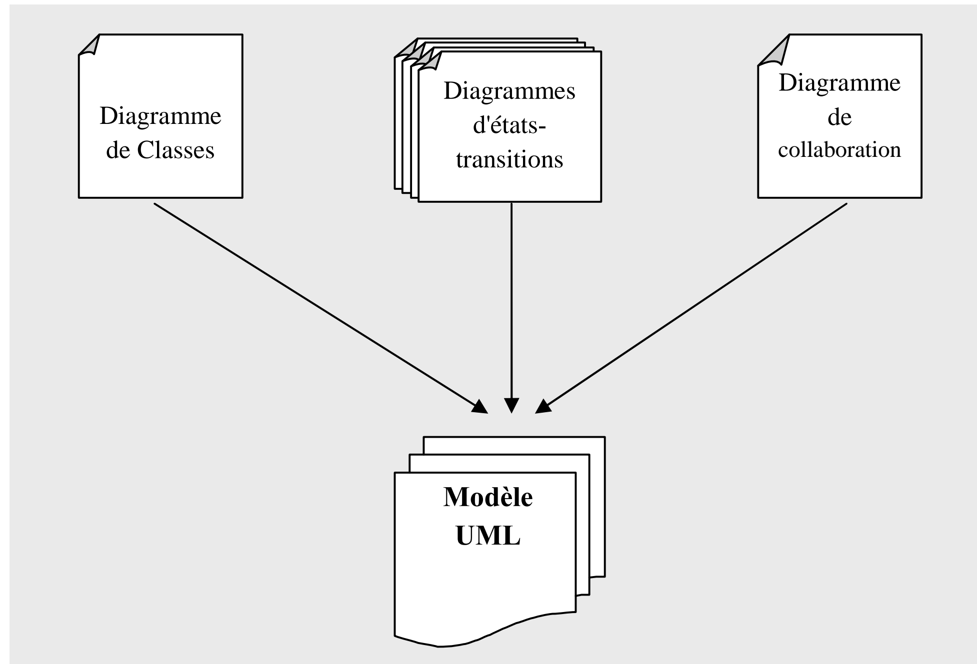


Fig.44 : Diagrammes UML utilisés par la Méthode de vérification

1.2 Translation du Modèle UML

Nous avons vu qu'il est associé à chaque classe un diagramme d'états-transitions qui décrit le comportement des ses instances. En effet, les instances d'une classe (objets) interagissent avec l'environnement par l'envoi des messages et l'exécution des méthodes internes, en réaction aux événements issus de l'extérieur.

Pour analyser et vérifier un ensemble d'objets en interaction représentés par des diagrammes d'états-transitions et des collaborations, Nous proposons de translater ces diagrammes (Modèles UML) à une spécification de logique de réécriture représentée par le langage déclaratif Maude.

Cette translation correspond exactement au niveau de spécification système proposé par le Model-checker du Maude. Chaque classe dans le modèle UML et son diagramme d'états-transitions associé sont traduit en un module orienté objet déclaré en Maude.

L'idée clé de la translation du diagramme d'états vers des spécifications Maude, est que les transitions entre états peuvent être spécifiées naturellement par des règles de réécriture de forme $t \rightarrow t'$ où t, t' sont des termes qui représente des états ou ils font partie de l'état global du système.

Maude offre aussi les concepts nécessaires pour supporter la spécification orientée objet. Il permet de déclarer des classes, des messages, des configurations …etc. en outre, il permet de spécifier les communications synchrones et asynchrones …etc.

Nous proposons de traduire les éléments du modèle UML comme suit :

1. **Classe et Attributs :** chaque classe dans le modèle UML est déclarée comme suit :

$$\textit{class}\ \ \text{Nom_Classe}\ \mid\ Att_1 : S_1, ..., Att_n : S_n .$$

Où :

class est le mot clé pour déclarer une classe en Maude.

Nom_Classe indique le nom de la classe déclarée.

$Att_1 : S_1, ..., Att_n : S_n .$ pour définie la partie statique d'une classe dont $Att_i : S_i$, $1 \leq i \leq n.$ sont les attributs att_i de sorte S_i .

Maude propose à ses utilisateurs des types prédéfinis (des entiers, des booléens, des Strings …etc). L'utilisateur peut définir ces propres types (sort) et les opérations applicables sur ces types. Nous avons vu dans le chapitre précédent comment on déclare des sortes et des opérations.

A toute fin utile, nous proposons que chaque classe doive contenir les attributs suivants :

- Attribut **STATUS** : c'est un attribut est de sorte **STATE**. Il indique l'état courant de l'objet instance. En autre termes, il prend les différents états présenté dans un diagramme d'états-transitions.
- Attribut **Blocked** : c'est un attribut est de sorte **Bool**. Il indique si l'objet instance est bloqué ou non. c'est un attribut très utile lors de spécification des communications synchrones dont l'objet appelant est bloqué jusqu'à ce que son message soit consommé pas l'objet appelé.
- Attribut **pour chaque Liens** : en effet, les liens sont des instances des associations dans le diagramme de classes. Il indique à quels autres objets l'objet courant est relié. Ces attributs doivent être déclarés de sorte Oid (Object Identifier) qui est défini par défaut dans full Maude.
- Eventuellement, nous pouvons ajouter un attribut "**stéréotype**" qui indique c'est l'objet est dans un état **normal**, un état de **succès** (il a accompli une tâche précise), un état d'erreur (l'objet à exhiber un comportement non désiré). L'attribut **Stype** de type **String** peut inclus les opérations suivante :

ops **Error Noraml Success** : -> String .

La figure (Fig.45) présente un exemple de translation d'une classe UML en Maude.

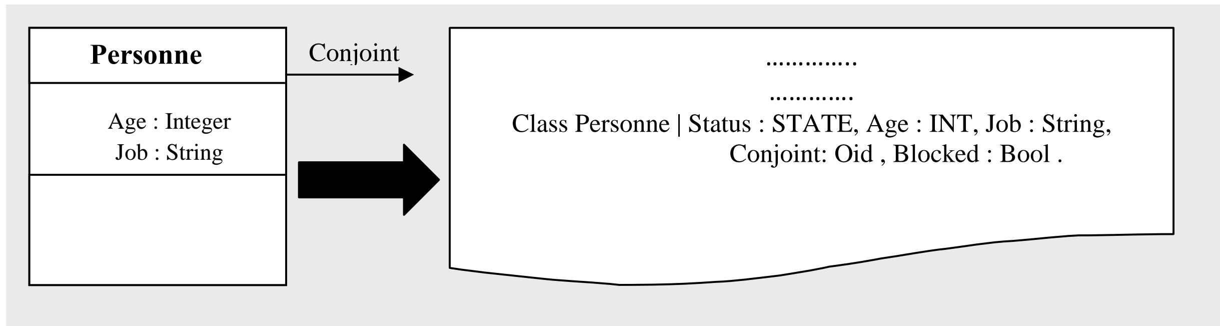


Fig.45 : Exemple de translation d'une classe en Maude

2. **Etats :** pour présenter les états (simples, composites), qui se trouvent dans les diagrammes d'état transition en Maude, nous avons déclaré la structure algébrique suivante :

sorts **SIMSTATE COMSTATE STATE** .
subsort SIMSTATE < COMSTATE .
subsort COMSTATE < STATE .
op **none** : -> COMSTATE [ctor] .
op _||_ : COMSTATE COMSTATE -> COMSTATE [ctor assoc comm id: none].

Pour chaque état dans le diagramme d'états-transitions, on déclare une opération avec le nom d'origine de l'état selon la syntaxe suivante :

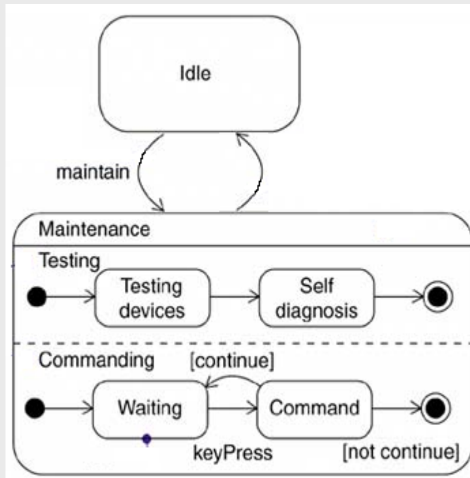
op Nom_etat : -> SIMSTATE.
Pour déclarer un état simple nommé : nom_etat.

op Nom_etatseq : SIMSTATE -> STATE .
Pour déclarer un état séquentiel nommé : Nom_etatseq.

op Nom_etatcon : COMSTATE -> STATE .
Pour déclarer un état concurrent nommé : Nom_etatcon.

La figure (Fig.46) représente une translation des états d'un diagramme d'états à une spécification Maude. Par exemple, l'utilisation des ces déclarations peut être comme suit :

- **Maintenance (Testing(Testing_device) || Commanding (Command))** signifié que l'état Maintenance est un état composite concurrent qui contient deux régions **Testing** et **Commanding**. Les deux flots sont exécutés en parallèle où le premier (**Testing**) est dans le sous-état **Testing_device** et le deuxième (**Commanding**) est dans le sous-état **Command**.



3. **Evénements :** les événements (signaux, appelles aux méthodes, événements déferrés, …etc.) sont traduits à des messages durant leurs translation à la spécification Maude. les messages en full Maude sont déclarés comme suit:

msg nom_message **: -> Msg.**

La forme générale qu'on adopte dans cette méthode est la suivante :

msg nom_event **:** Oid Oid $p_1,...,p_n$ **-> Msg.**

Où:

msg est un mot réservé pour déclarer un message.

nom_event est le nom de l'événement.

Oid Oid $p_1,...,p_n$, sont les paramètre du message dont le premier Oid est l'identificateur de l'objet émetteur. Le deuxième Oid identifié l'objet récepteur du message. $p_1,...,p_n$ Sont les sortes des paramètres du message s'il représente un appelle à une méthode avec paramètre.

Supposons qu'un objet O1 veut appeler la méthode M1 (a: integer, B : String) de l'objet O2. La syntaxe pour déclarer le message qui représente cet événement est la suivante :

msg M1: Oid Oid Int String -> Msg.

Un exemple d'instance de l'événement décrit en dessus est le suivant : M1 ***(O1, O2, 5, "Pomme")***

Les événements d'appels aux méthodes sont de type asynchrone dont l'objet appelant est bloqué jusqu'à ce sont message (Nom_Event) est consommé par l'appelé. Pour débloquer l'appelant, il faut le notifier par un autre message Ack_Nom_Event. Ce messages est généré par l'appelé une fois il a consommé le message (Nom_Event).

L'ensemble des méthodes et des signaux qui se trouve dans la classe ou qui se trouvent au niveau d'autres classe doivent être déclarés comme des messages dans la spécification Maude. La figure (Fig.47) représente un exemple de translation des événements aux messages en Maude.

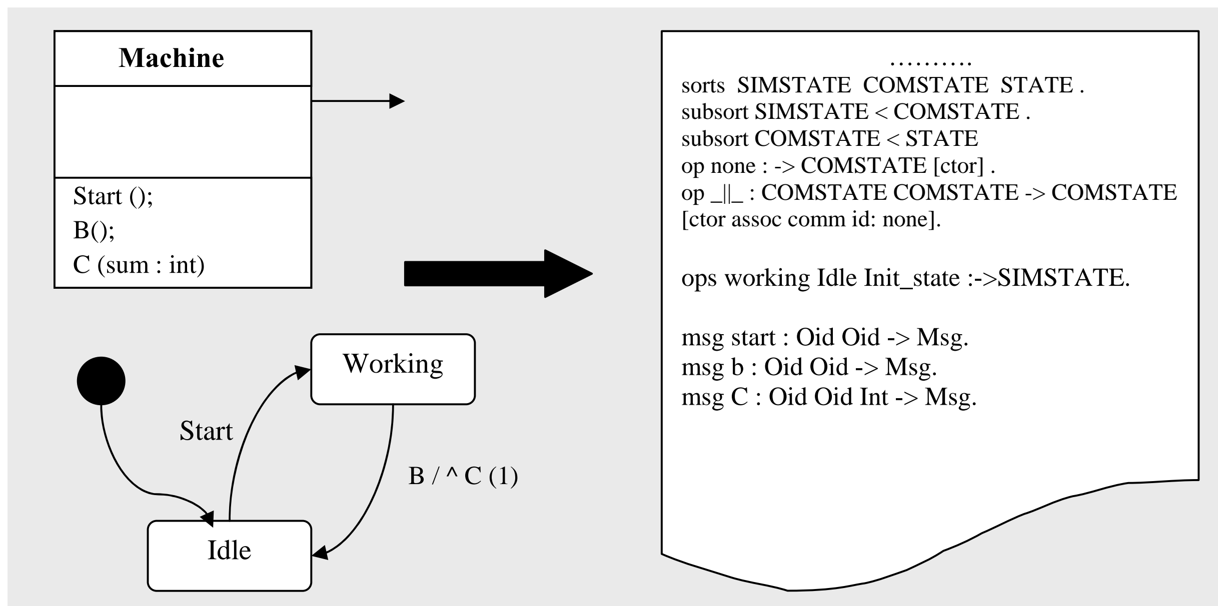


Fig.47 : Exemple de translation des événements

4. **Transitions :** Nous spécifions chaque transition dans le diagramme d'états par une règle de réécriture. Les règles de réécriture sont parfaitement adéquates pour décrire le changement d'états dans un objet.

   Une transition est représentée par une règle de réécriture conditionnelle dont la forme est :

   clr [Nom_Transition] : T => T' if C.

   Ou uniquement par une simple règle de réécriture ayant la forme **:**

   lr [Nom_Transition] : T => T'.

   Où : **[Nom_Transition]** est le nom de la transition.

T est une configuration qui contient l'événement trigger (message), l'objet dans un état S1 et le reste de configuration est un ensemble d'objets et des messages qui peuvent figurer dans le système.

T' est la nouvelle configuration dont l'objet qui a subit une transition interne est dans le nouveau état S2. L'événement trigger est disparu dans cette nouvelle configuration et les actions sont représentées par l'apparition de nouveaux messages et/ou des changements des valeurs des attributs.

C est la condition qui représente la garde de la transition. La dernière ne s'exécute que si la garde est évaluée à vrai.

Les transitions complexes sont représentées de la même façon qu'une règle simple. La différence majeure est que l'attribut **Status** de l'objet prend une valeur qui représente des états concurrents et non pas un état simple. La figure (Fig.48) représente des exemples de transitions et leur translation aux règles de réécriture en Maude.

