Table des matières

Table des matières 1
Index des figures 5
Index des tableaux 6
Index des définitions 7

Chapitre I. Introduction 9
1. Développer des systèmes temps réel fiables 9
2. L'émergence d'un consensus : UML 10
3. Les réponses apportées par l'ingénierie des exigences 10
4. Contributions apportées dans ce mémoire 11
5. Organisation du mémoire 12

Chapitre II. Contexte et positionnement des travaux 14
1. Les systèmes temps réel (STR) 14
1.1. Exigences posées par les STR 14
1.2. Méthodologies de conception de STR 16
1.2.1. Avant UML 16
1.2.2. Les méthodologies orientées UML 17
1.2.3. Les méthodes formelles dans la conception et la validation des STR 19
2. Systèmes temps réel et distribués (STRD) 21
2.1. Les exigences temporelles liées à la Qualité de Service des STRD 21
2.2. Méthodologies de conception de STRD 22
3. TURTLE (Timed UML and RT-LOTOS Environment) 24
3.1. Un profil dédié à la vérification de systèmes temps-réel et distribués 24
3.1.1. La sémantique TURTLE : le langage RT-LOTOS 24
3.1.2. Les diagrammes UML TURTLE 25
3.1.3. Un profil outillé 29
3.1.4. Positionnement du profil TURTLE 30
3.2. Méthodologie présentée en [APV 06] 31
4. Conclusion 32

Chapitre III. Proposition d'une méthodologie 34
1. Vue d'ensemble de la méthodologie 34
1.1. Définition informelle 34
1.2. Langage de description d'une méthodologie 35
1.2.1. Principe 35
1.2.2. Définition formelle 36
1.2.3. Méta-modèle 37
2. Guide de lecture 38

3. Vérification formelle d'exigences temporelles de modèles en contexte UML/SysML 39
3.1. Traitement des exigences 39
3.1.1. Recueil des exigences 40
3.1.2. Hypothèses de modélisation 42
3.1.3. Spécification des exigences 42
3.2. Analyse 43
3.3. Conception 46
3.4. Vérification formelle des exigences 47
4. Spécialisation dans le domaine des protocoles 50
4.1. Les règles de traitement des exigences 50
4.1.1. Exigences liées au type de protocoles envisagés et à la qualité de service 50
4.1.2. Hypothèses de modélisation 51
4.1.3. Les règles liées à la phase d'analyse 53
4.2. Les règles liées à la phase de conception 56
5. Conclusion 63

Chapitre IV. Langage de description d'exigences non-fonctionnelles temporelles 65
1. Etat de l'art et positionnement de nos travaux 65
2. Définition d'un langage de description d'exigences temporelles 68
2.1. Formalisation d'exigences temporelles SysML 68
2.1.1. Exemple de diagramme d'exigences 68
2.1.2. Définitions formelles 69
2.1.3. Méta-modèle UML 70
2.2. Timing Requirement Description Diagram (TRDD) 71
2.2.1. Exemple 72
2.2.2. Définition formelle 72
2.2.3. Méta-modèles UML 74
2.2.4. Pouvoir d'expression des TRDD 75
3. Composition d'exigences 76
3.1. Les concepts de composition et satisfaction 77
3.2. Proposition d'extension des diagrammes d'exigences 77
3.3. Définition formelle 79
4. Conclusion 79

Chapitre V. Vérification formelle d'exigences temporelles sur le modèle de conception 81
1. Approches de vérification des exigences temporelles 81
1.1. La vérification par contrôle de modèles 82
1.2. Vérification guidée par les observateurs 83
1.2.1. Mise en œuvre d'observateurs dans le simulateur Véda 83
1.2.2. Observateurs de machines synchrones 84
1.2.3. Observateurs pour les systèmes auto-validés 84
1.2.4. Le langage GOAL 85
1.2.5. Les observateurs du profil UML OMEGA 85
1.2.6. Méthodes de surcharge 86
1.2.7. Diagrammes de contexte et d'observation 86

2. Génération d'observateurs TURTLE à partir d'exigences temporelles 87
2.1. Construction d'un observateur TURTLE 87
2.1.1. Définition d'un observateur TURTLE 87
2.1.2. Méta-modèle UML d'une Tclass observateur 88
2.2. Vue d'ensemble 89
2.3. Traduction d'un TRDD vers le diagramme d'activités de l'observateur 90
2.3.1. Tables de traductions 90
2.3.2. Méta-modèles UML du diagramme d'activités d'un observateur 91
2.3.3. Algorithmes de construction du comportement de l'observateur 93
2.3.4. Exemples 95
2.3.5. Algorithmes de construction du comportement complet de l'observateur 96
2.4. Intégration de l'observateur dans le modèle 97
2.4.1. Définition des points d'observations 97
2.4.2. Construction du modèle observable : principe 98
2.4.3. Méta-modèle UML du modèle couplé avec l'observateur 99
2.4.4. Algorithme d'insertion d'un observateur dans le système à observer 99
3. Discussion sur l'approche de vérification guidée par les observateurs 101
3.1. Hypothèses 101
3.2. Placement des points d'observations 102
3.3. Vers une implantation complète de l'approche 104
4. Conclusion 105

Chapitre VI. Application : le projet SAFECAST 106
1. Le projet SAFECAST 106
1.1. Réseau sans fil : le médium PMR 107
1.2. Revue de travaux sur la modélisation et la vérification de protocoles de sécurité 108
1.3. Pourquoi utiliser TURTLE dans le projet SAFECAST ? 108
2. Exigences, architecture et services considérés 109
2.1. Exigences temporelles d'un protocole de sécurité 110
2.2. Architecture générique 110
2.2.1. Services de diffusion sécurisée 111
2.2.2. Services de gestion des groupes 111
3. Etude du délai d'établissement d'une communication chiffrée 112
3.1. Traitement des exigences 112
3.1.1. Recueil des exigences 112
3.1.2. Hypothèses de modélisation 113
3.1.3. Spécification d'une exigence temporelle 113
3.2. Modèle de conception pour la fonctionnalité Génération et Distribution des clés (Key_Renewal) 114
3.3. Construction des observateurs 115
4. Contributions et résultats 117
5. Conclusion 118

Chapitre VII. Conclusions et perspectives 119
1. Bilan des contributions 119

2. Perspectives 121
2.1. Enrichir les descriptions d'exigences non-fonctionnelles temporelles 121
2.2. Extension de la méthodologie dans les phases de déploiement et de codage 122
2.3. Perspectives liées au projet SAFECAST 123

Références 124

Annexes 131

Index des figures

Fig.1. Cycle de vie des systèmes selon la méthode UP……...18
Fig.2. Architecture à trois couches inspirée du modèle OSI……...22
Fig.3. Structure d'une Tclass……...26
Fig.4. Composition entre deux instances de classes……26
Fig.5. Exemple de diagramme global d'interactions TURTLE……..... 28
Fig.6. Exemple de diagramme de séquences…….. 29
Fig.7. La chaîne d'outils du profil TURTLE……...30
Fig.8. Méthodologie TURTLE pour la conception de systèmes temps réel……32
Fig.9. Langage de description de la méthodologie (DP)……36
Fig.10. Méta-modèle du DP ……37
Fig.11. DP de la phase de traitement des exigences……..40
Fig.12. Pattern du diagramme d'exigence incluant une exigence formelle…….. 43
Fig.13. DP de l'étape d'analyse……44
Fig.14. Placement des points d'observation dans un diagramme de séquences TURTLE……... 45
Fig.15. DP de l'étape de conception…… 46
Fig.16. Placement des points d'observations dans la classe observée…….. 47
Fig.17. DP de l'étape de vérification…… 48
Fig.18. Matrice de traçabilité de TURTLE pour la vérification d'exigences non-fonctionnelles… 50
Fig.19. Diagrammes de cas d'utilisation générique pour un protocole en mode connecté et non-connecté…… 54
Fig.20. Diagramme de séquences générique pour un protocole……... 54
Fig.21. Placement des points d'observation dans un diagramme de séquence TURTLE……55
Fig.22. Placement des points d'observation dans un DS pour vérifier une exigence de QoS de « jitter »……55
Fig.23. Placement des points d'observation dans un DS pour vérifier une exigence de QoS de « end to end delay »……..... 56
Fig.24. Description de l'abstraction des couches supérieures…….....57
Fig.25. Description du modèle du service sous-jacent……..57
Fig.26. Pattern de diagramme de classes d'un protocole à vérifier……...59
Fig.27. Placement des points d'observations dans la classe d'appel de fonctionnalité Service_A...60
Fig.28. Placement des points d'observations pour vérifier une exigence de QoS de « jitter »…….60
Fig.29. Placement des points d'observations dans les classes applications pour vérifier une exigence de QoS de « end to end delay »…… 61
Fig.30. Exemple pour introduire les modules d'Aléas……..62
Fig.31. Diagrammes d'activités et graphes d'accessibilité correspondant aux Aléas liés à un service non-fiable……..63
Fig.32. Récapitulatif de la méthodologie et présentation des contributions…… 64
Fig.33. Diagramme d'exigences TURTLE……69
Fig.34. Méta-modèle du diagramme d'exigences TURTLE……71