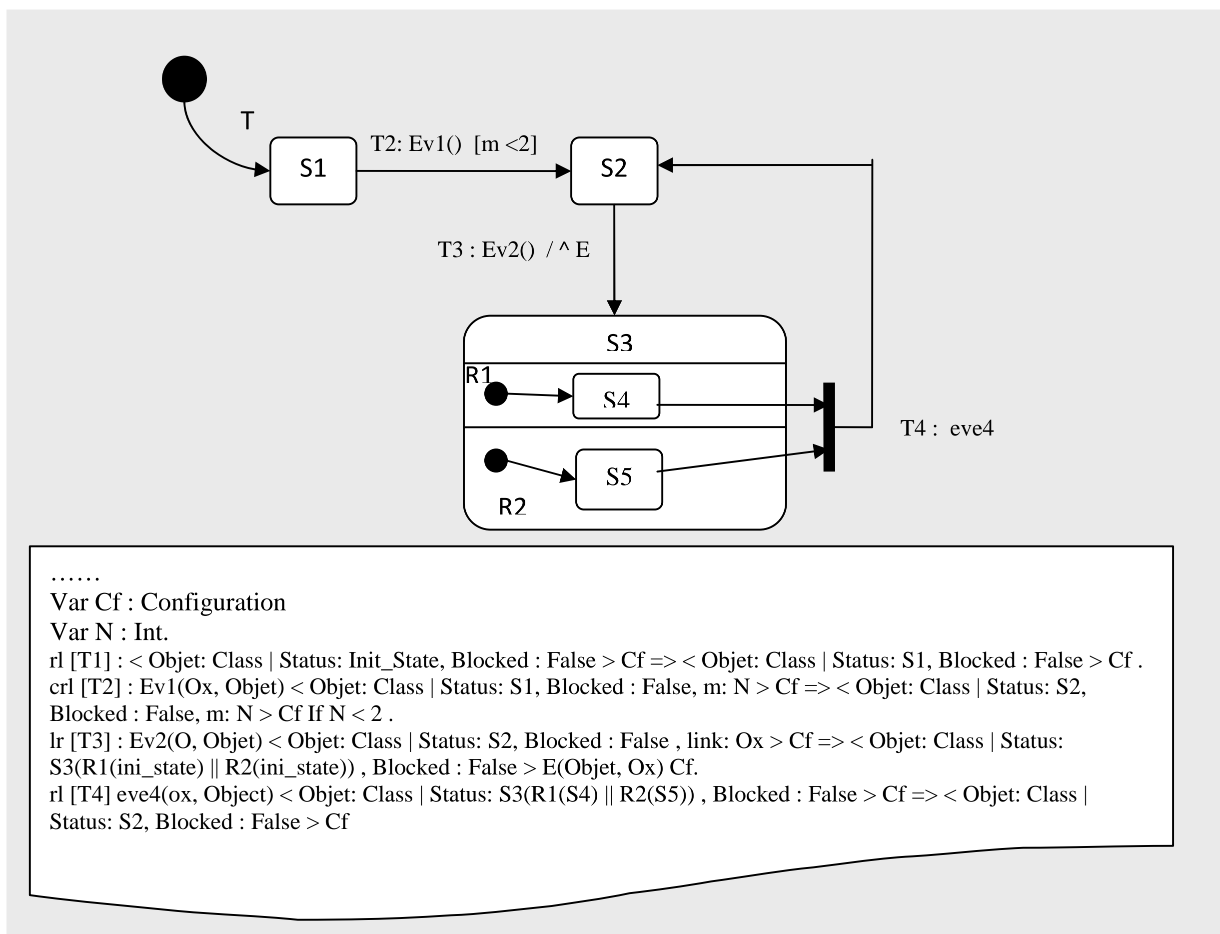


Fig.48 : Exemple de translation des Transitions

5. **Collaboration** : une fois la translation des classes et ses diagrammes d'états associés est terminée, il ne reste que déterminer la configuration initiale du système. Le diagramme de collaboration permet de donner des informations sur la configuration. Ainsi, nous pouvons extraire le nombre d'instances de chaque classe, les liaisons entre ces instances et l'état initial de chaque objet instance.

   La méthode qu'on propose, en outre que la vérification des propriétés temporelles, permet de vérifier l'exécution des collaborations vis-à-vis le comportement interne des objets spécifié par des diagrammes d'états-transitions. en autres termes, nous pouvons exécuter le système de tel façon qu'il permet d'assurer que l'enchainement des événements exécutées va parfaitement coller avec celui prévu dans le diagramme de collaboration.

   Pratiquement, nous proposons de spécifier les règles de réécriture de façon à exécuter seulement une séquence précise de transitions (événements). Pour ce fait, nous avons utilisé la structure algébrique suivante pour présenter la séquence des événements d'une collaboration :

```
sort seq col .
subsort Msg < seq .
op null : -> seq [ctor] .
op _-_ : seq seq -> seq [ctor assoc id: null ] .
op collaboration : seq -> col .
subsort col < Configuration.
```

   Où : **seq** est une liste qui représente la suite des événements. Cette structure n'est pas commutative afin de respecter l'ordre d'apparition des événements et son élément d'identité est null. **Collaboration (seq)** est l'opération qui représente la collaboration.

   Cette structure permet de spécifier les règles de réécriture ayant, pour ce type de vérification, la forme suivante :

```
Var
X : seq.
S1 S2 : state.
Cf : configuration.
Rl  [nom_regle]: collaboration (M1 - X) M1 < O : C | Status : S1 …> Cf =>
collaboration (X) < O : C | Status : S2 …> … Cf .
```

   Cette représentation permet d'exécuter uniquement les transitions (règles) qui vont à la fin exhiber parfaitement le comportement décrit par le diagramme de collaboration. On dit que la collaboration est vérifiée vis-à-vis le comportement internes des objets (diagrammes d'états transitions) si à la fin du calcul, la structure collaboration est vide (**collaboration(null)**). Ce qui signifier que l'ordre des événements et l'interaction entre objets correspond exactement à celle modélisée dans le diagramme de collaboration. Dans le cas ou la structure collaboration n'est pas vide, il est claire que la collaboration ne peut pas être terminée par rapport au comportement interne des objets. L'exemple II dans ce chapitre permet de donnée une vue plus claire sur ce type de vérification.

Apres la translation du modèle UML on obtiendra un ensemble de modules Maude. Chaque Module spécifie la structure statique et dynamique d'une classe d'objets. Ces modules sont inclus dans un autre Module qui va présenter le modèle en sa globalité. La configuration initiale dans ce module est définie selon le diagramme de collaboration qui contient des informations sur le nombre d'instance d'objets et les liens entre eux. Ce module sera utilisé dans l'étape de vérification pour vérifier des propriétés temporelles vis-à-vis le modèle.

1.3 Définition des propriétés à vérifier

Une fois, la translation du modèle UML vers des spécifications Maude est effectuée. L'utilisateur doit définir les propriétés qu'il souhaite vérifiées. C'est propriétés doivent être exprimées en logique temporelle linéaire. En effet, le LTL Model-checker du Maude ne supporte que les propriétés exprimées en cette logique. C'est à la charge d'utilisateur d'identifier ces propriétés et de l'exprimer en LTL.

Cette étape correspond exactement au niveau de spécification des propriétés proposé par Maude. Les propriétés vont être spécifiées dans un module Maude qui inclut en même temps le module MODEL-CHECKER et SATISFACTION définis par le model-checker du Maude.

Généralement, les propriétés LTL à vérifier par un modèle checker sont [72] :

1. **propriété d'invariance** : cette propriété permet d'assurer que le système n'exhibe jamais un comportement non désiré. On dit que la formule P est invariante dans un système si P est toujours vérifiée dans tous les états T atteignable à partir de t0 (T0 → t implique P(t)). Et on écrit en Maude ; **[] P** où P est une proposition (P :-> Prop).

   Parmi les exemples d'utilisation de cette propriété, on trouve :

- exclusion mutuelle **: []** $\neg(crit_1 \wedge crit_2)$.
- Absence de blocage : [] $(dg_1 \vee ... \vee dg_n)$.

2. **propriété de garanti (liveness)**: cette propriété permet de vérifier que le système va exhiber un comportement désiré. Parmi les exemples d'utilisation de cette propriété, on a : [] (q => <> P) où q et P sont des formules.

3. **propriété de "Reachability" :** cette propriété permet de vérifier si un comportement (désiré ou non désiré) va avoir lieu durant l'exécution du système. Généralement, cette propriété permet de rechercher sur les erreurs dans les spécifications.

   Parmi les exemples d'utilisation de cette propriété, on trouve :

- **<>** $\neg$ **P**
- **Q U P**

4. **Propriété de Réponse "reponsiveness" :** c'est l'une des propriétés importantes. Elle permit d'assurer que le système répond proprement aux stimuli.

Cette propriété est exprimé, en général, sous la forme : **[](request => O <> Ack).** Une application de cette règle peut être :

- accès garanti à la section critique : **[] (try1 => O <> crit1).**
- Séjour fini en section critique : **[] <> ¬crit1.**

5. **Propriété de réactivité :** cette propriété permet de vérifier qu'une partie du système (objet ou plusieurs) réagit toujours après l'apparition d'un comportement spécifié. Parmi les exemple de cette propriété on trouve l'exemple d'équité forte **[] <> (try1 ∧ ¬crit2) => []<> crit1** (Quand 1 veut entrer à la section critique et 2 n'est pas dans cette section, 1 automatiquement entre à la section critique).

6. **Propriété de Stabilité** : c'est la propriété qui permet de vérifier si la formule p est toujours satisfaite à partir de sa première apparition.
7. **Deadlcock :** c'est la propriété de blocage dans un état non désirable. On dit qu'un système concurrent est en Deadlock en cas ou tous les objets de ce système sont bloqués.
8. **Terminison :** c'est une propriété qui permet de vérifier la bonne terminaison d'une exécution.
9. **Propriété d'alternance** : permet d'exhibé deux comportements en alternance, par exemple : **[](A ↔ ¬OA)** permet de vérifier si A est vrai dans les états paires S0, S2, S4,… et fausse dans les états impaires.

La déclaration des prédicats est déjà couverte dans le chapitre précédent. Dans cette méthode nous allons suivre la même syntaxe décrit dans [85] pour déclarer les propriétés (chapitre3). Ces propriétés sont déclarées dans un module Maude avec la configuration initial du système. A la fin de cette étape, il ne reste que passer à l'étape d'analyse et de vérification.

1.4 Vérification du modèle

L'interpréteur Maude permet d'exécuter, d'analyser et de vérifier les deux niveaux de spécifications expliquées en dessus ; Le module global, qui représente une translation du modèle UML, et le module des prédicats qui spécifié les propriétés temporelle à vérifier.

Maude offre plusieurs commandes d'exécution et d'analyser des spécifications. On trouve entre autres :

- l'utilisation de la commande **Frewrite** et **Rewrite :** ces deux commandes permettent d'appliquer les règles de réécritures sur la configuration initiale. Cette configuration est un ensemble d'objets et de messages échangés entre eux. Chaque objet dans la configuration interagit avec les autres en exécutant, en concurrence, ses propres transitions internes qui sont représentées par des règles de réécritures. L'exécution se termine quand aucune règle de réécriture n'est applicable sur la configuration. Parfois, il préférable que le résultat ne soit pas l'état final du système, mais plutôt un des états intermédiaires du système. Pour ce fait, on

peut utiliser les commandes frew [**n**] et Rew [**n**]. ces deux commandes permettent d'afficher l'état du système après l'application de **n** règles de réécriture (n changements d'état).

- L'utilisation de la commande **Search :** Cette commande permet d'effectuer des recherches sur une configuration désirée à partir d'une configuration initiale. La commande **search**, comme nous avons vu dans le chapitre précédent, prenne comme paramètres l'état initial, l'état recherché, et le type de recherche. On peut vérifier également avec cette commande si un objet, par exemple, attient un état X. où X peut être un état désiré, un état d'échec ou un état d'erreur …etc.

  Lors de l'exécution de la commande, l'interpréteur Maude affiche le nombre des solutions trouvées. En autres termes, il affiche les chemins entre la configuration initiale à la configuration recherchée. La commande **show search path** permet d'afficher ces chemins et les règles de réécritures appliquées pour chaque chemin.

  Parfois la présence d'un seul chemin vers un état non désiré est suffisante pour dire que le modèle UML à besoin de modifications. L'utilisateur peut analyser le chemin pour localiser exactement l'erreur et les transitions ayant engendrées cette erreur de conception.

  Plusieurs propriétés peuvent être vérifiées sur le système en utilisant la commande **search**. Tels que les propriétés de reachability et d'invariance. Néanmoins, des propriétés plus complexes peuvent être vérifiées sur le système avec le LTL Modèle checker du langage Maude.

- Afin de vérifier les propriétés LTL élaborées dans l'étape 3, l'utilisateur doit appeler la fonction **modelcheck**. Cette fonction est définie dans le module MODEL-CHECKER. elle est invoquée sur l'interpréteur Maude selon la syntaxe :

  Red modelCheck(initState, LTLprop).

  Où : **red** est la commande **reduce** du langage Maude.

  initiState est l'état initial du système (configuration initiale).

  LTLProp est la propriété à vérifiée exprimée en logique temporelle linéaire.

  Le résultat d'exécution de cette commande est True 'vrai' dans le cas où la propriété temporelle est vérifiée. Dans le cas échéant, le résultat est un contre exemple qui représente un cas d'exécution où la propriété n'était pas satisfaite.

  L'analyse du contre exemple permet à l'utilisateur de détecter les causes du problème dans sa spécification. Par conséquent, il peut porter les corrections nécessaires sur son modèle UML. Evidement, les modifications portées sur le modèle UML peuvent être faite aussi sur la spécification Maude. Ce qui permet à l'utilisateur de refaire la vérification des mêmes propriétés temporelles sur la nouvelle spécification.

A la fin de la présentation de cette méthode, nous allons donner, dans les sections suivantes, des exemples de vérification des modèles UML en utilisant cette méthode qui est basée sur Maude et son LTL modèle Checker.

2. Exemple I : Problème de dîner des Philosophes

Notre premier exemple est le problème de dîner philosophes. Considérons qu'on à des philosophes installés autour d'une table ronde en passant leurs temps à penser et à manger.

Pour qu'un philosophe puisse mange, il doit prendre d'abord la fourche qu'il partage avec son voisin droite. Si la fourche est détienne par son voisin, il doit attendre jusqu'à ce qu'elle sera libre. Une fois il détient la fourche droite, il prend la fourche qu'il partage avec son voisin gauche. Si la fourche est détenue aussi par son voisin, il n'a qu'attendre jusqu'à ce qu'elle soit libre.

Une fois le philosophe détient les deux fourches, il commence à manger. Quand il termine il dépose les deux fourches. Le philosophe ne libère jamais une fourche tant qu'il n'a pas encore mangé. La figure (Fig.49) représente un exemple du problème de dîner de philosophes

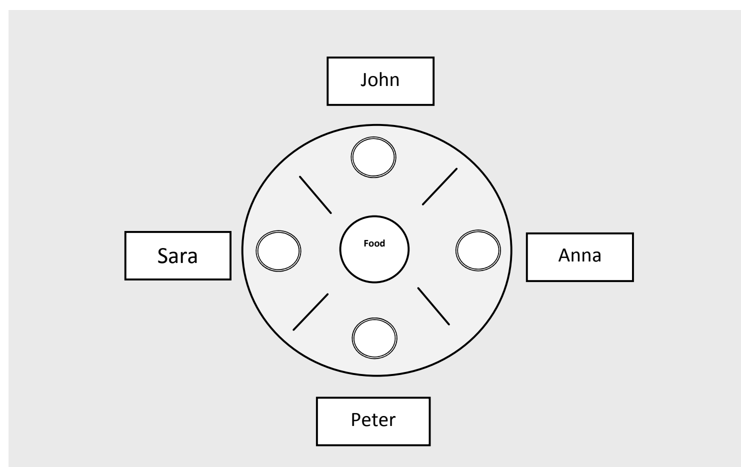


Fig.49 : Problème de dîner des philosophes

2.1 Modèle UML pour le problème de dîner des philosophes

Un des modèles UML qui sont conçu pour représenté le problème de dîner de philosophes et celui proposé en [66]. Nous appliquons notre méthode de vérification sur ce modèle UML qui est composé de diagrammes suivants :

- **Diagramme de classe** : la structure statique du modèle est spécifiée par le diagramme de classes représenté dans la figure (Fig.50). le diagramme contient deux classes; la classe **Philosopher** pour représenter les philosophes et la classe **Fork** pour représenter les fourches.

La classe philosopher contient deux méthodes **eat** () et **think** () pour manger et penser. La classe fork contient deux méthodes **get** () et **release** () pour prendre et déposer la fourche.

Deux associations sont définies, l'association **left** et **right** pour modéliser le voisinage des philosophes et des fourches.

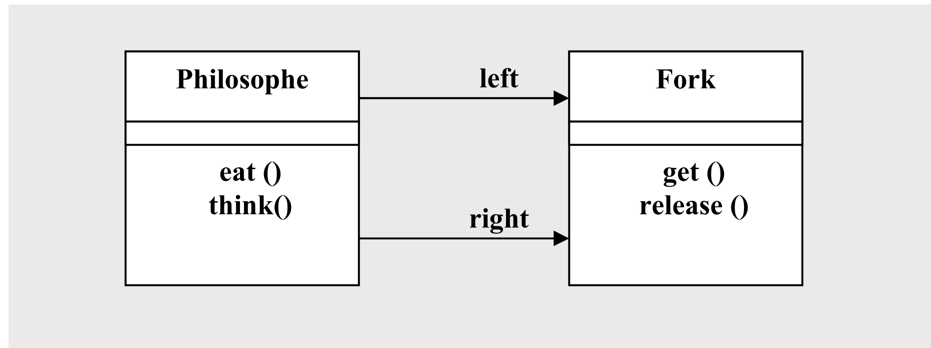


Fig.50 : Diagramme de Classe de problème de dîner des philosophes [66]

- **diagramme d'états-transitions:** pour chaque classe il est associé un diagramme d'états-transition qui décrive le comportement des ses objets instances. Ainsi, nous avons deux diagrammes d'états pour la classe Philosopher et la classe Fork.

  Le diagramme d'état-transition de la classe Philosopher est représenté dans la figure (Fig.51). Un objet philosophe peut être à l'état "penser" (**Thinking**), il passe à l'état d'attente de son voisin gauche (**waitingLetf**), quand il prend la fourche droite. L'appel get() est un appel bloquant et le philosophe est bloqué jusqu'à ce qu'il détient la fourche. L'objet philosophe passe à l'état manger (**Eating**) une fois il prend la fourche gauche. Quand il terminer manger, il passe à l'état de libération des deux fourches (**Releasing**). A la fin, il revient à l'état **thinking** à nouveau.

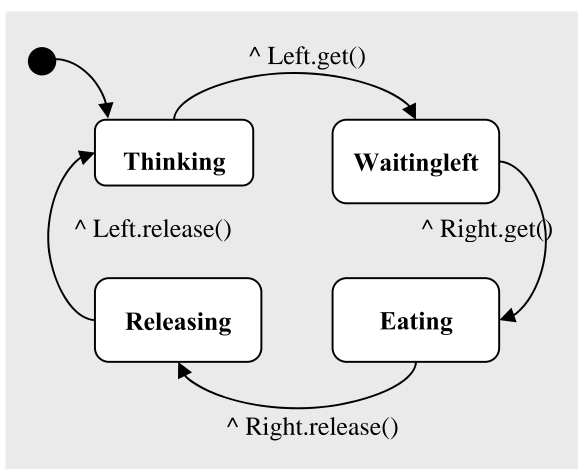


Fig.51 : Diagramme d'états-transitions de la classe Philosopher [66]

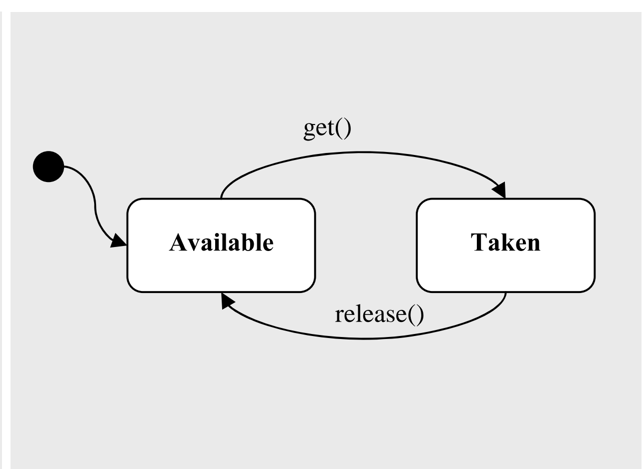


Fig.52 : diagramme d'états-transitions de la classe Fork [66]

La figure (Fig.52) représente le diagramme d'états-transition associé à la classe Fork. Une fourche selon ce diagramme passe de l'état initial à l'état disponible (**Available**). Elle change son état **Available** vers l'état **Taken** avec la présence d'un événement **get**(). Elle retourne à l'état **Available** si elle est déposé à l'exécution de la méthode **release ().**

- **Diagramme de collaboration :** La figure (Fig.53) montre un diagramme de collaboration de quartes philosophes partageant quatre fourches. Les philosophes s'interagissent entre eux via l'utilisation des fourches.

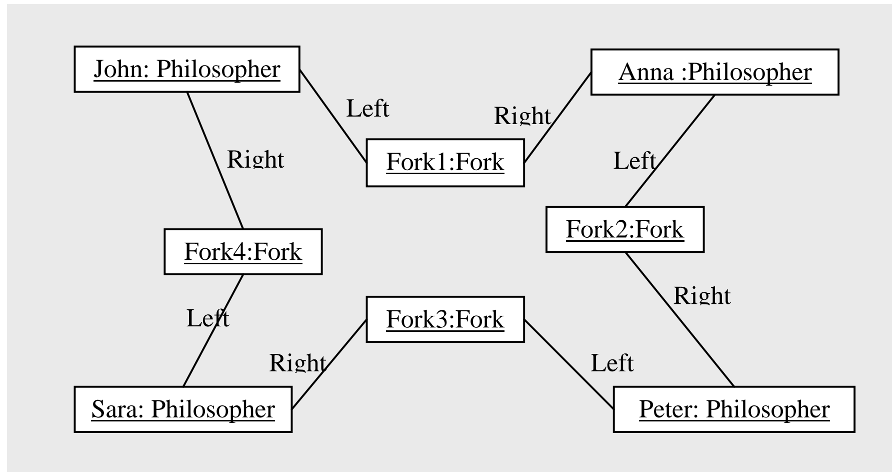


Fig.53 : diagramme de Collaboration [66]

Un diagramme de collaboration permet de décrire un ou plusieurs comportements désirés dans le système. Toutefois, quelques comportements non désirés peuvent avoir lieu lors d'exécution du système. Le diagramme de collaboration, présenté dans la figure (Fig.53), peut exhiber un comportement non désiré et met le système dans un état d'interblocage (deadlock). L'instance présentée dans la figure (Fig.54) représente le scénario qui engendre cet interblocage.

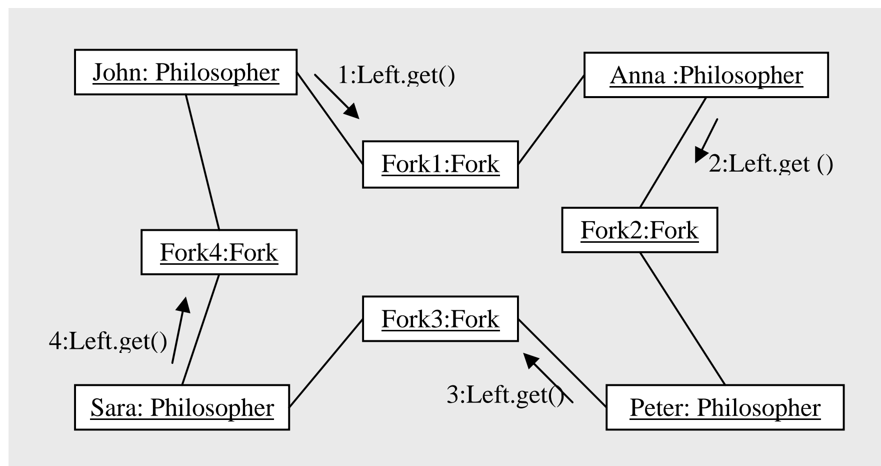


Fig.54 : diagramme de collaboration présentant un Deadlock [66]

Dans les sections suivantes, nous allons traduire ce modèle à une spécification Maude, sur laquelle nous allons vérifier des propriétés temporelles.

2.2 Translation du Modèle UML

La deuxième étape qui suit l'élaboration du modèle est l'étape de translation du dernier à une spécification Maude. Cette opération s'effectue par la construction de deux modules Orientés objet (**SMFork**) et (**SMPhilosopher**) respectivement pour la classe Fork et son diagramme d'états et la classe Philosopher et son diagramme d'états-transition. La translation s'effectuée comme suit :

- **Translation des classes et d'attributs** : la classe Philosopher (Fig.49) est translatée sous la forme :

  class Philosopher | STATUS : STATE, BLOCKED : BOOL , RIGHT : Oid , LEFT : Oid "

  Où l'attribut **left** et **right** indiquant les voisinages du philosophe, et ce sont exactement les liens représentés dans le diagramme de classes.
  La classe fork est translaté de la même façon que la classe philosopher. Elle a sous Maude la forme :

  Class Fork | STATUS : STATE , BLOCKED : BOOL , RIGHT : Oid , LEFT : Oid.

- **Translation des états :** les états dans le diagramme d'état-transition associé à la classe Philosopher, sont déclarés comme suit :

  op INITIALSTATE THINKING WAITINGLEFT EATING RELEASING : -> SIMSTATE

  Tous ces états sont des états simples. De la même façon on déclare les états du diagramme d'états-transitions de la classe Fork:

  op INITIALSTATE FINALSTATE TAKEN AVAILABLE : -> SIMSTATE .

- **Translation des événements** : les événements apparurent lors de l'invocation des méthodes ou l'envoi des signaux. Ainsi, nous remplaçons chaque méthode ou signal par un message dans la spécification Maude. Par conséquent, nous avons pour les deux classes, les déclarations suivantes:

  msg get(_,_) : Oid Oid -> Msg .

  msg release(_,_) : Oid Oid -> Msg .

  L'appel aux méthodes get et release est un appel bloquant, pour débloquer l'appelant on est besoin de spécifier un Ack_Evenement qui est déclarés comme suit:

  msg ACKget(_,_) : Oid Oid -> Msg

  msg ACKrelease(_,_) : Oid Oid -> Msg .

- **Translation des Transitions :** les transitions des deux diagrammes d'état-transitions sont écrites en Maude Comme il est présenté dans les lignes qui suivent l'étiquette ***(12) dans les figure (Fig.55) et (Fig.56). ces deux dernières figures montrent les deux modules Orienté objets SMFork et SMPhilosophe qui sont le résultat de la translation du modèle UML.

Avant de passer à la définition des propriétés à vérifier, nous devrons définir un Module global qui représente le système. Ce module inclut les deux modules SMFork, SMphilosopher et l'état initial du système. L'état initial est extrait à partir du diagramme de collaboration (Fig.53). Le module global MCHECK est représenté dans la figure (Fig.57). L'étiquettes *** (1) et *** (2) montrent l'inclusion des modules SMFork et SMphilosopher, et l'étiquette *** (3) montre la déclaration de l'état initial du système.

```
load full-maude
( omod SMPhilosopher is
protecting NAT .
protecting STRING .
protecting BOOL .
*** definition des sortes STATE SIMSTATE et COMSTATE
sorts  SIMSTATE  COMSTATE  STATE .                                              ***(1)
subsort SIMSTATE < COMSTATE .                                                   ***(2)
subsort COMSTATE < STATE.                                                       ***(3)
op none : -> COMSTATE [ctor] .                                                  ***(4)
op _||_ : COMSTATE COMSTATE : -> COMSTATE [ctor assoc id:none].                 ***(5)

*** pour donner des nom aux objets.
subsort STRING < Oid .

*** définition de la classe philosopher
class Philosopher | STATUS : STATE , BLOCKED : BOOL , RIGHT : Oid , LEFT : Oid .   ***(6)

*** déclaration des différente état dans le diagramme d'états-transitions
op INITIALSTATE  THINKING WAITINGLEFT EATING RELEASING : -> SIMSTATE .          ***(7)

*** Déclaration des événements
msg get(_,_) : Oid Oid -> Msg .                                                 *** (8)
*** un appel bloquant nécessite un message ack_Msg qui informe l'appelant que sont messages et consommé   .
msg ACKget(_,_) : Oid Oid -> Msg .                                              *** (9)
msg release(_,_) : Oid Oid -> Msg .                                             *** (10)
msg ACKrelease(_,_) : Oid Oid -> Msg .                                          *** (11)

***déclaration de variables
vars ph x1 x2 cal : STRING .
var CF : Configuration.

***règles de réécriture qui représentent les transitons dans le diagramme d'états-transition   ***(12)
rl [t1] : <ph : Philosopher | STATUS : INITIALSTATE > CF => <ph : Philosopher | STATUS : THINKING > CF .

rl [t2a] : <ph : Philosopher | STATUS : INITIALSTATE , BLOCKED : FALSE, LEFT : x1 > CF => <ph : Philosopher |
STATUS : WAITINGLEFT , BLOCKED : TRUE, LEFT : x1 > get(ph , x1) CF .
rl [t2b] : ACKget(x1 , ph) <ph : Philosopher | STATUS : WAITINGLEFT, BLOCKED : TRUE, LEFT : x1 > CF => <ph :
Philosopher | STATUS : WAITINGLEFT, BLOCKED : FALSE, LEFT : x1 > CF .

rl [t3a] : <ph : Philosopher | STATUS : WAITINGLEFT, BLOCKED : FALSE, RIGHT : x2 > CF => <ph : Philosopher |
STATUS : EATING, BLOCKED : TRUE, RIGHT : x2 > get(ph , x2) CF .
rl [t3b] : ACKget(x2 , ph) <ph : Philosopher | STATUS : EATING, BLOCKED : TRUE, RIGHT : x2 > CF => <ph :
Philosopher | STATUS : EATING , BLOCKED : FALSE, RIGHT : x2 > CF .

rl [t4a] : <ph : Philosopher | STATUS : EATING, BLOCKED : FALSE, RIGHT : x2 > CF => <ph : Philosopher | STATUS :
RELEASING, BLOCKED : TRUE, RIGHT : x2 > release(ph , x2) CF .
rl [t4b] : ACKrelease(x2 , ph) <ph : Philosopher | STATUS : RELEASING, BLOCKED : TRUE, RIGHT : x2 > CF => <ph :
Philosopher | STATUS : RELEASING , BLOCKED : FALSE, RIGHT : x2 > CF .

rl [t5a] : <ph : Philosopher | STATUS : RELEASING , BLOCKED : FALSE, LEFT : x1 > CF => <ph : Philosopher |
STATUS : THINKING, BLOCKED : TRUE, LEFT : x1> release(ph , x1) CF .
rl [t5b] : ACKrelease(x1 , ph) <ph : Philosopher | STATUS : THINKING, BLOCKED : TRUE, LEFT : x1> CF => <ph :
Philosopher | STATUS : THINKING, BLOCKED : FALSE, LEFT : x1> CF .

endom)
```