Fig.35. Diagramme de description d'exigence temporelle de P_TRDD…… 72
Fig.36. Grammaire pour la construction de description d'exigences temporelles…… 73
Fig.37. Méta-modèle du TRDD……..75
Fig.38. Méta-modèle décrivant les règles de construction du TRDD…… 75
Fig.39. Exemples de TRDD……76
Fig.40. Proposition d'extension des diagrammes d'exigences TURTLE……...78
Fig.41. Méta-modèle d'une classe stéréotypé « TObserver »…… 89
Fig.42. Méta-modèle du diagramme d'activités d'un observateur…… 92
Fig.43. Méta-modèle des règles de construction du diagramme d'activités d'un observateur…….. 93
Fig.44. Application des algorithmes 2, 3 et 4 aux exemples de la Fig.39 (chapitre IV page 68)….. 96
Fig.45. Génération des observateurs…….. 98
Fig.46. Méta-modèle du package TIF incluant des observateurs……..... 99
Fig.47. Caractérisation du placement des points d'observations……..103
Fig.48. Vers l'implantation de nos travaux……...105
Fig.49. Architecture de modélisation de communications de groupes sécurisées……111
Fig.50. Diagramme d'exigence EC_Duration…….. 114
Fig.51. Diagramme de classes simplifié du protocole de gestion de clés……115
Fig.52. Diagramme d'activités d'une entité du protocole de gestion de clés et de l'observateur….116
Fig.53. Récapitulatif de la méthodologie et présentation des contributions……120
Fig.54. Caractérisation du dispositif d'observation pour le déploiement……122

Index des tableaux

Tab.1. Taxonomie des exigences non-fonctionnelles temporelles inhérentes aux systèmes temps réel……15
Tab.2. Sémantique des éléments du diagramme d'activité TURTLE……... 27
Tab.3. Guide de lecture du chapitre III……. 38
Tab.4. Base de données répertoriant les exigences à traiter dans la phase de recueil d'exigences……41
Tab.5. Base de données répertoriant les exigences enrichies dans la phase de spécification d'exigences……42
Tab.6. Langages visuels basés sur la description de scenarii…… 66
Tab.7. Langages visuels basés sur les chronogrammes……67
Tab.8. Sémantique des éléments du TRDD……...73
Tab.9. Table de traduction des descriptions d'exigences temporelles TURTLE en diagramme d'activités TURTLE……...…… 91
Tab.10. Table de traduction des descriptions d'exigences temporelles TURTLE en relation dans le diagramme de Classes TURTLE……98
Tab.11. Outils de vérification de protocoles de sécurité basés sur les systèmes de transitions…….108
Tab.12. Base de données des exigences liées à l'établissement de connexion et au renouvellement de clés……112

Tab.13. Base de données présentant l'exigence de délai de renouvellement de clés…... 113

Index des définitions

Définition 1 : Processus de développement……....36
Définition 2 : Diagramme d'exigences……...69
Définition 3 : Exigence informelle…… 69
Définition 4 : Exigence temporelle formelle…….. 70
Définition 5 : Observateur…….. 70
Définition 6 : TRDD……... 73
Définition 7 : Diagramme d'exigences étendu……... 79
Définition 8 : Observateur…….. 88

Chapitre I. Introduction

1. Développer des systèmes temps réel fiables

Le domaine du temps réel touche des applications, très variées dans la vie quotidienne, telles que le guidage ou la navigation (de bateaux, d'avions, de train ou d'automobile), le contrôle de processus industriels, la robotique, les télécommunications (téléphones portables, satellites), les transactions bancaires ou même les loisirs (jeux vidéo). Si certaines défaillances apparaissent dans ce type de systèmes, cela peut conduire à des pannes, des pertes de données ou de temps qui peuvent être catastrophiques pour certaines applications (avions, automobiles, bateaux, trains, centrales nucléaires). A cette criticité en vies humaines s'ajoute également une notion de criticité économique et le besoin de satisfaire le client.

Le problème de la fiabilité des systèmes temps réel est ainsi posé depuis plusieurs décennies. Les domaines des transports et des banques, pour n'en citer que quelques uns, ont déjà une longue tradition de la recherche de la fiabilité, eu égard à la criticité des systèmes qu'ils déploient. Ces domaines sont ceux où les méthodes de développement de systèmes sont les plus rigoureuses (pensons aux organismes de certification en aéronautique qui tendent aussi à être intégrés dans le domaine de l'automobile), faisant parfois appel aux *méthodes formelles* que le reste de l'industrie juge, avec une opinion préconçue, souvent inutilement compliquées et lourdes à mettre en œuvre.

L'intérêt de ces méthodes formelles n'a cessé de croître avec l'essor des systèmes distribués. Des défaillances locales qui n'avaient auparavant aucune conséquence, peuvent maintenant entrainer des pannes plus significatives de par la « connectivité » des systèmes déployés à grande échelle (Internet par exemple) et en interdépendances croissantes. L'on assiste donc à un intérêt grandissant pour les méthodes, langages et techniques permettant d'améliorer rapidement et efficacement la qualité de systèmes répondant aux attentes du client. L'utilité de ces méthodes formelles est avérée en matière de validation de systèmes a priori puisqu'une vérification formelle de modèle permet de confronter un modèle de conception aux exigences à valider pour détecter les erreurs de conception sans attendre la réalisation d'un prototype voire même du système effectif.

2. L'émergence d'un consensus : UML

L'*Unified Modeling Language* (UML) [UML] [BOO 03] est un langage pour documenter et spécifier graphiquement tous les aspects d'un système à logiciel prépondérant. L'ambition des promoteurs d'UML est de fédérer en une seule notation les meilleures caractéristiques des différents langages de modélisation qui utilisent le paradigme objet. En tant que standard de l'OMG (*Object Management Group*) [OMG], UML jouit d'une popularité sans précédent à la fois dans le monde industriel et académique. UML se veut être un médium, sous forme de modèle graphique, servant à harmoniser les différents acteurs concourant à la réalisation d'un système, et à garantir la cohérence et la qualité de la conception.

UML présente l'avantage de pouvoir se décliner sous forme de « profils » spécialisés en fonction des domaines d'application considérés (pour le temps réel par exemple [OMG 05] [OMG 07]). Ces profils permettent aussi de combler des lacunes d'UML en termes de sémantique pour servir de pivot à des outils de validation (dans notre cas la vérification formelle). Les outils dédiés aux profils « UML temps réel », c'est-à-dire aux personnalisations d'UML destinées à mieux prendre en compte le caractère réactif de ces systèmes mais aussi l'expression d'exigences et de mécanismes temporels, pêchent par le volet méthodologique. La manière d'utiliser le profil est insuffisamment et informellement documentée (manque de modèles génériques par exemple). De surcroit, l'objectif de *traçabilité des exigences* si souvent affiché n'est que partiellement atteint dans la mesure où certaines exigences doivent être décrites hors du modèle UML et reliées de manière ad-hoc et non automatisable à la validation de modèle.