Fig.55 : spécification Maude pour la classe Philosopher et son diagramme d'états-transitions

```
load full-maude
( omod SMFork is
protecting NAT .
protecting STRING .
protecting BOOL .

*** definition des sortes STATE SIMSTATE et COMSTATE
sorts  SIMSTATE  COMSTATE  STATE .                                        ***(1)
subsort SIMSTATE < COMSTATE .                                             ***(2)
subsort COMSTATE < STATE.                                                 ***(3)
op none : -> COMSTATE [ctor] .                                            ***(4)
op _||_ : COMSTATE COMSTATE : -> COMSTATE [ctor assoc id:none].           ***(5)

*** pour donner des nom aux objets.
subsort STRING < Oid .

*** Ddéclaration de la classe Fork
class Fork | STATUS : STATE , BLOCKED : BOOL , RIGHT : Oid , LEFT : Oid .  *** (6)

*** déclaration des différente état dans le diagramme d'états-transitions
op INITIALSTATE  FINALSTATE TAKEN AVAILABLE : -> SIMSTATE .               *** (7)

*** Déclaration des événements
msg get(_,_) : Oid Oid -> Msg .                                           *** (8)
*** un appel bloquant nécessite un message ack_Msg qui informe l'appelant que sont messages et consommé   .
msg ACKget(_,_) : Oid Oid -> Msg .                                        *** (9)
msg release(_,_) : Oid Oid -> Msg .                                       *** (10)
msg ACKrelease(_,_) : Oid Oid -> Msg .                                    *** (11)

*** déclaration des variables
vars fk x1 x2 cal : STRING .
var CF : Configuration.

***règles de réécriture qui représentent les transitons dans le diagramme d'états-transition   ***(12)
rl [t1] : <fk : fork | STATUS : INITIALSTATE> CF => <fk : fork | STATUS : AVAILABLE > CF .

rl [t2] : get(cal , fk) <fk : fork | STATUS : AVAILABLE > CF => <fk : fork | STATUS : TAKEN > ACKget(fk , cal) CF .

rl [t3] : release(cal , fk) <fk : fork | STATUS : TAKEN > CF =>  <fk : fork | STATUS : AVAILABLE > ACKrelease(fk ,
cal) CF .

endom)
```

Fig.56 : spécification Maude qui représente la classe Fork et son diagramme d'états-transitions

2.3 Définition des propriétés à vérifier

Il s'agit, dans cette étape, de définir les propriétés à vérifier. Ainsi, nous allons proposer un ensemble de propriétés que leur vérification indique si le modèle UML contient des erreurs ou pas. Nous allons commencer par la déclaration des propositions suivantes :

- La proposition **think(Name)** : cette proposition est évaluée à vraie lorsque le philosophe «Name» entre à l'état **Thinking**. Cette proposition est déclarée en MCHECK comme suit :

```
op think: String -> Prop .
eq < name : Philosopher | STATUS : THINKING , BLOCKED : false > CCF  |=
think(name) = true .
```

- La proposition **eat(Name)** : cette proposition est évaluée à vraie lorsque le philosophe «Name» entre à l'état **Eating**. Cette proposition est déclarée en MCHECK comme suit :

```
op eat: String -> Prop .
eq < name : Philosopher | STATUS : EATING , BLOCKED : false > CCF  |=
eat(name) = true .
```

- La proposition **available (Name)** : cette proposition est évaluée à vraie lorsque la fourche «Name» est dans l'état **Available**. Cette proposition est déclarée en MCHECK comme suit :

```
op available: String -> Prop .
eq < name : Fork | STATUS : AVAILABLE , BLOCKED : false > CCF |= available
(name) = true .
```

- La proposition **taken(Name)** : cette proposition est évaluée à vraie lorsque la fourche «Name» est dans l'état **Taken**. Cette proposition est déclarée en MCHECK comme suit :

```
op taken: String -> Prop .
eq < name : Fork | STATUS : TAKEN , BLOCKED : false > CCF |= available
(name) = true .
```

Ces propositions sont déclarées en dessous de l'étiquette 4 dans le module MCHECK, voir l'étiquette *** (4) dans (Fig.57).

Les formules logiques que nous allons vérifier sur notre exemple sont :

1- **~ [] think(philosophe) :** cette propriété permet de vérifier qu'un philosophe ne reste jamais infiniment dans l'état thinking (il pense et il change d'états).
2- **~ [] eat(philosophe) :** cette propriété permet de vérifier qu'un philosophe ne reste jamais infiniment dans l'état eating (il mange et il change d'états).
3- **~ [] available (fork) :** cette propriété permet de vérifier qu'un fourche ne reste jamais infiniment dans l'état available.
4- **~ [] taken (fork) :** cette propriété permet de vérifier qu'un fourche ne reste jamais infiniment dans l'état Taken.
5- **[] ~ (eat("John") /\ eat("Anna") /\ eat("Peter") /\ eat("Sara")) :** cette propriété permet de vérifier l'exclusion mutuelle.
6- **~ [](eat("John") \/ eat("Anna") \/ eat("Peter") \/ eat("Sara")) :** cette propriété permet de vérifier la présence d'inter-blocage (deadlock).
7- **[](eat("John") \/ eat("Anna") \/ eat("Peter") \/ eat("Sara")) :** cette propriété permet de vérifier l'absence d'inter-blocage.

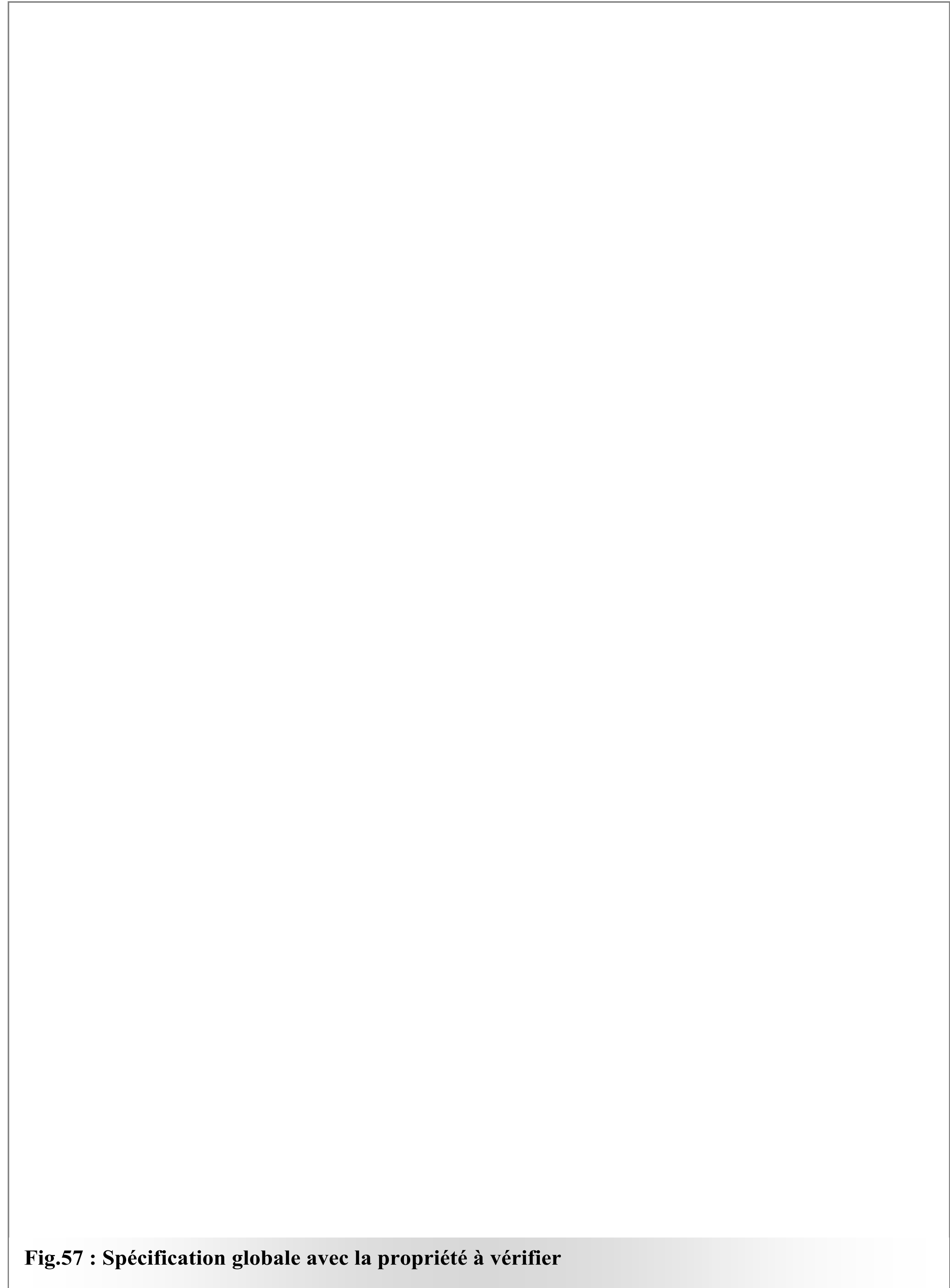


Fig.57 : Spécification globale avec la propriété à vérifier

8- **[] (eat(philosophe) => <> release(philosophe))**: cette propriété garantie qu'un philosophe quand il mange, il fait une suite d'actions puis il passe à l'état penser.
9- **[] (think(philosophe) => <> eat(philosophe))**: cette propriété garantie qu'un philosophe pense et fait une suite d'actions puis il passe à l'état manger.
10- **[] <> ~ eat(philisophe):** cette propriété permet de vérifier le séjour fini en section critique (utilisation de deux fourches).
11- **[] (think(philosophe) <=> ~ O eat(philosophe)):** cette propriété permet de vérifier l'alternance entre les deux états manger et penser.

L'objectif principal derrière la définition de ces propriétés est la vérification de la présence d'un Deadlock dans le modèle UML. La vérification de ces propriétés est le sujet de la prochaine section.

2.4 Vérification du modèle

Apres l'ouverture d'une session Maude, les commandes suivantes sont introduites afin de vérifier les propriétés définies en dessus.

```
1:  (red modelCheck(initState, ~ [] (eat(philo:String))).)
2:  (red modelCheck(initState, ~ [] (think(philo:String))).)
3:  (red modelCheck(initState, ~ [] available(fork:String)).)
4:  (red modelCheck(initState, ~ [] taken(fork:String)).)
5:  (red modelCheck(initState, [] ~ (eat("John") /\ eat("Anna") /\
    eat("Peter") /\ eat("Sara"))).)
6:  (red modelCheck(initState, ~[](eat("John") \/ eat("Anna") \/ eat("Peter")
    \/ eat("Sara"))) .)
7:  red modelCheck(initState, [] (eat("John") \/ eat("Anna") \/
    eat("Peter") \/ eat("Sara") ) ) .)
8:  (red modelCheck(initState, [] (eat("John") => <> release("John"))).)
9:  (red modelCheck(initState, [] (think("John") => <> eat("John"))).)
10: (red modelCheck(initState, [] <> ~ (eat("John"))).)
11: (red modelCheck(initState, [] (think("John") <=> ~ O eat("John"))).)
```

L'exécution du model-checking s'effectue par l'ouverture d'une session Maude et l'introduction du module MCHECK. Puis l'introduction des commandes de vérification citées en dessus.

Les résultats du model-checking de ces propriétés par rapport au Modèle UML sont :

- **Propriétés individuelles** : le résultat de model-checking des propriétés 1, 2, 3 et 4 est **True**. Ce qui signifie qu'un objet philosophe ne soit pas infiniment en un état de manger ou de penser. et qu'un objet fourche n'a pas uniquement un seul état durant toute ça vie. L'objectif derrière ces propriétés est de vérifier que le système est dynamique et ses objets changent d'états. La figure (Fig.58) représente le résultat d'exécution de ces propriétés.

okba@localhost: /home/okba/maude-linux - Terminal - Konsole

Session Édition Affichage Signets Configuration Aide

```
rewrites: 1027 in 20ms cpu (14ms real) (51350 rewrites/second)
reduce in MCHECK :
  modelCheck(initState,~[]eat(philo:String))
result Bool :
  true

rewrites: 1026 in 0ms cpu (3ms real) (~ rewrites/second)
reduce in MCHECK :
  modelCheck(initState,~[]think(philo:String))
result Bool :
  true

rewrites: 1003 in 0ms cpu (3ms real) (~ rewrites/second)
reduce in MCHECK :
  modelCheck(initState,~[]available(fork:String))
result Bool :
  true

rewrites: 1002 in 10ms cpu (3ms real) (100200 rewrites/second)
reduce in MCHECK :
  modelCheck(initState,~[]taken(fork:String))
result Bool :
  true

Maude>
```

Terminal

Fig.58 : résultat de model-checking de quelques propriétés temporelles

- **Propriété d'exclusion mutuelle :** la propriété 5 (**[] ~ (eat("John") /\ eat("Anna") /\ eat("Peter") /\ eat("Sara")**)) permet de vérifier l'exclusion mutuelle (l'utilisation des fourches) entre les quatre philosophes. Le résultat de vérification de cette propriété est **True**, voir figure (Fig.59). ce résultat permet de conclure que le la modèle permet d'assurer l'exclusion mutuelle. Mais est ce qu'il permet le non famine de tous philosophes ? c'est les prochaines propriétés qui répondent à cette question.

```
rewrites: 5484 in 50ms cpu (51ms real) (109680 rewrites/second)
reduce in MCHECK :
  modelCheck(initState,[]~(eat("John")/\ eat("Anna")/\ eat("Peter")/\ eat("Sara")))
result Bool :
  true
```

Fig.59 : résultat de model-checking de propriété d'exclusion mutuelle

- **Propriétés de présence d'interblocage (Deadlock) :** nous avons vu que le système assure l'exclusion mutuelle pour l'utilisation des fourches. La propriété 6 (**~ [] (eat("John")** \/ **eat("Anna")** \/ **eat("Peter")** \/ **eat("Sara")**)) permet de vérifier le blocage dans le système. Le résultat de la propriété est vrai (True), voir figure (Fig.60).

```
rewrites: 1213 in 10ms cpu (4ms real) (121300 rewrites/second)
reduce in MCHECK :
  modelCheck(initState,~[](eat("John")\/ eat("Anna")\/ eat("Peter")\/ eat("Sara")))
result Bool :
  true
```

Fig.60 : résultat de model-checking de propriété de la présence de blocage (Deadlock)

- **Propriétés d'absence d'interblocage (Deadlock) :** supposons que nous n'avons pas évalué la propriété précédente mais nous avons la propriété inverse (absence de blocage). Cette dernière est la propriété 7 (**[] (eat("John")** V **eat("Anna")** V **eat("Peter")** V **eat("Sara"))**). Le résultat est un contre exemple, voir figure (fig.61). par conséquent, la propriété d'absence d'interblocage est non satisfaite dans le système.

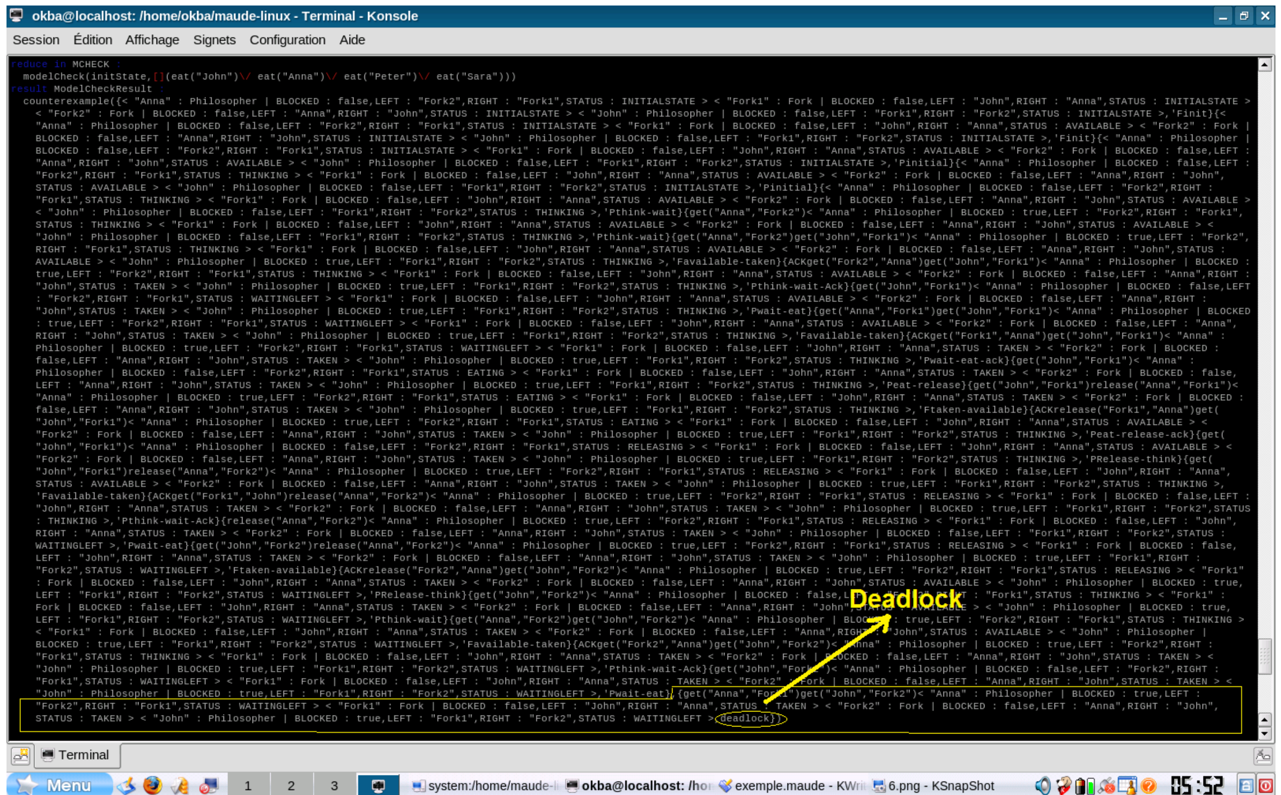


Fig.61 : résultat de model-checking de propriété d'absence de blocage (Deadlock) : un contre exemple

La dernière partie du contre exemple est :

{get("Anna","Fork1") get("John","Fork4") get("Peter","Fork2") get("Sara","Fork3") <"Anna" : Philosopher | BLOCKED : true, LEFT : "Fork2",RIGHT : "Fork1",STATUS : WAITINGLEFT > < "Fork1" : Fork | BLOCKED : false, LEFT : "John", RIGHT : "Anna", STATUS : TAKEN > < "Fork2" : Fork | BLOCKED : false, LEFT : "Anna", RIGHT : "Peter", STATUS : TAKEN > < "Fork3" : Fork | BLOCKED : false, LEFT : "Peter", RIGHT : "Sara", STATUS : TAKEN > < "Fork4" : Fork | BLOCKED : false, LEFT : "Sara", RIGHT : "John", STATUS : TAKEN > < "John" : Philosopher | BLOCKED : true, LEFT : "Fork1",RIGHT : "Fork4",STATUS : WAITINGLEFT > < " Peter" : Philosopher | BLOCKED : true, LEFT : "Fork3",RIGHT : "Fork2", STATUS : WAITINGLEFT > < "Sara" : Philosopher | BLOCKED : true, LEFT : "Fork4",RIGHT : "Fork3",STATUS : WAITINGLEFT >,deadlock})

Cette partie représente l'état du blocage dans le système ou chaque philosophe détient une fourche et demande une deuxième. Les message get() dans la figure représentent les demandes. L'interpréteur Maude détecte qu'il s'agit d'un interblocage et il affiche à la fin du contre exemple le mot Deadlock. En résumé, le modèle UML qui modélise le problème de diner de philosophe n'assure pas l'interblocage des objets. Cette solution n'assure pas le non famine de tous les philosophes. A ce point, nous pouvons arrêter le modèle checking. Toutefois, nous allons terminer avec les propriétés qui nous restent.

- **Propriétés de réponse :** la propriété 8 **([] (eat("John") => <> release("John")))** permet de vérifier qu'un philosophe passe toujours de l'état manger à l'état de libération des fourches. Le résultat de vérification de cette propriété est Vrai, voir figure (Fig.62).

```
rewrites: 5367 in 50ms cpu (51ms real) (107340 rewrites/second)
reduce in MCHECK :
  modelCheck(initState,[](eat("John")=> <> release("John")))
result Bool :
  true
```

Fig.62 : résultat du model-checking de propriété N° 8

Par contre, la vérification de la propriété 9 **([] (think("John") => <> eat("John")))** donne un contre exemple ce qui signifier que la propriété n'est pas toujours satisfaite à cause d'interblocage.

La propriété 10 (**[]** <> ~ **eat(philisophe)**) permet de vérifier le séjour fini en section critique. En autres termes, elle permet de vérifier qu'un objet philosophe ne se bloque jamais dans un état de manger. Le résultat de cette propriété est **True**, voir figure (Fig.63).

```
rewrites: 5300 in 50ms cpu (49ms real) (106000 rewrites/second)
reduce in MCHECK :
  modelCheck(initState,[]<> ~ eat("John"))
result Bool :
  true
```

Fig.63 : résultat du model-checking de propriété N° 9

- **Propriétés d'alternance :** la propriété 11 (**[] (think(philosophe)** <=> ~ O **eat(philosophe)**)) permet de vérifier l'alternance entre les deux états think et eat. Le résultat est évidement un contre exemple parce qu'un objet philosophe peut avoir plus de deux états et le comportement de système n'exclue pas la présence de blocage.

Le model-checking de la solution proposée (modèle UML) pour le problème de dîner des philosophes à montrer que la solution assure l'utilisation des ressources (fourches) en exclusion mutuelle. Toutefois, l'inter-blocage peut se manifeste à n'importe quel moment dans le comportement du système (vérification des propriétés 6 et 7).

L'objectif de cet exemple est d'appliquer la méthode que nous avons proposée sur un modèle UML. Nous avons vu comment on peut translater le modèle UML à une spécification Maude, comment on définit les propriétés temporelles, et finalement comment on vérifié ces propriétés avec le LTL model-checker du langage Maude.

Le prochain exemple, présente un modèle plus complexe avec lequel on applique une technique de vérification des collaborations vis-à-vis le comportement de ses objets. Cette technique est basée toujours sur notre méthode de vérification des modèle UML en utilisant Maude et son LTL model-checker.

3. Exemple II : ATM-BANK

Notre deuxième exemple et celui présenté dans [62], c'est un exemple d'un ATM (Automatic Teller Machine). Le principe de base de se système est le suivant : quand l'utilisateur introduit sa carte, l'ATM lui demande de saisir son code PIN. L'ATM transfert les données relatives à la carte et le code PIN à la Bank afin de les vérifier. Une fois la carde est valide et le code PIN est correct, l'utilisateur peut introduire la somme qu'il veule retirer et l'ATM le vire cette somme s'il la possède. Dans le cas où le code PIN est erroné, l'ATM demande à l'utilisateur de ré-entrer son code PIN. La carte ne sera plus valide après la saisie de trois codes PIN erronés et L'ATM rejette la carte après la troisième tentative.

3.1 Modèle UML ATM-BANK

Le modèle UML qui représente le système est modélisé comme suit:

- **Diagramme de classe** : la structure statique du modèle est représentée par le diagramme de classes représenté dans la figure (Fig.64). ce diagramme est composé de deux classes ; la classe ATM et la classe Bank. Les deux classes sont reliées par l'association Link.

  Trois signaux sont déclarés dans la classe ATM, **PINVerified**, **reenterPIN** et **abort**. La classe Bank contient trois attributs ; le booléen **cardValid**, les deux entiers **MaxNumIncorrect** et **numIncorrect**. La classe Bank est composée aussi de la méthode **verifyPIN** et le signal **done**.

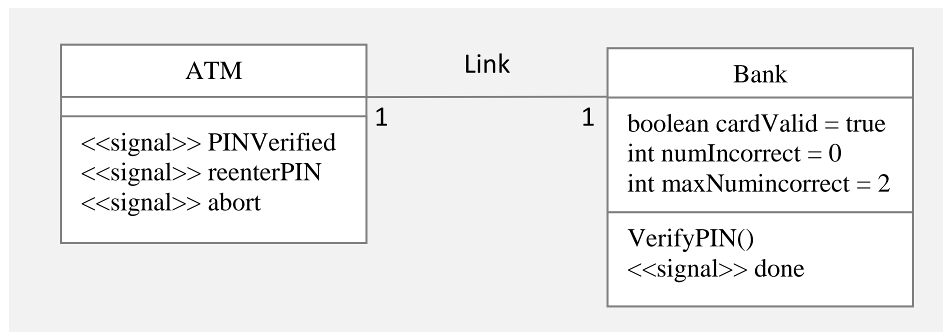


Fig.64 : Diagramme de classe du modèle ATM/Bank [62]

- **Diagramme d'états-transition** : Le comportement de l'objet ATM est spécifié par la machine d'états-transitions présenté dans la figure (Fig.65). ce diagramme est composé des états simple **CardEntry**, **PINEntry**, **Verification**, **AmountEntry**, **returningCard** et l'état composite **GivingMoney**.

  Le comportement de l'objet Bank est présenté dans le diagramme d'états-transitions figuré dans (Fig.66). Ce diagramme est composé de l'état **idle** et l'état composite Verifying. Ce dernier est composé de deux régions concurrentes qui spécifient respectivement la vérification de la carte et le code PIN.