Une qualité importante du langage UML est de se construire autour des fonctions que le système doit offrir (c'est-à-dire les *exigences fonctionnelles*) ; ce sont les *cas d'utilisations*. Ce concept permet d'avoir un vue opérationnelle du système en termes de mécanismes à concevoir, ce qui convient bien pour décrire des solutions logicielles (architectures, comportement). Mais UML est mal adapté pour décrire le problème à résoudre - l'objet même du cahier des charges - les *exigences non-fonctionnelles* (dans le domaine temporel, par exemple, le temps de sortie des trains d'atterrissage d'un avion). Or ces *exigences non-fonctionnelles* influencent toutes les définitions des mécanismes définis dans des *cas d'utilisations*.

3. Les réponses apportées par l'ingénierie des exigences

Pour palier le problème de l'expression des *exigences non-fonctionnelles*, l'ingénierie des exigences [LAM 06] [HUL 04] définit le concept d'objectif et permet de formuler plus précisément les *exigences*, *fonctionnelles* ou *non-fonctionnelles*. L'utilisation des techniques de modélisation orientées « objectifs » [LAM 06] permet d'élaborer le cahier des charges d'un système de manière rigoureuse. En effet, à partir d'un objectif donné, en se posant la question du « pourquoi », on peut identifier des objectifs de haut niveau à satisfaire ; en se posant la question du « comment », on peut identifier l'ensemble des mécanismes qui vont permettre d'atteindre les objectifs en question.

Dans ce contexte, l'OMG a récemment normalisé SysML [SysML] amenant avec lui, entre autres concepts, l'idée de pouvoir formuler des exigences au moyen d'un *diagramme d'exigences*. Ceux-ci concourent à assurer, à des fins documentaires, un suivi de ces exigences dans le cycle de développement du système par la construction d'une *matrice de traçabilité*. Notons néanmoins que

dans les premiers outils disposant de plug-ins SysML (par exemple, TAU G2.3.1 [TAU]), les exigences sont décrites de manière totalement informelle et sans lien automatisé avec les fonctionnalités de validation de modèles.

4. Contributions apportées dans ce mémoire

Les travaux présentés dans ce mémoire se concentrent sur les phases amont (traitement des exigences – analyse – conception) du cycle de développement de systèmes temps réel à logiciel prépondérant en donnant une place majeure à la vérification formelle des exigences temporelles.

Les principales contributions de ce mémoire visent donc à :

- **Définir un volet « méthodologie » d'utilisation de profils UML/SysML temps réel reposant sur des outils de vérification formelle qui distingue explicitement aspects fonctionnels et non-fonctionnels**. La méthodologie présentée dans ce mémoire intègre l'aspect formalisation et vérification pour assurer la traçabilité des exigences non-fonctionnelles temporelles qui sont occultées dans de très nombreuses méthodologies UML (par exemple [BOO 00] [ROQ 04] [DOU 04]). Avant de présenter la méthodologie en question, ce mémoire introduit un langage de description de processus méthodologique qui intègre la notion de *production/consommation* de biens livrables (en l'occurrence des diagrammes UML et SysML). Notons que la méthodologie proposée s'appuie sur des diagrammes UML et SysML (d'exigences, d'analyse de conception) génériques et réutilisables bien qu'elle ait pour finalité d'être instanciée dans le profil UML TURTLE [APV 04] [APV 06] (*Timed UML and RT-LOTOS Environment*).

- **Spécialiser la méthodologie dans le domaine des protocoles en prenant en compte les exigences liées au concept de *Qualité de Service* (QoS)**. Pour utiliser le profil TURTLE en conception de protocoles, ce mémoire établit un ensemble de règles de définition d'exigences, mais aussi de construction de diagrammes d'analyse et de conception liés au concept de *qualité de service* dans les protocoles temps réel (en terme de *gigue* et *délai de bout en bout*). Indépendantes de TURTLE, ces règles sont généralisables à d'autres profils UML.

- **Enrichir les capacités de modélisation du profil UML TURTLE pour la phase de *traitement des exigences***. Les *diagrammes d'exigences* de SysML [SysML] sont étendus pour séparer les exigences informelles des exigences formelles et en particulier des exigences non-fonctionnelles temporelles présentant une occurrence de début et de fin afin de mesurer leur distance temporelle. Ces dernières sont définies non pas par des formules de logique [HUT 04], mais par un langage visuel basé sur la représentation en chronogrammes et appelé *diagramme de description d'exigences temporelles* (*Timing Requirement Description Diagram ou TRDD*). Ces contributions sont présentées au travers de définitions formelles et de méta-modèles afin de les rendre réutilisable dans des profils UML/SysML autres que TURTLE.

- **Construire un dispositif automatisé permettant de tracer les exigences non-fonctionnelles par vérification guidée par observateurs**. Ce dispositif repose sur la génération automatique d'observateurs à partir d'une spécification des exigences temporelles par les diagrammes d'exigences étendus et les TRDD. Cette génération automatisable s'effectue en deux grandes étapes : la construction du comportement de l'observateur (à partir de la spécification de

l'exigence en TRDD) et la construction du modèle observable contenant le modèle du système couplé avec l'observateur. Ce modèle observable est dérivé du modèle TURTLE de conception. Ces contributions sont présentées sous formes d'algorithmes validés expérimentalement et en cours d'implantation dans l'outil TTool [TTOOL] qui supporte le profil TURTLE.

- **Caractériser l'insertion d'un observateur dans le modèle à observer**. Cet aspect présente un caractère fondamental de la *vérification guidée par observateurs*. Si ces derniers ne sont pas greffés correctement dans le modèle cela conduit à un mauvais diagnostic de l'observateur. Nous avons donc proposé des patterns de placement « correct » des observateurs caractérisant différentes exigences temporelles que l'on cherche à vérifier dans le modèle (mesurer la durée d'une fonctionnalité ou d'un service, mesurer la *gigue* au niveau de l'application ou le *délai de bout en bout* d'un protocole).

- **Instancier l'ensemble de nos travaux dans un projet industriel.** Le projet RNRT SAFECAST [SFC] dédié à la conception d'un protocole de communication de groupes sécurisée, a permis d'instancier sur un système distribué et temps réel complexe la méthodologie et le traitement d'exigences proposés dans ce mémoire. En complément à la vérification orientée sécurité décrite en [L 4.1], nous avons apporté à EADS (leader du projet) des preuves de satisfaction (ou non) d'exigences temporelles. L'on notera en particulier notre contribution aux livrables du projet ([L 2.5] [L 4.1] [L4.2] [L 4.3] [L 3.4] [L 4.4]).

5. Organisation du mémoire

Ce mémoire est structuré de la manière suivante.

Le chapitre II définit le contexte et les champs d'application de nos travaux. Nous introduisons, dans un premier temps, les exigences posées par les systèmes temps réel et dans un second temps les méthodologies associées à la conception de ce type de systèmes. Puis, nous spécialisons la discussion précédente vers les systèmes qui sont à la fois temps réel et distribués. Nous décrivons ensuite le profil UML temps réel TURTLE [APV 04] et les éléments de méthodologie qui avaient été proposés pour ce profil avant que ne démarrent les travaux présentés dans ce mémoire [APV 06]. Enfin, nous dressons un bilan général des différents travaux présentés dans ce chapitre en identifiant les points durs non-traités pour annoncer les chapitres suivants de ce mémoire.

Le chapitre III présente une proposition de méthodologie pour la vérification des exigences temporelles dans un système temps réel et en particulier les protocoles. Nous introduisons, tout d'abord, un langage de description de processus méthodologique utilisé dans la suite du chapitre. Ensuite, nous présentons une méthodologie de conception de systèmes temps réel en contexte UML/SysML, en se dégageant d'outils UML particuliers dans un premier temps puis en se focalisant sur le profil UML TURTLE. Enfin, cette méthodologie est spécialisée pour la conception de protocoles temps réel.