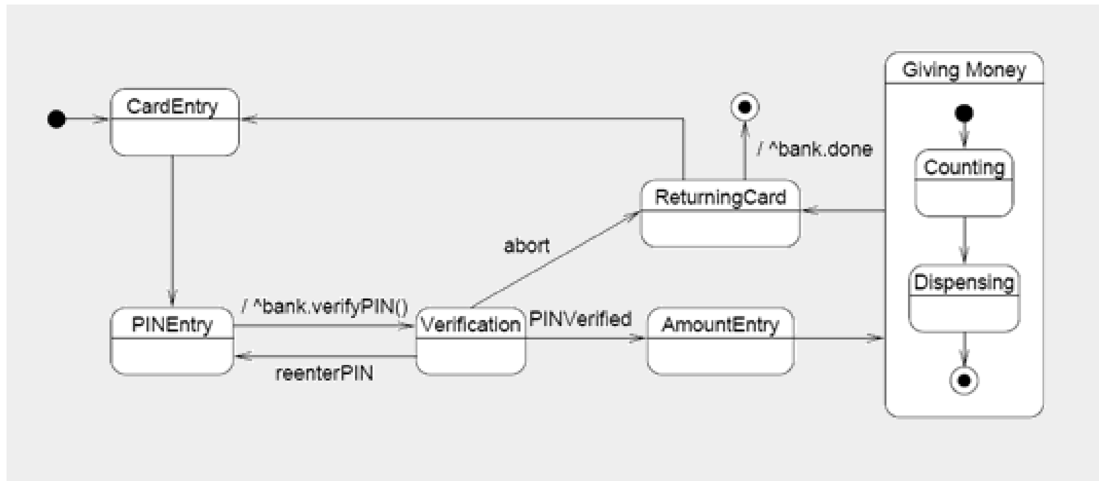


Fig.65 : Diagramme d'état-transition de la classe ATM [62]

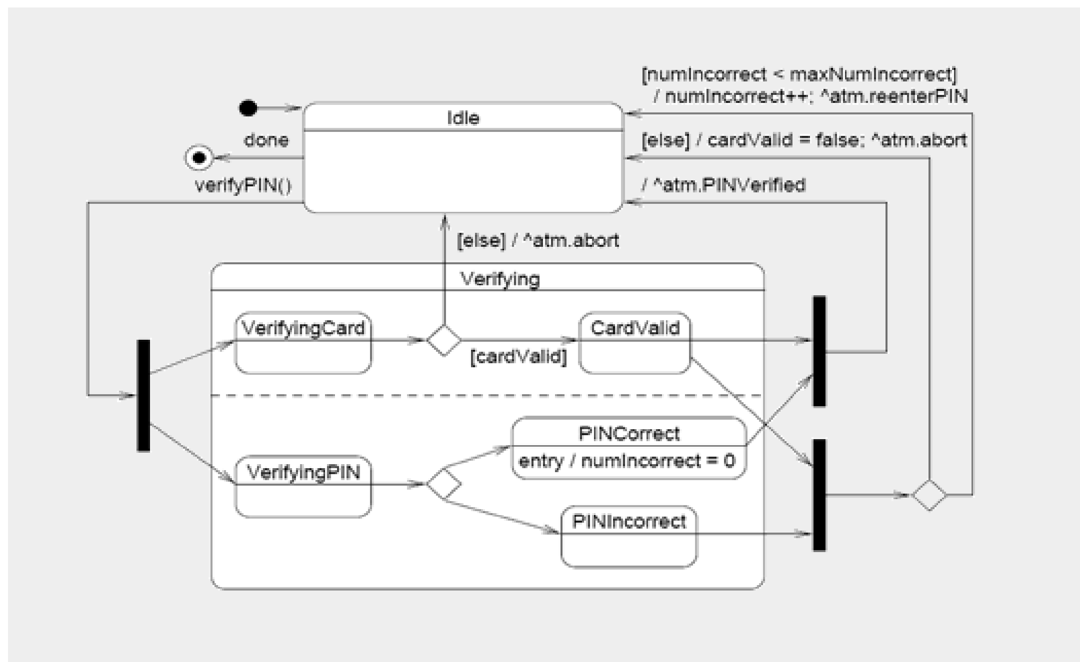


Fig.66 : Diagramme d'état-transition de la classe BANK [62]

- **Diagrammes de collaborations :** Les concepteurs du modèle proposent deux diagrammes de collaboration qui représentent des scénarios différents d'interactions entre l'ATM et la Bank. Le diagramme représenté dans la figure (Fig.67) spécifie une collaboration juste c.-à-d. une collaboration dont l'ordre d'interactions peut se produire dans une exécution tel qu'il est prévu dans le diagramme de collaboration.

  Par contre, le diagramme de collaboration figuré dans (Fig.68) présente une collaboration erronée dont l'ordre d'interactions entres les objets ne peut jamais se produire dans le

système réel à cause du comportement des objets (les spécifications des diagrammes d'états-transitions).

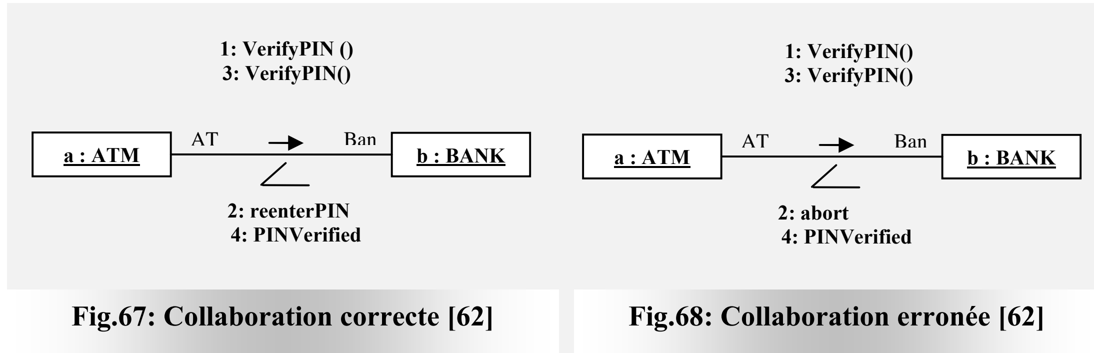


Fig.67: Collaboration correcte [62] **Fig.68: Collaboration erronée [62]**

Nous allons montrer à travers cet exemple une technique de vérification des collaborations vis-à-vis le comportement des objets spécifie par des diagrammes d'états-transitions. Cette technique est basée sur notre méthode de translation et de vérification.

3.2 Translation du Modèle UML

La translation du modèle UML s'effectué de la même manière que l'exemple précédent. La figure (Fig.69) représente le module Maude qui spécifie la classe ATM et son diagramme d'états-transition. Et la figure (Fig.70) représente le module Maude qui spécifie la classe Bank et son diagramme d'états.

La translation montre que notre méthode tient compte aux concepts avancés du diagramme d'états-transitions tels que les états concurrents, les états séquentiels, les transitions complexes qui utilisent le join et le fork et les choix …etc. et les règles de réécritures suivantes représente la transition complexe (**T6**) dans le diagramme d'états-transitions de la Bank (Fig.66);

```
crl [T6A] : < BQ : BANK | STATUS : Verifying( CardValid || PINIncorrect ) ,
cardValid : true , numIncorrect : N , MaxNumIncorrect : N' , Link : L >
collaboration(abort(BQ , L ) - s) => < BQ : BANK | STATUS : Idle , cardValid
: false , numIncorrect : N , MaxNumIncorrect : N' , Link : L > abort(BQ , L
) collaboration(s) if N >= N' .

crl [T6B] : < BQ : BANK | STATUS : Verifying( CardValid || PINIncorrect ) ,
cardValid : b , numIncorrect : N , MaxNumIncorrect : N' , Link : L >
collaboration(reenterPIN(BQ , L ) - s) => < BQ : BANK | STATUS : Idle ,
cardValid : b , numIncorrect : N + 1 , MaxNumIncorrect : N' , Link : L >
reenterPIN(BQ , L ) collaboration(s) if N < N' .
```

Tant qu'il s'agit de vérifier des collaborations de cette exemple, nous ne somme pas besoin de définir des propriétés temporelles. Nous allons juste appliquer ce que nous avons expliqué précédemment sur la vérification de collaboration. Ainsi, nous passons directement à la quatrième étape.

```
load full-maude

( omod SMATM is
protecting INT .  protecting NAT . protecting STRING . protecting BOOL .
*** Objects names are Strings.
subsort String < Oid .
*** definition of the sort state simple state and composed state
sorts  SIMSTATE  COMSTATE  STATE .                                          *** [1]
subsort SIMSTATE < COMSTATE .                                               *** [2]
subsort COMSTATE < STATE .                                                  *** [3]
op none : -> COMSTATE [ctor] .                                              *** [4]
op _||_ : COMSTATE COMSTATE -> COMSTATE [ctor assoc comm id: none]          *** [5]
*** Specification   of collaboration structure
sort seq col .                                                              *** [6]
subsort Msg < seq .                                                         *** [7]
op null : -> seq [ctor] .                                                   *** [8]
op _-_ : seq seq -> seq [ctor assoc id: null ] .                            *** [9]
op collaboration : seq -> col .
 subsort col < Configuration .                                              *** [10]
ops CardEntry PINEntry Verification AmountEntry ReturningCard Counting Dispensing InitialState FinalState : ->
SIMSTATE                                                                    *** [11]
op GivingMoney : COMSTATE -> STATE .                                        *** [12]
*** definition of  the ATM class
class ATM | STATUS : STATE , BLOCKED : Bool , Link : Oid .                  *** [13]
*** definition of events
msgs verifyPIN reenterPIN PINverified abort done
 ACKverifyPIN : Oid Oid -> Msg .                                            *** [14]
*** Some variables declaration
vars AT L : String . var X : STATE . vars N N' : Nat . var b : Bool . var s : seq .
var CF : Configuration .
*** transitions of the State machine are represented by Rewriting Logic rules
*** in order to draw a way to execute the system in respect to the given collaboration. the op Collaboration is added to
the rules
*** to get rules that represent only the transition remove collaboration.  *** [15]
rl [t1] : < AT : ATM | STATUS : InitialState >  => < AT : ATM | STATUS : CardEntry >  .
rl [t2] : < AT : ATM | STATUS : CardEntry >  => < AT : ATM | STATUS : PINEntry >  .
rl [t3] : < AT : ATM | STATUS : PINEntry , BLOCKED : false , Link : L > collaboration(verifyPIN(AT , L) - s) => < AT : ATM |
STATUS : Verification , BLOCKED : true , Link : L > verifyPIN(AT , L) collaboration(s) .
rl [t3ack] : ACKverifyPIN(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : true , Link : L >  => < AT : ATM | STATUS :
Verification , BLOCKED : false , Link : L >  .
rl [t4] : reenterPIN(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : false , Link : L >  => < AT : ATM | STATUS : PINEntry
, BLOCKED : false , Link : L >  .
rl [t5] : PINverified(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : false , Link : L >  => < AT : ATM | STATUS :
AmountEntry , BLOCKED : false , Link : L >  .
rl [t6] : abort(L , AT) < AT : ATM | STATUS : Verification , BLOCKED : false , Link : L >  => < AT : ATM | STATUS : ReturningCard
, BLOCKED : false , Link : L >  .
rl [t7] : < AT : ATM | STATUS : AmountEntry , BLOCKED : false , Link : L >  => < AT : ATM | STATUS : GivingMoney(InitialState) ,
BLOCKED : false , Link : L >  .
rl [t8] : < AT : ATM | STATUS : GivingMoney(InitialState) , BLOCKED : false , Link : L >  => < AT : ATM | STATUS :
GivingMoney(Counting) , BLOCKED : false , Link : L >  .
rl [t9] : < AT : ATM | STATUS : GivingMoney(Counting) , BLOCKED : false , Link : L >  => < AT : ATM | STATUS :
GivingMoney(Dispensing) , BLOCKED : false , Link : L >  .
rl [t10] : < AT : ATM | STATUS : GivingMoney(Dispensing) , BLOCKED : false , Link : L >  => < AT : ATM | STATUS :
GivingMoney(FinalState) , BLOCKED : false , Link : L >  .
rl [t11] : < AT : ATM | STATUS : GivingMoney(FinalState) , BLOCKED : false , Link : L >  => < AT : ATM | STATUS :
ReturningCard , BLOCKED : false , Link : L >  .
rl [t12] : < AT : ATM | STATUS : ReturningCard , BLOCKED : false , Link : L > collaboration( done(AT , L ) - s) => < AT : ATM |
STATUS : FinalState , BLOCKED : false , Link : L > done(AT , L ) collaboration(s) .
rl [t13] : < AT : ATM | STATUS : FinalState , BLOCKED : false , Link : L > CF => < AT : ATM | STATUS : FinalState , BLOCKED :
false , Link : L > CF .
endom )
```

Fig.69 : spécification Maude pour la classe ATM et sa machine d'état

```
load full-maude

( omod SMBank is
protecting INT .  protecting NAT . protecting STRING . protecting BOOL .

ops Idle VerifyingCard VerifyingPIN CardValid PINCorrect PINIncorrect InitialState FinalState : -> SIMSTATE .
                                                                                                    *** [11]
op Verifying : COMSTATE -> STATE .                                                                  *** [12]
*** definition  de la classe Bank
class BANK | STATUS : STATE , BLOCKED : Bool , cardValid : Bool , numIncorrect : Nat , MaxNumIncorrect : Nat ,
Link : Oid .                                                                                        *** [13]
*** definitions des événements.
msgs verifyPIN reenterPIN PINverified abort done
 ACKverifyPIN : Oid Oid -> Msg .                                                                    *** [14]
*** Some variables declaration
vars BQ L : String .  var X : STATE .  vars N N' : Nat . var b : Bool .
var s : seq . var CF : Configuration .

*** les règles de réécriture qui représentent les transitions du diagrammes d'états-transitions     *** [15]

rl [t1] : < BQ : BANK | STATUS : InitialState , BLOCKED : false >  => < BQ : BANK | STATUS : Idle , BLOCKED : false
>  .
rl [t2] : verifyPIN(L , BQ) < BQ : BANK | STATUS : Idle , BLOCKED : false , Link : L >  => < BQ : BANK | STATUS :
Verifying( VerifyingCard || VerifyingPIN ) , BLOCKED : false , Link : L > ACKverifyPIN(BQ , L)  .
crl [t3a] : < BQ : BANK | STATUS : Verifying( VerifyingCard || X ) , cardValid : b >  => < BQ : BANK | STATUS :
Verifying( CardValid || X ) , cardValid : b >  if b = true .
crl [t3b] : < BQ : BANK | STATUS : Verifying( VerifyingCard || X ) , cardValid : b , Link : L > collaboration( abort(BQ , L)
- s)  => < BQ : BANK | STATUS : Idle , cardValid : b > abort(BQ , L) collaboration(s) if b = false .
rl [t4a] : < BQ : BANK | STATUS : Verifying( X || VerifyingPIN ) >  => < BQ : BANK | STATUS : Verifying( X ||
PINIncorrect ) >  .
rl [t4a] : < BQ : BANK | STATUS : Verifying( X || VerifyingPIN ) , numIncorrect : N >  => < BQ : BANK | STATUS :
Verifying( X || PINCorrect ) , numIncorrect : 0 >  .
rl [t5] : < BQ : BANK | STATUS : Verifying( CardValid || PINCorrect ) , Link : L > collaboration(PINverified(BQ , L ) - s)
=> < BQ : BANK | STATUS : Idle , Link : L > PINverified(BQ , L ) collaboration(s) .
crl [t6] : < BQ : BANK | STATUS : Verifying( CardValid || PINIncorrect ) , cardValid : true , numIncorrect : N ,
MaxNumIncorrect : N' , Link : L > collaboration(abort(BQ , L ) - s) => < BQ : BANK | STATUS : Idle , cardValid : false ,
numIncorrect : N  , MaxNumIncorrect : N' , Link : L > abort(BQ , L ) collaboration(s) if N >= N' .
crl [t6] : < BQ : BANK | STATUS : Verifying( CardValid || PINIncorrect ) , cardValid : b , numIncorrect : N ,
MaxNumIncorrect : N' , Link : L > collaboration(reenterPIN(BQ , L ) - s) => < BQ : BANK | STATUS : Idle , cardValid : b ,
numIncorrect : N + 1 , MaxNumIncorrect : N' , Link : L > reenterPIN(BQ , L ) collaboration(s) if N < N' .
rl [t8] : done(L , BQ ) < BQ : BANK | STATUS : Idle , BLOCKED : false , Link : L >  => < BQ : BANK | STATUS :
FinalState , BLOCKED : false , Link : L >   .
endom)
```

Fig.70 : spécification Maude pour la classe Bank et sa machine d'états-transition

3.3 Vérification des Collaborations

Dans cette exemple nous ne vérifions pas des propriétés temporelles, mais plutôt nous vérifions si les deux collaborations (Fig.67, Fig.68) terminent leurs exécution par rapport aux spécifications des objets BANK et ATM. La figure (Fig.71) montre le module Maude qui représente la spécification globale du système.

La première collaboration est spécifiée en Maude par la structure suivante :

```
collaboration(verifyPIN("a","b") - reenterPIN("b","a") - verifyPIN("a","b") -
PINverified("b","a")).
```

La deuxième est déclarée comme suit :

```
collaboration(verifyPIN("a","b") - reenterPIN("b","a") - abort("a","b") - PINverified("b","a")).
```

Nous définissons deux états initiaux. Le premier état est défini pour vérifier la première collaboration où le deuxième est dédié à la deuxième collaboration. Le premier état est spécifié dans le module global comme suit :

```
eq initState-coll1 = < "a" : ATM | STATUS : InitialState , BLOCKED : false , Link : "b" > < "b" : BANK | STATUS : InitialState , BLOCKED : false , cardValid : true , numIncorrect : 0 , MaxNumIncorrect : 5 , Link : "a" > collaboration(verifyPIN("a" , "b") - reenterPIN("b" , "a") - verifyPIN("a" , "b") - PINverified("b" , "a") ) .
```

Le deuxième état est déclaré comme suit :

```
eq initState-coll2 = < "a" : ATM | STATUS : InitialState , BLOCKED : false , Link : "b" > < "b" : BANK | STATUS : InitialState , BLOCKED : false , cardValid : true , numIncorrect : 0 , MaxNumIncorrect : 5 , Link : "a" > collaboration(verifyPIN("a" , "b") - reenterPIN("b" , "a") - abort("a" , "b") - PINverified("b" , "a") ) .
```

```
load full-maude
( omod MCHECK is
including SMBank .        including SMAtm .
op initState-coll1 : -> Configuration .                                   *** [1]
eq initState-coll1 = < "a" : ATM | STATUS : InitialState , BLOCKED : false , Link : "b" > < "b" : BANK | STATUS :
InitialState , BLOCKED : false ,  cardValid : true ,  numIncorrect : 0 , MaxNumIncorrect : 5 , Link : "a" >
collaboration(verifyPIN("a" , "b") - reenterPIN("b" , "a") - verifyPIN("a" , "b") - PINverified("b" , "a")  ) .
op initState-coll2 : -> Configuration .                                   *** [2]
eq initState-coll2 = < "a" : ATM | STATUS : InitialState , BLOCKED : false , Link : "b" > < "b" : BANK | STATUS :
InitialState , BLOCKED : false , cardValid : true , numIncorrect : 0 , MaxNumIncorrect : 5 , Link : "a" >
collaboration(verifyPIN("a" , "b") - reenterPIN("b" , "a") - abort("a" , "b") - PINverified("b" , "a")  ) .
endom)
*** commands to verify the first collaboration
***  we use two commans to verify the collaboration
(rew initState-coll1 .)
(search [1] initState-coll1 =>! ( C:Configuration  collaboration(null) ) . )
(rew initState-coll2 .)
(search [1] initState-coll2 =>! ( C:Configuration  collaboration(null)) . )
```

Fig.71 : spécification Maude du module à vérifier

Nous utilisons la commande **search** pour vérifier si le système arrive à exécuter les deux collaborations. Les deux commandes sont introduites après le module global qui représente le système. Elles sont définies comme suit :

```
(search [1] initState-coll1 =>! ( C:Configuration  collaboration(null) ) . )
(search [1] initState-coll2 =>! ( C:Configuration  collaboration(null) ) . )
```

Nous pouvons utiliser également la commande **rew** comme suit :

```
(rew initState-coll1 .)
(rew initState-coll2 .)
```

L'exécution des commandes donnes les résultats présentés par la figure (Fig.67). Le résultat de la première commande ‘search’ donne une solution. Ce qui signifie que la collaboration est correcte et que les objets arrivent à interagir entre eux selon la spécification de cette collaboration. Ce résultat est renforcé aussi par le résultat de la commande (rew initState-coll1) qui donne un état final avec une liste vide (liste qui représente la collaboration), ce qui confirme que l’objet ATM et Bank arrivent à exécuter la collaboration de la figure (Fig.67)

Par contre, la deuxième commande search donne aucune (0) solution pour la deuxième collaboration. La commande rew confirme que les objets ATM et Bank ne peuvent pas exécuter cette collaboration. Ce qui signifier que cette collaboration (Fig.68) est erronée.

```
okba@localhost: /home/okba/maude-linux - Terminal - Konsole
Session  Édition  Affichage  Signets  Configuration  Aide
[okba@localhost maude-linux]$ maude atmbank.maude
                     \||||||||||||||||||/
                   --- Welcome to Maude ---
                     /||||||||||||||||||\
            Maude 2.3 built: Feb 14 2007 17:53:50
            Copyright 1997-2007 SRI International
                   Wed Feb 27 16:07:23 2008

       Full Maude 2.3 `(February 12th`, 2007`)

rewrites: 19723 in 60ms cpu (56ms real) (328716 rewrites/second)
Introduced module SMAtm

rewrites: 18145 in 40ms cpu (45ms real) (453625 rewrites/second)
Introduced module SMBank

rewrites: 10147 in 20ms cpu (24ms real) (507350 rewrites/second)
Introduced module MCHECK
rewrites: 994 in 10ms cpu (10ms real) (99400 rewrites/second)
rewrite in MCHECK :
  initState-coll1
result Configuration :
  collaboration(null)< "a" : ATM | BLOCKED : false,Link : "b",STATUS :
    ReturningCard > < "b" : BANK | BLOCKED : false,Link : "a",MaxNumIncorrect :
    2,STATUS : Idle,cardValid : true,numIncorrect : 0 >

rewrites: 1509 in 20ms cpu (18ms real) (75450 rewrites/second)
search [1] in MCHECK : initState-coll1 =>! C:Configuration collaboration(null).

Solution 1
C:Configuration --> < "a" : ATM | BLOCKED : false,Link : "b",STATUS :
    ReturningCard > < "b" : BANK | BLOCKED : false,Link : "a",MaxNumIncorrect :
    2,STATUS : Idle,cardValid : true,numIncorrect : 0 >
rewrites: 982 in 0ms cpu (5ms real) (~ rewrites/second)
rewrite in MCHECK :
  initState-coll2
result Configuration :
  collaboration(abort("a","b")- PINverified("b","a"))< "a" : ATM | BLOCKED :
    false,Link : "b",STATUS : PINEntry > < "b" : BANK | BLOCKED : false,Link :
    "a",MaxNumIncorrect : 2,STATUS : Idle,cardValid : true,numIncorrect : 1 >

rewrites: 1456 in 10ms cpu (6ms real) (145600 rewrites/second)
search [1] in MCHECK : initState-coll2 =>! C:Configuration collaboration(null).

No solution.

Maude>
Terminal
```

Fig.72: Résultat de vérification des collaborations

Nous avons présenté une nouvelle méthode de vérification des modèles UML en utilisant Maude et son LTL model-checker. Cette méthode se base sur la translation des modèles UML à des spécifications formelles en Maude. La méthode permet de vérifier des propriétés temporelles exprimées en logique temporelle linaire, tels que le deadlock, l'invariance, les reachbility, la réponse…etc. elle permet aussi de vérifier des collaborations, même des diagrammes de séquences, vis-à-vis le comportement de leurs objets.

Nous avons présenté les quatre étapes de cette méthode et puis

Et nous avons terminé le chapitre par deux exemples illustratifs. Il ne reste dans la partie suivante qu'a conclure le travail en donnant les perspectives de ce dernier.

Conclusion

Dans ce travail, nous avons proposé une méthode de vérification des diagrammes UML en utilisant la logique de réécriture et son langage Maude. La méthode repose sur la translation du modèle UML en une spécification formelle écrite en Maude dont le comportement du système modélisé est représenté par des règles de réécritures.

Les propriétés à vérifiées dans le système sont des propriétés exprimées en logique temporelle linéaire. Ces propriétés vont être vérifiées vis-à-vis la spécification Maude en utilisant le LTL model-checker du langage Maude.

Nous avons introduit dans le premier chapitre les concepts de base du génie logiciel. Puis nous avons décrit brièvement le langage UML. Nous avons terminé le chapitre par une présentation du processus unifie qui représente un cadre général de développement avec UML.

Nous avons consacré le deuxième chapitre au diagramme d'états-transition qui est utilisé par notre approche de model-checking d'UML. Ce diagramme spécifie le comportement internes des objets et leurs réactions aux stimuli provenant de l'extérieur. C’est le diagramme le plus visé par les travaux de formalisations du langage unifié. Nous avons commencé le chapitre par la description syntaxique et sémantique du diagramme d'états-transitions, et nous l'avons terminé par un survol sur les travaux de formalisation de ce dernier.

Nous nous somme intéressé dans le chapitre trois à la présentation de la logique de réécriture et son langage déclaratif Maude. Cette logique permet de spécifie formellement les systèmes concurrents. Nous avons donné les concepts de base de cette logique dans la deuxième partie du chapitre après la définition des méthodes formelles dans la première. La troisième partie était une description du langage Maude et son utilisation. Nous avons terminé le chapitre par les concepts du model-checking et du Maude LTL model-checker.

Dans le chapitre quatre, nous avons proposé une méthode à quatre étapes pour la vérification (model-checking) des modèles UML. Nous avons vu que la première étape s'agit d'élaborer un modèle UML en utilisant les diagrammes de classe, de collaboration et d'états-transitions. Nous avons proposé dans la deuxième étape une technique de translation des diagrammes, précédemment cités, à une spécification formelle en Maude. La définition des propriétés à vérifier consiste le vif de la troisième étape. Ces propriétés doivent êtres exprimés par l'utilisateur en logique temporelle linéaire. Elles sont vérifiées, dans la quatrième étape, par LTL model-checker vis-à-vis la spécification Maude/modèle UML. Nous avons présenté aussi au sein du chapitre quatre, des exemples illustrant l'utilisation de la méthode pour vérifier des modèles.

La méthode proposé dans ce travail gain beaucoup de puissance de fait qu'elle repose sur des concepts formelles de la logique de réécriture et du langage Maude. Les modèles UML translatés aux Maude vont gagner ainsi une sémantique précise et non ambiguë. Ces modèles sont vérifiés aussi par un Model-checker puissant, permettant au concepteur de détecter les erreurs dans le modèle UML qu'il élabore.

Malgré que la translation dans cette méthode s'effectuée manuellement et qu'elle nécessite un connaisseur du langage Maude. Nous tendances actuelles vers l'automatisation de cette méthode sont grandes. Nous proposons comme une perspective à ce travail la réalisation d'un outil qui décharge le concepteur la phase de translation. Cet outil, éventuellement, peut être développé en utilisant des grammaires graphiques "Graph Grammars".

Références

[1] L. Audibert, **UML 2.0**, Institut Universitaire de Technologie de Villetaneuse – Département Informatique, France, web : http ://www-lipn.univ-paris13.fr/audibert/pages/enseignement/cours.htm.

[2] M. O'Docherty, **Object-Oriented Analysis and Design: Understanding System Development with UML 2.0**, John Welly & sons, USA, 2005.

[3] M. Lai. **UML la notation unifié de modélisation objet, de java aux EJB**, Dunod, 2e édition 2000.

[4] P.A Muller, N. Geartner. **Modélisation objet avec UML**, Eyrolles, 2e édition 2000, Deuxième tirage 2001.

[5] P. Roques, F. Vallée, **UML en Action, de l'analyse des besoins à la conception en java**, Eyrolles 3e tirage 2001.

[6] H.E. Erikson, M.Penker, B. Lyons, D. Fado, **UML 2 ToolKit**, Welly publishing, USA, 2004.

[7] N. Lopez, J. Migueis, E. Pichon.**Intégrer UML dans vos projets**, Paris, 2000.

[8] M. Fowler, **UML Distilled third edition: A brief guide to the standard object modeling language** , Addison-Wesley, USA, 2003

[9] D. Pilone, N. Pitman, **UML 2.0 in a Nutshell**, O'rielly, 2005.

[10] I. Jacobson, G. Booch, J. Rumbaugh, **The Unified Process**, IEEE Software, vol. 16, no. 3, pp. 96-102, May/Jun, 1999.

[11] G. Booch, J. Rumbaugh, I. Jacobson, **the Unified Modeling Language User Guide SECOND EDITIO**, Addison-Wesley Professional, USA, 2005.

[12] UML Reference Manual

[13] Object Management group, **UML 2.1.1 specification**, OMG Document, 2008. web:// http://www.omg.org/technology/documents/formal/uml.htm

[14] M. L. Crane, J. Dingel. **On the Semantics of UML State Machines: Categorization and Comparison**, Technical Report 2005-501, School of Computing, Queen's University, Canada, 2005

Références

[15] G.Alknsis, ***The Analysis of UML State Machine Formal Checking Methods***, Deprtement of Applied Computer Science, Riga Technical University. 2007, http://ceur-ws.org/Vol-252/paper15.pdf

[16] D. Varró. ***A formal semantics of UML Statecharts by model transition systems***. In Proceeding of the 1st International Conference on Graph Transformation (ICGT 2002), volume 2505 of Lecture Notes in Computer Science, pages 378-392. Springer, 2002.

[17] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. ***Understanding UML: A formal semantics of concurrency and communication in real-time UML***. In Formal Methods for Components and Objects, volume 2852 of Lecture Notes in Computer Science, pages71-98. Springer, 2003.

[18] R. Eshuis and R. Wieringa. ***Requirements-level semantics for UML statecharts***. In Proceedings of Formal Methods for Open Object-Based Distributed Systems MOODS, pages 121-145. Kluwer Academic Publishers, 2000.

[19] S. Gnesi, D. Latella, and M. Massink. ***Modular semantics for a UML statecharts diagrams kernel and its extension to multicharts and branching time model-checking***. The Journal of Logic and Algebraic Programming, 51:43-75, 2002.

[20] D. Latella, I. Majzik, and M. Massink. ***Automatic verification of UML statechart diagrams using the SPIN model-checker***. Formal Aspects of Computing, 11(6):637-664, 1999.

[21] D. Latella, I. Majzik, and M. Massink. ***Towards a formal operational semantics of UML statechart diagrams.*** In Proceedings of the 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), pages 331-347. Kluwer, 1999.