Le chapitre IV est consacré aux diagrammes de la phase de traitement des exigences dans le profil TURTLE. Après avoir présenté un état de l'art des travaux dédiés aux langages de description formelle d'exigences temporelles, nous introduisons les diagrammes utilisés en TURTLE pour spécifier les exigences non-fonctionnelles temporelles. Il s'agit d'une combinaison des diagrammes

d'exigences SysML et des diagrammes de description d'exigences temporelles (*Timing Requirement Description Diagram ou* TRDD en abrégé). Enfin, nous proposons des extensions aux diagrammes d'exigences TURTLE pour décrire l'aspect compositionnel des exigences temporelles en introduisant les notions de raffinement et de satisfaction d'exigences fonctionnelles.

Le chapitre V traite du volet « vérification d'exigences non-fonctionnelles temporelles » de TURTLE. Après avoir recensé les différentes méthodes de vérification d'exigences temporelles, nous présentons le processus de vérification d'exigences temporelles guidée par observateurs dans le profil TURTLE. Enfin, nous précisons les hypothèses de fonctionnement de ce processus avant de caractériser le placement des *points d'observations*, élément crucial dans la *vérification guidée par les observateurs*.

Le chapitre VI décrit l'instanciation de nos travaux dans un projet industriel, SAFECAST [SFC], qui soulève une problématique de communication de groupe sécurisée. Après avoir brièvement décrit les objectifs de ce projet, nous positionnons notre contribution à SAFECAST par rapport aux travaux du domaine. Ensuite, nous proposons des canevas d'exigences et d'architecture incluant les différents services du protocole SAFECAST. Puis nous présentons le modèle TURTLE pour la vérification formelle d'une exigence temporelle concernant la durée du service de génération et de distribution des clés, point dur de l'architecture SAFECAST.

Le chapitre VII conclut ce mémoire en dressant un bilan de nos contributions et en ouvrant des perspectives à notre travail.

Chapitre II. Contexte et positionnement des travaux

Le travail présenté dans ce mémoire est centré sur la vérification d'exigences temporelles et le développement d'une méthodologie qui intègre pleinement ce traitement. Aussi, ce chapitre dresse-t-il un état de l'art en matière d'exigences et de méthodologies proposées par d'autres auteurs pour les systèmes temps réel centralisés, puis distribués.

Ce chapitre est structuré comme suit. La section 1 recense dans un premier temps les exigences posées par les systèmes temps réel et dans un second temps les méthodologies associées à la conception de ce type de systèmes. La section 2 spécialise la discussion précédente vers les systèmes qui sont à la fois temps réel et distribués. La section 3 présente le profil UML temps réel TURTLE adossé au langage formel RT-LOTOS et expose les éléments de méthodologie publiés jusqu'ici pour ce profil, antérieurement aux (et indépendamment des) contributions présentées dans ce mémoire. Enfin, la section 4 dresse un bilan général des différents travaux présentés dans ce chapitre et identifie les points durs non-traités,.

1. Les systèmes temps réel (STR)

1.1. Exigences posées par les STR

[HUL 04] distingue deux classes d'exigences dans un système :

- Les *exigences fonctionnelles* représentent les différentes fonctionnalités que devra satisfaire le système et correspondent à l'aspect opérationnel de ce système. En matière de vérification formelle, on associe généralement ce type d'exigences à des « propriétés générales » telles que l'absence de blocage non désiré (« deadlock »), l'absence de fonctionnement cyclique infini non désiré (« livelock ») ou encore la possibilité de revenir à l'état initial.
- Les *exigences non-fonctionnelles* correspondent aux critères de qualité attendus du système, par exemple en termes de performances temporelles. A titre d'exemple, pensons aux différentes garanties temporelles (par exemple des délais maximum ou minimum) que doit satisfaire un système temps réel.

Dans le domaine du temps réel, les *exigences non-fonctionnelles temporelles* se répartissent plus précisément en deux catégories [WAH 94]:

- Les exigences où le temps est exprimé de manière qualitative (ou *logique*). On ne considère alors qu'un ordre partiel entre événements (par exemple un ascenseur doit être à l'arrêt pour que la porte s'ouvre).
- Les exigences où le temps est exprimé de manière quantitative. On considère dans ce cas l'ordre des événements mais aussi les distances temporelles entre ces derniers (par exemple la porte d'un ascenseur doit s'ouvrir deux secondes après l'arrêt de la cabine).

Dans le domaine de la vérification formelle d'exigences temporelles quantitatives, il existe deux manières de représenter le temps [ALU 94] :

- En l'exprimant de manière *discrète* en considérant le temps comme une horloge préalablement définie. Le modèle temporel correspondant est donc basé sur une horloge discrète. Cette expression du temps peut s'avérer être un facteur limitant : par exemple, si un système a des constantes de temps variant de 1 à 10000, considérer une horloge discrète de granularité 1 engendre un nombre d'états prohibitif dans le modèle. D'autre part, l'utilisation du temps discret, par la construction d'une horloge discrète, suppose la connaissance exacte de la durée totale d'exécution du système, c'est-à-dire des durées des diverses opérations à réaliser.
- En l'exprimant de manière *continue* en considérant le temps comme une succession d'horloges et de frontières temporelles. Pour cela on construit un modèle plus fin où l'on peut utiliser différentes horloges pour spécifier le comportement temporel du système. Outre la meilleure prise en compte des différentes granularités temporelles du système, cette représentation du temps permet aussi de prendre en compte une *incertitude sur les contraintes temporelles* du système (durée des tâches, des communications).

Ce mémoire se focalise sur les *exigences non-fonctionnelles temporelles* quantitatives où le temps est représenté de manière *continue*. Ces dernières correspondent à des propriétés de *vivacité*[1] *bornée* [ALU 93] issues de la classe des propriétés de *sûreté*[2] (« *safety* ») liées à la correction partielle du système. Les propriétés de *vivacité bornée* se voient affecter une date butoir pour laquelle la propriété n'est plus satisfaite [ALU 93] ; ceci implique donc que le système doit être borné temporellement pour pouvoir être vérifié.

Le Tab.1 présente les exigences non-fonctionnelles correspondant à des propriétés de *vivacité bornée* définies dans [ALU 93]. On note la relation de satisfaction d'une exigence E sur un système S par $S \models E$.

Nom de la classe d'exigence	Définition
Délai maximum (*Promptness*)	E assure qu'un événement devra obligatoirement se produire avant un certain délai D_{max}. $S \models E$ est vrai si cet événement s'est produit avec un délai strictement inférieur au délai maximum D_{max}.
Délai minimum (*Minimal Delay*)	E assure qu'un événement ne devra jamais se produire avant un certain délai D_{min}. $S \models E$ est vrai si cet événement s'est produit avec un délai strictement supérieur au délai minimum D_{min} et inférieur à la date butoir où l'exigence n'est plus satisfaite (vivacité bornée).
Ponctualité (*Punctuality*)	E assure qu'un événement devra obligatoirement se produire à un instant T. $S \models E$ est vrai si cet événement se produit à cet instant T.

[1] Une propriété de vivacité assure qu'un événement sous certaines conditions devra fatalement se produire.
[2] Une propriété de sûreté assure qu'un événement sous certaines conditions ne doit jamais se produire.

Périodicité (*Periodicity*)	E assure qu'un événement devra obligatoirement se produire régulièrement à un instant T. S ⊨ E est vrai si cet événement se produit aux instants modulo T.
Délai borné (*Interval Delay*)	E assure qu'un événement devra obligatoirement se produire entre/hors un intervalle temporel $]T_{min} ; T_{max}[$. S ⊨ E est vrai si cet événement se produit entre/hors cet intervalle temporel $]T_{min} ; T_{max}[$.

Tab.1. Taxonomie des exigences non-fonctionnelles temporelles inhérentes aux systèmes temps réel

Le choix d'exprimer les exigences dans les bornes strictes (inférieures et/ou supérieures) n'est pas anodin ; il correspond au non-déterminisme temporel de la sémantique du langage TURTLE (cf. chapitre V section 3.1) sur lequel s'instancie la méthodologie présentée au chapitre III de ce mémoire.