[22] J. Jürjens. ***A UML statecharts semantics with message-passing***. In Proceedings of the 2002 ACM Symposium on Applied Computing (SAC'02), pages 1009-1013. ACM Press, 2002.

[23] E. Börger, A. Cavarra, and E. Riccobene. ***Modeling the dynamics of UML state machines***. In Proceedings of the International Workshop on Abstract State Machines, Theory and Applications, volume 1912 of Lecture Notes in Computer Science, pages 223-241.Springer, 2000.

[24] K. Compton, J.K. Huggins, and W. Shen. ***A semantic model for the state machine in the Unified Modeling Language***. In Dynamic Behaviour in UML Models: Semantic Questions, Workshop Proceedings, UML 2000 Workshop. Ludwig-Maximilians-universität München, Institut für Informatik, 2000.

[25] W. Shen, K. Compton, and J. K. Huggins. A ***toolset for supporting UML static and dynamic model checking***. In Proceedings of the 26th International Computer Software and Applications Conference (COMPSAC 2002), pages 147-152. IEEE Computer Society, 2002.

[26] Y. Jin, R. Esser, and J.W. Janneck. ***A method for describing the syntax and semantics of UML statecharts***. Software and Systems Modeling, 3(2):150-163, 2004.

[27] R. Esser and J.W. Janneck. ***Moses: A tool suite for visual modeling of discrete-event systems***. In Symposia on Human-Centric Computing, pages 272-279. IEEE Computer Society, 2001.

[28] J.W. Janneck and P.W. Kutter. ***Mapping automata - simple abstract state machines***. Technical Report 49, Computer Engineering and Networks Laboratory, ETH Zurich, 1998.

[29] S. Bernardi, S. Donatelli, and J. Merseguer. ***From UML sequence diagrams and statecharts to analysable Petri net models***. In Proceedings of the 3rd International Workshop on Software and Performance (WOSP'02), pages 35-45. ACM Press, 2002.

[30] J. Merseguer, J. Campos, S. Bernardi, and S. Donatelli. ***A compostional semantics for UML state machines aimed at performance evaluation***. In Proceedings of the 6th International Workshop on Discrete Event Systems, pages 295-302. IEEE Computer Society Press, 2002.

[31] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. ***GreatSPN-1.7 – graphical editor and analyzer for timed and stochastic Petri nets***. Performance Evaluation, 24(1-2):47-68, 1995.

[32] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. ***PROD reference manual***. Technical Report Series B, Number 13, Helsinki University of Technology, 1995.

[33] S. Donatelli and L. Ferro. ***Validation of GSPN and SWN models through the PROD tool***. In Proceedings of the 12th International Conference on Modeling Techniques and Tools, volume 2324 of Lecture Notes in Computer Science. Springer, 2002.

[34] L. Baresi and M. Pezzé. ***On formalizing UML with high-level Petri nets. In Proceedings of Concurrent Object-Oriented Programming and Petri Nets***, volume 2001 of Lecture Notes in Computer Science, pages 271-300. Springer, 2001.

[35] S. Christensen, J.B. Joergensen, and L.M. Kristensen. ***Design/CPN - a computer tool for colored Petri nets***. Lecture Notes in Computer Science, 1217, 1997.

[36] ARTIS s.r.l**. Artifex 3.1** - tutorial, 1994. Torino, Italy.

[37] M. Pezzé. Cabernet: **A customizable environment for the specifcation ande analysis of real-time systems.** Technical report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy, 1994.

[38] P. King and R. Pooley. **Using UML to derive stochastic Petri net models**. In Proceedings of the 15th UK Performance Engineering Workshop (UKPEW'99), pages 45-56. Department of Computer Science, The University of Bristol, 1999.

[39] R. Breu, R. Grosu, F. Huber, B. Rumpe, and W. Schwerin. ***Systems, views and models of UML***. In The Unified Modeling Language, Technical Aspects and Applications, pages 93-109. Physica Verlag, 1998.

[40] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. ***Towards a formalization of the Unified Modeling Language***. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), volume 1241 of Lecture Notes in Computer Science, pages 344-366. Springer, 1997.

[41] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, and R. Weber. ***The design of distributed systems - an introduction to FOCUS***. Technical Report SFB 342/2/92 A, Technische Universität München, 1993.

http://www4.informatik.tu-muenchen.de/reports/TUM-I9202.ps.gz.

[42] H. Fecher, M. Kyas, and J. Schönborn. ***Semantic issues in UML 2.0 state machines***. Technical Report 0507, Christian-Albrechts-Universität zu Kiel, 2005.

[43] S. Meng, Z. Naixiao, and L.S. Barbosa. ***On semantics and refinement of UML statecharts: a coalgebraic view***. In Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), pages 164-173. IEEE Computer Society, 2004.

[44] J. Rutten. **Universal co-algebra: a theory of systems**. Theoretical Computer Science, 249:3-80, 2000.

[45] G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. **Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML.** In Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 2000), volume 1939 of Lecture Notes in Computer Science, pages 323-337. Springer, 2000.

[46] M. Gogolla and F. Parisi-Presicce. **State diagrams in UML: A formal semantics using graph transformations**. In Proceedings of the Workshop on Precise Semantics for Modelling Techniques (PSMT'98), pages 55{72. Technische Universität München, TUMI9803, 1998.

[47] S. Kuske. **A formal semantics of UML state machines based on structured graph transformation**. In Proceedings of the 4th International Conference on the Unifed Modeling Language (UML 2001), volume 2185 of Lecture Notes in Computer Science, pages 241-256. Springer, 2001.

[48] D. Varro, G. Varro, and A. Pataricza. **Designing the automatic transformation of visual languages**. Science of Computer Programming, 44(2):205-227, 2002.

[49] D. Varro. **Towards symbolic analysis of visual modelling languages**. In Proceedings of the International Workshop on Graph Transformation and Visual Modelling Techniques, volume 72 of Electronic Notes in Theoretical Computer Science, pages 57-70. Elsevier, 2002.

[50] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueb, J. Rushby, V. Rusu,H. Saîdi, N. Shankar, E. Singerman, and A. Tiwari. **An overview of SAL**. In Proceedings of the 5th NASA Langley Formal Methods Workshop, pages 187-196, 2000

[51] G. Kwon. **Rewrite rules and operational semantics for model checking UML statecharts**. In Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 2000), volume 1939 of Lecture Notes in Computer Science, pages 528-540. Springer, 2000.

[52] K. McMillan. **Symbolic Model Checking: An Approach to the State Explosion Problem**. Kluwer Academic, 1993.

[53] D.B. Aredo. **Semantics of UML statecharts in PVS**. Research Report 299, Department of Informatics, University of Oslo, 2000.

[54] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. **Analysing UML active classes and associated state machines - a lightweight formal approach**. In Proceedings of Fundamental Approaches to Software Engineering (FASE 2000), volume 1783 of Lecture Notes in Computer Science, pages 127-146. Springer, 2000.

[55] P.D. Mosses. **CoFI: The common framework initiative for algebraic specification and development**. In Proceedings of TAPSOFT '97, volume 1214 of Lecture Notes in Computer Science. Springer, 1997.

[56] G. Reggio. **Metamodelling behavioural aspects: the case of the UML state machines** (complete version). Technical report, DISI - Universitµa di Genova, Italy, 2002.

[57] G. Reggio. **An "extreme" approach to metamodelling**. Technical report, DISI, Universita di Genova, Italy, 2002.

[58] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. **A CASL formal definition of UML active classes and associated state machines**. Technical Report DISI-TR-99-16, DISI-Universitµa di Genova, Italy, 1999.

[59] X. Zhan and H. Miao. **An approach to formalizing the semantics of UML statecharts**. In Proceeding of the 23rd International Conference on Conceptual Modeling (ER 2004), volume 3288 of Lecture Notes in Computer Science, pages 753-765. Springer, 2004.

[60] X. Zhan, H. Miao, and L. Liu. **Formalizing the semantics of UML statecharts with Z**. In Proceedings of the 4th International Conference on Computer and Information Technology (CIT '04), pages 1116-1121. IEEE Computer Society, 2004

[61] T. Schäfer, A. Knapp, and S. Merz. **Model checking UML state machines and collaborations**. In Electronic Notes in Theoretical Computer Science, volume 55. Elsevier, 2001.

[62] A. Knapp and S. Merz. **Model checking and code generation for UML state machines and collaborations**. In Proceeding of the 5th Workshop on Tools for System Design and Verification (FM-TOOLS 2002), Report 2002-11. Institut für Informatik, Universitüt Augsburg, 2002.

[63] G. Holzmann. **The model checker SPIN**. IEEE Transactions on Software Engineering, 23(5):279-295, 1997

[64] K.G. Larsen, P. Pettersson, and W. Yi. **UPPAAL in a nutshell**. International Journal on Software Tools for Technology Transfer, 1(1-2):134-152, 1997.

[65] J. Lilius and I. Porres Paltor. **Formalising UML state machines for model checking**. In Proceedings of The Unified Modeling Languge (UML'99), volume 1723 of Lecture Notes in Computer Science, pages 430-445. Springer, 1999.

[66] J. Lilius and I. Porres Paltor. **vUML: A tool for verifying UML models**. In Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99), pages 255-258. IEEE Computer Society, 1999.

[67] P. Gagnon, F. Mokhati, M. Badri: **Applying Model Checking to Concurrent UML Models**, in Journal of Object Technology, vol. 7, no. 1, January- February 2008, pp. 59-84, http://www.jot.fm/issues/issue_2008_01/article1/

[68] S. Jansamak and A. Surarerks. **Formalization of UML statechart models using concurrent regular expressions.** In Proceedings of the 27th Australasian Computer Science Conference (ACSC 2004), volume 26 of CRPIT, pages 83{88. Australian Computer Society, 2004.

[69] K. Lano, J. Bicarregui, and A. Evans. **Structured axiomatic semantics for UML models**. In Rigorous Object-Oriented Methods (ROOM 2000). Electronic Workshops in Computing (eWiC), 2000

[70] T. Aoki, T. Tateishi, and T. Katayama. **An axiomatic formalization of UML models.** In Practical UML-Based Rigorous Development Methods - Countering or integrating the eXtremists. Workshop of the pUML-Group held together with UML 2001, volume P-7 of LNI, pages 13-28. German Informatics Society, 2001.

[71] V.K. Garg and M.T. Ragunath. C**oncurrent regular expressions and their relationship to Petri nets**. Theoretical Computer Science, 96:285-304, 1992.

[72] P. C. Ölveczky. **Formal Modeling and Analysis of Distributed Systems in Maude**. Lecture Notes INF3230/INF4230, Department of Informatics, UNIVERSITY OF OSLO, 16 janvier 2008.

[73] E.M Clarke, J. M. Wing et al. **Formal methods : State of the art and futur directions**, in Strategic Directions in Computing Research Workshop, Cambridge, MA , ETATS-UNIS (06/1996) 1996, vol. 28, no 4 (290 p.) (4 p.3/4), pp. 626-643

[74] J. Meseguer. **Conditional rewriting logic as a unified model of concurrency**. Theoretical Computer Science, 96:73–155, 1992.

[75] R. Bruni and J. Meseguer. **Semantic foundations for generalized rewrite theories**. Theoretical Computer Science, 360(1-3):386–414, 2006.

[76] J. Meseguer. **Rewriting logic as a semantic framework for concurrency**: a progress report. en U. Montanari and V. Sassone, editors, Concur'96, volume 1119 of Lecture Notes in Computer Science, pages 331–372. Springer, 1996.

[77] N. Martí-Oliet and J. Meseguer. **Rewriting logic: Roadmap and bibliography.** Theoretical Computer Science, 285, 2002.

[78] K. Megzari, **REFINER : Environnement logiciel pour le raffinement d'architectures logicielles fondé sur une logique de réécriture**, thèse de doctorat, université de Savoie, France, 2004.

[79] N. Martí-Oliet and J. Meseguer. **Rewriting logic as a logical and semantic framework.** En first international workshop on rewriting logique and its application, volume4 of Electronic Notes in Theoritical Computer science, Elsevier, 1996.

[80] J.Goguen et al, **Introducing obj3**, SRI-CSL-88-9, SRI International, USA, 1988.

[81] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. **All About Maude - A High-Performance Logical Framework**, volume 4350 of Lecture Notes in Computer Science. Springer, 2007.

[82] M.Clavel F. Duran, S. Eker, P. Lincoln, N. Marti'-Oliet, J. Meseguer, C.Talcott. **MAUDE MANUAL (Version 2.3),** SRI Inter, 2007, web http://maude.cs.uiuc.edu/maude1/manual/

[83] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti'-Oliet, J. Meseguer, J.F. Quesada: **Maude: specification and programming in rewriting logic**. SRI International (January 1999), http://maude.cs.uiuc.edu/maude1/manual/

[84] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti'-Oliet, J. Meseguer, J.F. Quesada : **A tutorial on Maude**. SRI International (March 2000), http://maude.cs.uiuc.edu/maude1/tutorial/

[85] S. Eker, J. Meseguer, and A. Sridharanarayanan. **The Maude LTL model checker**. dans F. Gadducci and U. Montanari, editors, Fourth International Workshop on Rewriting Logic and its Applications, volume 71 of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[85] S. Eker, J. Meseguer, and A. Sridharanarayanan. **The Maude LTL model checker and its Implementation**, en model checking software, volume 2648/2003 of lectures Notes in computer science. Springer 2003.

[86] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. **Metalevel computation in Maude**. en C. Kirchner and H. Kirchner, editors, Second International Workshop on Rewriting Logic and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science. Elsevier, 1998.

[87] G. Denker, J. Meseguer, and C. Talcott. **Protocol Specification and Analysis in Maude**. In N. Heintze and J. Wing, editors, Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana, 1998.

[88] P. C. Ölveczky, J. Meseguer, and C. L. Talcott. **Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude**. Formal Methods in System Design, 2006.

[89] P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. **An object-oriented algebraic steamboiler control specification**. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, Formal Methods for Industrial Application: Specifying and Programming the Steam-Boiler Control, volume 1165 of Lecture Notes in Computer Science, pages 379–402. Springer, 1996.

[90] P. C. Ölveczky and S. Thorvaldsen. **Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude.** In 20th International Parallel and Distributed Processing Symposium (IPDPS 2006). IEEE Computer Society Press, 2006.

[91] F. Durán and J. Meseguer. **The Maude specification of Full Maude**. Manuscript, May 1999. http://maude.cs.uiuc.edu/papers/.

[92] E. M Clarke, O. Grumberg, D. A. Peled, **Model Checking,** MIT Press, 1999.

[93] E. M Clarke, B. H. Schlingloff, **Handbook of Automated reasoning**, Elsevier Science Publishers, USA, 2001.

[97] S. Merz, **Model Checking: A Tutorial Overview**, in Lecture Notes in Computer Science 2067, pp 3-38, 2001.