1.2. Méthodologies de conception de STR

Cette section recense des méthodologies de conception de systèmes temps réel. La section 1.2.1 présente brièvement les méthodologies antérieures à la normalisation d'UML. La section 1.2.2 dresse un panorama des méthodologies dédiées au langage UML. Enfin, la section 1.2.3 recense des méthodologies reposant sur les méthodes formelles.

1.2.1. Avant UML

La méthode MERISE [TAR 83] orientée « systèmes d'informations », a rencontré un succès certain dans les années 80. La conception du système d'information repose sur un *cycle d'abstraction* composé d'étapes afin d'aboutir à un système d'information fonctionnel proche de la réalité. La méthode MERISE préconise d'analyser séparément données et traitements dans chaque étapes. Le passage à une autre étape correspond à un point de validation et de décision. Il s'agit donc de valider une à une chacune des étapes en prenant en compte les résultats de l'étape précédente. Une telle démarche est beaucoup trop rigide à cause des nombreux points de validation nécessitant de remonter au système général à la fin de chaque *cycle d'abstraction*. Enfin, si nous mentionnons MERISE de par sa large diffusion et les comparaisons fréquentes avec les méthodologies basées UML, il faut rappeler que cette méthode n'a pas du tout été conçue pour les STR.

Moins rigide que MERISE du point de vue validation, SA-RT (*Structured Analysis - Real Time*) [HAT 87] se répandit vers le début des années 1990 comme l'un des standards de description systèmes temps réel. SA-RT est une méthode graphique de description d'un système temps réel complexe par *analyse fonctionnelle descendante* [HAT 87]. L'analyse chemine du général vers le particulier et le détaillé contrairement à MERISE qui opère des retours successifs au système général. SA-RT s'appuie sur différentes descriptions du système à concevoir : vue statique présentant les transformations de données et vue dynamique présentant les réactions du système face à des stimuli de son environnement. Le temps est représenté dans la vue dynamique de manière logique par des machines à états finis (ce formalisme permettant de décrire la réaction du système face à des stimuli extérieurs au système).

Ni MERISE, ni SA-RT ne prennent en compte le fait qu'un système ou une partie d'un système peut être vue comme un ensemble de modules (notion de *modularité*) éventuellement réutilisables (notion de *portabilité*). D'où l'introduction de méthodes de conception orientée objet qui

intègrent les concepts de *modularité* et *portabilité*. Ces méthodes basées sur des langages de modélisation orienté objet (comme OMT[3] [RUM 91]) ont été utilisées dans l'industrie pour la conception des systèmes temps réel avant la normalisation d'UML. Cependant ces méthodes ne prennent pas en compte explicitement les besoins du client et le fait que ce dernier peut changer d'avis pendant un projet.

En introduisant le concept de développement adaptatif, les méthodes de types AGILE [AGIL] (dont la plus utilisée est la méthode *eXtrem Programming* [KEN 00]) présentent de très bonnes solutions de rapidité dans le développement et de prise en compte de modifications demandées par les clients durant le cycle de développement. L'aspect itératif et incrémental présenté dans les méthodes AGILE est donc pris en compte dans la méthodologie présentée au chapitre III de ce mémoire. Impliquer le client dans le projet favorise la conception d'un système qui réponde à ses attentes. Il faut donc construire un modèle du système autour des *cas d'utilisation* qui correspondent aux besoins et exigences des utilisateurs. L'on parle alors de « bonne pratique » (notion de *best practices*) dans le développement de système.

1.2.2. Les méthodologies orientées UML

En intégrant des modèles orientés autour des *cas d'utilisation* et les notions de *modularité* et *portabilité*, les méthodes de type UP (*Unified Process)* [BOO 00] s'imposent très souvent dans des grands projets (méthode *Rational Unified Process* ou RUP). Elles sont construites autour du langage semi-formel UML [UML]. Un processus unifié (UP) est *« itératif et incrémental, centré sur l'architecture, construit à partir des cas d'utilisations et piloté par les risques »* [BOO 00]. Un processus unifié doit être vu comme une trame commune de meilleures pratiques de développement, et non comme une tentative d'élaborer un processus universel.

La Fig.1, montre le cycle itératif d'une méthode UP, découpé en 5 activités de développement :

- Le *recueil des besoins*. Cette activité permet de définir les différents besoins (ou exigences) et de faire la distinction entre besoins *fonctionnels* qui conduisent à l'élaboration des modèles de *cas d'utilisations* et les besoins *non-fonctionnels* qui forment une liste d'exigences à satisfaire. Le modèle de cas d'utilisation présente le système du point de vue de l'utilisateur sous forme de cas d'utilisation et d'acteur. Il est représenté par un *diagramme de cas d'utilisations* UML.
- L'*analyse*. L'objectif de cette activité est d'accéder à une compréhension des besoins et des exigences. Ceci correspond à la livraison de spécifications du modèle d'analyse. Ce dernier livre une spécification complète des exigences fonctionnelles (*les cas d'utilisations* appelées très souvent *besoins*) et les structure sous forme de scénarii pour faciliter la compréhension. Le modèle correspondant est décrit en UML par des *diagrammes de séquences* et des *diagrammes globaux d'interaction* ; il permet d'établir des passerelles avec le modèle de conception.
- La *conception*. Cette activité permet d'acquérir une compréhension approfondie des contraintes liées au langage de programmation, à l'utilisation et à la collaboration des

[3] OMT est le langage de modélisation orienté objet qui a été fusionné dans UML (principalement dans les diagrammes de conception).

différents composants du système ainsi qu'au système d'exploitation. Dans cette activité, on détermine les principales interfaces et on les transcrit en *diagrammes de classes* UML. Ceci permet ensuite de coder dans l'activité suivante avec un langage orienté objet de type Java ou C++. Ceci prépare la phase d'implémentation en décomposant le système en sous-systèmes qui correspondent aux différentes classes du système et en définissant le comportement interne de chaque classe par des diagrammes UML, par exemple des *diagrammes d'activités* ou des *machines à états*.

- L'*implémentation et déploiement* se basent sur la conception pour programmer le système sous forme de classes (codes sources, scripts, binaires ou exécutables). Les objectifs principaux de cette activité concernent la planification de l'intégration des composants pour chaque itération, et la production des classes et des sous-systèmes sous forme de code source. Le modèle de déploiement est décrit en UML par des *diagrammes de déploiement*.
- Le *test* permet enfin de confronter les résultats de l'implémentation avec des *cas de test* planifiés pour chaque itération à partir du recueil des besoins.

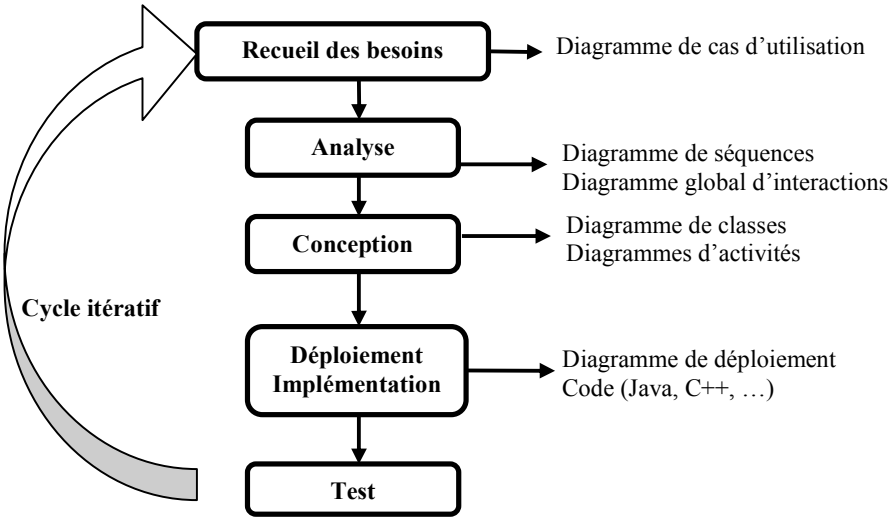

Fig.1. Cycle de vie des systèmes selon la méthode UP

La méthode 2TUP (*2 Track Unified Process*) [ROQ 04] est une méthode UP qui apporte une réponse aux contraintes de changements continuels imposés aux systèmes d'information. Le terme « *2 Track* » signifie littéralement que le processus suit deux chemins. Il s'agit des chemins *fonctionnels* formulés par le client et d'*architecture technique* définis par l'équipe de conception. Ce dernier chemin permet la conception d'un prototype dépendant le moins possible des aspects *fonctionnels* et réutilisable dans d'autres projets. Cette distinction nous parait donc une bonne piste dans la distinction des différentes *exigences fonctionnelles* (distingués par *fonctionnels* et *techniques*) dans le système mais ne permet pas spécifier les *exigences non-fonctionnelles* contrairement à la méthodologie présentée dans ce mémoire.

ROPES (*Rapid Object-Oriented for Embedded Systems*) [DOU 04] spécialise la méthode UP dans le cadre de la conception de systèmes temps réel. Cette méthode encourage la synthèse

automatique de code pour faciliter la génération rapide de prototypes à partir des modèles spécifiés dans le profil UML SPT [OMG 05] (cf. section 3.1.4) qui spécialise la norme UML dans le domaine des systèmes temps réel. La classification des exigences fonctionnelles (*prioritaire*, *risque*, *standardisation possible*) [DOU 04] présente une bonne solution pour définir de manière complète des *exigences fonctionnelles* (fonctionnalités). La méthodologie proposée au chapitre III reproduit la même approche en intégrant un élément manquant dans le cycle de développement à savoir la définition des *exigences non-fonctionnelles temporelles*, élément prépondérant dans le développement de systèmes temps réel.

Enfin, le survol de toutes ces méthodes de conception basées sur le langage semi-formel UML très utilisées dans le milieu industriel nous amène à faire le constat que la validation du système est basée uniquement sur le test du code. Ceci est regrettable car des fautes de conception dues à un défaut de validation *a priori* sont découvertes lors de la première phase d'implémentation. Des solutions peuvent être apportées par l'utilisation de méthodes formelles, permettant de générer un modèle exécutable pour détecter des erreurs de conception *a priori* lors des premières étapes du cycle de développement d'un système temps réel.

1.2.3. Les méthodes formelles dans la conception et la validation des STR

Parmi les nombreuses taxonomies applicables aux langages formels de modélisation, nous distinguons dans cette section :

- D'une part, les langages tels que la méthode B [ABR 96], qui relèvent du « *correct par construction* » dans la mesure où le concepteur démarre par une spécification embryonnaire et fait évoluer son modèle par une série de transformations qui garantissent à chaque étape la préservation des propriétés qui étaient satisfaites à l'étape antérieure. A chaque étape du développement, des *obligations de preuve* sont imposées. Une *obligation de preuve* est une propriété qui doit être satisfaite pour que le système construit soit « *correct par construction* » [ABR 96].
- D'autre part, les langages qui impliquent que le modèle doit être vérifié a posteriori et de manière systématique à la suite de chaque grande étape du le cycle de développement.

L'approche proposée ultérieurement dans ce mémoire repose sur cette seconde catégorie de langages.

S'appuyant également sur la deuxième catégorie de langage, les travaux de [BAB 01] présentent une méthode de développement de systèmes embarqués basée sur le langage SDL [SDL 99] et appelée PROSEUS (*Development method for PROtotyping embedded SystEms by using UML and SDL*). [BAB 01] fait l'hypothèse que l'analyse des besoins fonctionnels et la conception ont été spécifiées en UML puis traduits en SDL. Afin d'intégrer les contraintes temps réel dans le modèle du système, [BAB 01] propose un typage des signaux échangés entre système et événements qui modélise les contraintes de temps de l'application à développer. Les paramètres des signaux contiennent des caractéristiques temporelles exprimées en logique temps réel [JAH 86] et définies selon le mode de propagation [BAB 98]. L'étude se focalise ensuite sur la phase de définition de l'architecture de l'application, phase fondamentale dans les systèmes temps réel embarqués car les choix effectués au niveau de l'architecture influent directement sur les performances du système [BAB 01]. La définition de l'architecture matérielle et logicielle (qui peut être exprimée par des

diagrammes UML de déploiement et de composant) s'appuie sur des structures types définies en SDL qui représentent les unités de traitement du système, les médiums de communication et l'environnement. A chaque signal, on associe des contraintes temporelles à partir du modèle précédent. Cette approche autorise une modélisation fine des contraintes du système (liées aux unités de traitement du système, aux médiums de communication et à l'environnement). Ceci nous a donné des pistes pour définir les cycles itératifs de la méthodologie présentée dans ce mémoire (cf. chapitre III section 3.1.2 et 4.1.2). Le modèle du système doit être conçu dans un premier temps dans un contexte atemporel. Ensuite, les paramètres temporels sont insérés non pas dans les signaux comme en [BAB 01] mais dans les diagrammes d'activités TURTLE (cf. section 3.1.2.2 de ce chapitre).

S'inscrivant entièrement dans un processus UP, ACCORD$|_{UML}$ [ACCOR] est une méthodologie orientée développement de modèles d'applications temps-réel distribuées et embarquées. L'objectif général de la méthodologie est de masquer autant que possible les aspects d'implantation autour d'une approche dirigée par les modèles (patrons de conception, raffinement automatique de modèle, génération de code, validation par construction de modèles…), afin de permettre aux développeurs de se concentrer sur les aspects métier du système (fonctionnalités, contraintes de performance…) [PHA 04]. ACCORD concerne la spécification de systèmes temps réel décomposés par des *objets temps réel* étendus par les concepts suivants [GER 04] : une boîte à lettre pour la réception des requêtes et un contrôleur de concurrence et d'état spécifié par une machine à états arbitrant l'exécution des messages en fonction de l'état et des contraintes de concurrences associées reposant sur la représentation *continue* du temps. La validation dans la méthode ACCORD concerne trois aspects [PHA 04] : la *vérification*, la *simulation* et le *test*. L'aspect *vérification* utilise les méthodes formelles par le biais de l'outil AGATA [AGAT] en générant un modèle exécutable pour vérifier que le système décrit par le modèle de conception satisfait bien les exigences définies dans les diagrammes d'exigences. Ces dernières sont dérivées pour être spécifiées formellement soit par des formules de logique soit par des observateurs (une présentation de ces différentes techniques de vérification est proposée dans le chapitre V section 1). Les exigences vérifiées incluent les propriétés de sûreté, vivacité, blocages temporels et comportements non-conformes au cahier des charges reposant sur la description d'un temps *logique*. La méthode ACCORD$|_{UML}$ repose sur le profil UML ACCORD [ACCOR] mais peut être facilement dégagée de celui-ci pour être réutilisée dans d'autres environnements et intégrer l'aspect « validation » (soit la *vérification*, la *simulation* et le *test*) dans toutes les étapes du cycle de conception. Tout comme la méthode ACCORD, nous souhaitons, dans ce mémoire, nous dégager de profils spécifiques pour définir une méthodologie de conception de systèmes temps réel prenant en compte la phase de traitement des exigences (ce qui n'est pas défini dans la méthode ACCORD$|_{UML}$). La méthodologie présentée dans ce mémoire couvre donc la phase de spécification des *exigences non-fonctionnelles temporelles*, à formaliser au demeurant en vue d'une vérification formelle du modèle du système à concevoir.

Incluant le traitement des exigences par l'utilisation de *diagrammes d'exigences* SysML [SysML], MeMVaTEx [MeMV] (Méthode de Modélisation pour la Validation et la Traçabilité des Exigences) propose une méthodologie qui considère les exigences comme des classes conceptuelles définies dans les premières étapes de la modélisation et tracées durant tout le cycle de vie de développement du système. Cette méthode [ALB 07] permet, tout comme la méthodologie présentée dans ce mémoire, de faire la liaison entre les langages UML 2.0 [UML] et SysML [SysML]. MeMVaTEx est basée sur le profil UML MARTE [OMG 07] (cf. section 3.1.4) de modélisation de

systèmes temps réel. MeMVaTEx est orientée développement d'application temps réel dans le domaine de l'automobile. Le cycle de développement s'inspire de celui présenté dans le projet EAST-EEA [EAST]. Les exigences initiales sont données sous forme textuelle puis traduites par des diagrammes d'exigences SysML. Chaque exigence peut être connectée aux autres exigences, greffée à une partie du modèle ou encore représentée par un *cas de test*. Les exigences sont ensuite construites et tracées durant toutes les phases de développement en utilisant les mécanismes de traçabilité des exigences du méta-modèle SysML (concept de *composition*, *vérification*, *satisfaction*). La traçabilité des exigences est définie par construction de références entre le modèle et les documents externes où sont définies les exigences. Ces documents contiennent le résultat de la validation et vérification des exigences. Une *matrice de traçabilité* SysML contenant les résultats de validation des exigences peut alors être produite. Cependant l'aspect « formalisation des exigences » n'est pas décrit exhaustivement dans MeMVaTEx [ALB 07]. Nous présupposons donc que cette méthode n'inclut pas encore un processus de description formelle des exigences temporelles. Centrée autour de la vérification formelle d'exigences temporelles, l'approche proposée dans ce mémoire prend au contraire en compte l'aspect « recueil d'exigences » et la manière de formaliser les *exigences non-fonctionnelles* par une dérivation (cf. chapitre IV). De plus, la méthode MeMVaTEx repose sur le profil UML MARTE [OMG 07] (cf. section 3.1.4) pour le développement d'applications automobiles. La méthodologie présentée dans ce mémoire se veut être plus générale ; elle se dégage d'outils UML spécifiques et s'applique aux systèmes temps réel en général avec une particularisation au domaine des systèmes temps réel et distribués.

2. Systèmes temps réel et distribués (STRD)

2.1. Les exigences temporelles liées à la Qualité de Service des STRD

Afin d'optimiser les ressources d'un réseau et de garantir de bonnes performances aux applications critiques, le concept de *Qualité de Service* (QoS) a permis de standardiser les exigences non-fonctionnelles liées, dans le domaine temporel, aux débits et aux temps de réponse des applications qui utilisent les services à concevoir. Généralement les *exigences non-fonctionnelles temporelles* pour le développement de protocoles sont incluses dans ce concept. On distingue deux classes d'exigences liées aux contraintes de QoS temporelles [STE 93] [EXP 03] :

- La gigue entre la réception de deux paquets (*Jitter*). Cette exigence correspond à l'intervalle temporel de réception par une même machine de protocole de deux paquets consécutifs.
- Le délai de bout en bout pour la réception d'un paquet (*End_to_End_Delay*). Cette exigence est équivalente au délai de transmission d'un paquet entre deux machines de protocoles.

Un lien existe entre ces classes d'exigences et les différentes propriétés définies dans le Tab.1 en section 1.1 de ce chapitre. Par exemple le *Jitter* peut être associé à la propriété de délai maximum ou minimum et le *End to end delay* peut être associé à la propriété de délai borné.

Parmi les travaux dédiés à la spécification des exigences liées à la QoS, [EXP 03] présente *XQoS*, un langage de spécification basé sur XML pour définir, entre autre, les exigences de QoS en termes de temps (par les exigences de gigue et de délai bout en bout), mais aussi la synchronisation et l'ordre des messages. Ces travaux, qui concernent uniquement l'aspect spécification et non la

vérification de ce type d'exigences, nous ont inspiré pour définir la taxonomie des exigences non-fonctionnelles temporelles liés à la QoS présenté dans le paragraphe précédent. Concernant les protocoles temps réel, [TOU 05] propose de définir la QoS par des contraintes quantitatives sur l'appel d'un service offert par un composant qui est vu comme une ressource. La formalisation de la QoS repose donc sur les contraintes d'utilisation de cette ressource (en termes de capacité mémoire, occupation du CPU, ressources réseau, ...). Cette formalisation des exigences dans l'architecture orientée QoS *Qinna* [TOU 05] se construit sous forme de contrats sous forme de bases de données. Les contraintes temporelles sont traitées par analyse des performances temporelles et non formalisées au niveau des contrats car difficiles à encapsuler dans ces derniers. Au contraire, l'approche présentée dans ce mémoire se focalise sur la description d'exigences temporelles telles que la gigue et le délai de bout en bout (cf. section 1). Ceci permet de décrire entre autres les *exigences non-fonctionnelles* auxquelles un protocole est confronté (par exemple, une application multimédia qui impose des exigences temps réel en termes de gigue et de délai de bout en bout).

2.2. Méthodologies de conception de STRD

Dans le domaine des protocoles, il est essentiel de pouvoir définir de manière complète et non-ambigüe les règles de dialogue (*protocole*), utilisées pour l'échange de données et la synchronisation entre les entités du protocole. La Fig.2 décrit le pattern d'architecture qu'il est convenu d'appliquer aux protocoles en s'inspirant de [ZIM 80]. L'architecture est décrite par un empilement de modules ou *couches de protocole* [ZIM 80]. Un *protocole (N)* est défini par l'ensemble des règles de dialogue et des formats de messages caractérisant la communication des entités de la *couche (N)*. Les *entités de protocole (N)*, fournissent un *service (N)*, décomposé en *primitives de services(N)*, à la couche supérieure en coopérant entre elles suivant le *protocole (N)*. Les entités de protocoles utilisent le *service (N-1)* pour échanger des messages (les *(N) PDUs* sur la Fig.2). Il est communément admis ([BOC 90] [PEH 90] [COU 91]) de distinguer trois niveaux dans une architecture dédiée à la vérification :

- Le niveau N+1 décrit l'abstraction des couches supérieures qui vont utiliser les primitives de service du *protocole (N)*.
- Le niveau N décrit les entités de protocole à vérifier qui fournissent le *service (N)* en utilisant le *service sous-jacent (N-1)*.
- Le niveau N-1 décrit le modèle du *service sous-jacent (N-1)* qui est utilisé par les *entités de protocoles (N)* pour échanger leurs *(N) PDUs*.

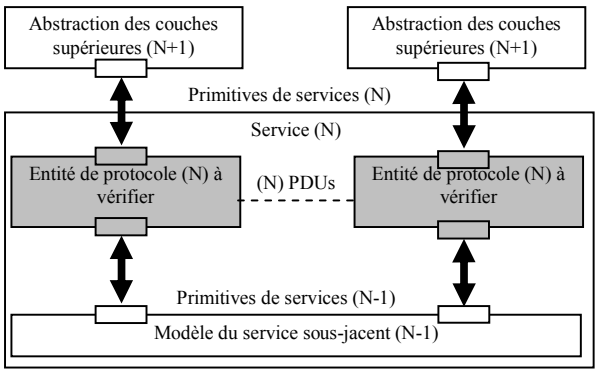

Fig.2. Architecture à trois couches inspirée du modèle OSI

L'architecture à trois couches décrite par la Fig.2 est réutilisée dans la méthodologie présentée dans ce mémoire (cf. règle C.1 de la section 4.2 du chapitre III). En termes d'abstractions et d'itérations dans le cycle de conception d'un protocole, cette méthodologie se base sur les travaux de [BOC 90] et [COU 91] (cf. Fig.20 présentée dans la section 4.1.3 du chapitre III) qui préconisent la conception d'un protocole (N) en utilisant trois niveaux d'abstraction :

- La *spécification abstraite du service (N)*, ayant pour objet de représenter l'ordonnancement temporel des primitives de services, telles qu'elles sont visibles par les utilisateurs de la couche (N). Cette phase correspond à la *phase d'expression des besoins utilisateurs*.
- La *spécification abstraite du protocole (N)*, ayant pour objet de raffiner la spécification précédente en introduisant les fonctionnalités des entités de protocoles de la couche (N), ainsi que le « mappage » des (N) PDUs échangés entre les entités sur les primitives de services de la couche (N-1). Cette phase correspond à la phase de *conception de l'architecture*.
- La *spécification concrète du protocole (N)*, correspond à la *phase d'implémentation* (prise en compte de l'environnement système/matériel particulier sur lequel le protocole sera implémenté).

Dans le domaine de la vérification des protocoles, la représentation du service sous jacent est un élément très important pour la conception d'un modèle de protocole. La méthodologie de conception de protocoles de niveau Transport présentée en [COU 92] développe l'idée de concevoir un protocole autour de la description du service sous-jacent (ou *medium*). Il s'agit d'une démarche itérative et incrémentale, basée sur le modèle du service sous-jacent. [COU 92] considère deux propriétés :

P1. Le *médium* (ou la couche de service sous-jacent) est capable de transmettre des données de taille arbitraire. Ce qui veut dire que l'on ne considère pas la *segmentation de données*.

P2. Le *médium* est considéré comme *fiable* (aucune perte, aucune duplication, pas de déséquencement).

En fonction de ces deux propriétés injectées ou non dans le modèle du médium, la conception du protocole est caractérisée par les étapes suivantes :

- La première étape impose la satisfaction des propriétés P1 et P2 par le médium qui sera donc vu comme fiable et sans segmentation de données.
- Dans la deuxième étape, le médium satisfait seulement la propriété P2. Cette étape de conception correspond donc à l'intégration des mécanismes de segmentation/réassemblage des données.
- La troisième étape correspond à l'introduction des mécanismes de reconnexion. Le médium ne satisfait plus les propriétés P1 et P2.
- La dernière étape correspond au raffinement (à l'éclatement) des données dans le protocole. Chaque PDU considéré dans les étapes précédentes est représenté par les services de la couche sous-jacente. Le médium ne satisfait pas non plus les propriétés P1 et P2.

Ces travaux nous ont fortement inspiré pour définir les cycles d'itérations de la méthodologie présentée dans ce mémoire (voir la définition des *Aléas* au chapitre III section 3.1.2).

Tout comme les méthodologies UML présentées en sections 1.2.2 et 1.2.3, les méthodologies de conception de systèmes temps réel et distribués orientées UML [LEP 00] [POP 06] [MUS 07] prennent en compte la spécification des *exigences fonctionnelles* (appelées très souvent *besoins*) par des cas d'utilisation. [MUS 07] présente une approche de modélisation à base de scénarii : les *Uses Cases Maps* (UCM). Les UCM autorisent la définition de contraintes temporelles [HAS 06] assimilables à des *exigences non-fonctionnelles temporelles*. Mais cette méthodologie UML [MUS 07] n'intègre pas explicitement la spécification d'*exigences non-fonctionnelles*, car UML ne permet pas de spécifier et de différencier explicitement ce type d'exigences par rapport aux fonctionnalités identifiées par les *cas d'utilisations*. Au contraire, la méthodologie présentée dans ce mémoire permet d'exprimer les *exigences non-fonctionnelles* temporelles dans un contexte SysML et de manière formelle par construction de chronogrammes (cf. chapitre IV) qui sont le point de départ de la vérification formelle de ces exigences par le biais du profil UML temps réel TURTLE.

3. TURTLE (Timed UML and RT-LOTOS Environment)

La méthodologie proposée dans ce mémoire se veut être générique avant d'être instanciée au profil UML temps réel TURTLE. Ce dernier est présenté en section 3.1, avant d'aborder en section 3.2 les éléments de méthodologie [APV 06] qui avaient été proposés pour TURTLE avant le démarrage de nos travaux.

3.1. Un profil dédié à la vérification de systèmes temps-réel et distribués

En tant que « profil UML temps réel», le langage de modélisation TURTLE spécialise la notation UML de l'OMG [UML] pour les besoins de la modélisation de systèmes dans lesquels le respect de contraintes temporelles est une préoccupation première.

3.1.1. La sémantique TURTLE : le langage RT-LOTOS

Le langage TURTLE repose sur l'algèbre de processus temporisée RT-LOTOS (*Real Time LOTOS*) [COU 00] construite sur la technique de description formelle LOTOS [BOL 89] (*Langage of Temporal Ordering of Sequences*) normalisée à l'ISO [IS 8807].

Un processus LOTOS est une boîte noire qui communique avec son environnement au travers de portes et sur le principe d'une offre de rendez-vous. Des échanges monodirectionnels ou bidirectionnels de valeurs sont autorisés lors de la synchronisation. Le parallélisme et la synchronisation entre processus s'expriment par des opérateurs de composition : mise en séquence, synchronisation sur certaines portes, choix non-déterministe et entrelacement (composition parallèle sans synchronisation). Enfin, l'opérateur de composition de préemption permet de décrire le fait qu'un processus puisse interrompre à tout moment d'autres processus. La finalité de ce langage est de décrire l'ordonnancement temporel des actions. La version de LOTOS normalisée en 1988 n'offre ni opérateur temporel, ni mécanisme temporel de base de type « temporisateur » par exemple.

RT-LOTOS étend LOTOS avec trois opérateurs temporels principaux :

- L'opérateur de délai (noté ***delay(d)***) qui permet de retarder un processus d'une certaine quantité de temps d.
- L'opérateur de latence (noté ***latency(l)***) qui permet de retarder un processus d'une certaine quantité de temps choisie de manière non-déterministe au sein de l'intervalle de latence [0 ; l].

Notons que cet opérateur porte sur la ou les premières actions du processus auquel il est appliqué. Par ailleurs, il n'a d'effet que s'il porte sur une action interne ou une action externe offerte avant la date l, l'occurrence d'une action externe étant, en tout état de cause, soumise à la date de l'offre faite par l'environnement.

- L'opérateur de restriction temporelle (noté ***a{T}***) limite le temps pendant lequel une action observable *a* peut-être offerte à son environnement. Le délai d'expiration commence à courir à partir du moment où l'action est offerte.

On peut noter que la combinaison d'un opérateur de délai et de latence permet de traiter les intervalles temporels.

RT-LOTOS permet aussi de faire la distinction entre actions urgentes et non-urgentes. Ces dernières dépendent soit de l'environnement avec lesquelles elles se synchronisent, soit d'un opérateur temporel (délai, latence). Les actions urgentes qui ne dépendent ni de l'environnement ni d'opérateurs temporels sont exécutées instantanément (de manière *atomique*). Ces actions urgentes peuvent être aussi le résultat d'une synchronisation toujours offerte par l'environnement.

3.1.2. Les diagrammes UML TURTLE

Dans sa version initiale [APV 04], le profil UML TURTLE personnalisait les diagrammes de classes et d'activité UML pour adresser les aspects « architecture » et « comportement » d'une conception de systèmes temps réel ou distribués. Les travaux présentés en [APV 06] ont introduit l'aspect analyse du système à l'aide d'un diagramme de séquences. Ceci a amené à la synthèse automatique d'une conception TURTLE à partir des diagrammes d'analyse TURTLE (*diagrammes de séquences* notés SD et *diagrammes globaux d'interactions* notés IOD). Cette synthèse permet de donner une sémantique formelle aux IOD et SD. Ces deux types de diagrammes ont été enrichis avec des opérateurs dédiés aux systèmes temps-réel et distribués. Notons que l'aspect « déploiement » est aussi intégré dans le profil TURTLE pour la définition de *diagrammes de déploiement* [APV 06]. Ces derniers ne sont pas traités dans ce mémoire qui se limite aux phases amont du cycle de développement d'un système temps réel.

3.1.2.1 Les diagrammes de conception TURTLE

a) Le diagramme de classes TURTLE

Un diagramme de classes TURTLE se démarque d'un diagramme UML classique par l'ajout d'opérateurs de composition aux associations entre classes (cf. Fig.4). Aux classes UML de base caractérisées par un nom, des attributs et des méthodes, le profil TURTLE ajoute des classes stéréotypées par « *Tclass* » (cf. Fig.3) dotées d'attributs particuliers appelés *gates* qui sont autant de portes de communications. Ces *gates*, instances du type abstrait *Gate* sont bidirectionnelles. Ajoutons à ceci que toute *Tclass* contient un diagramme d'activité modélisant son comportement (cf. Fig.3).

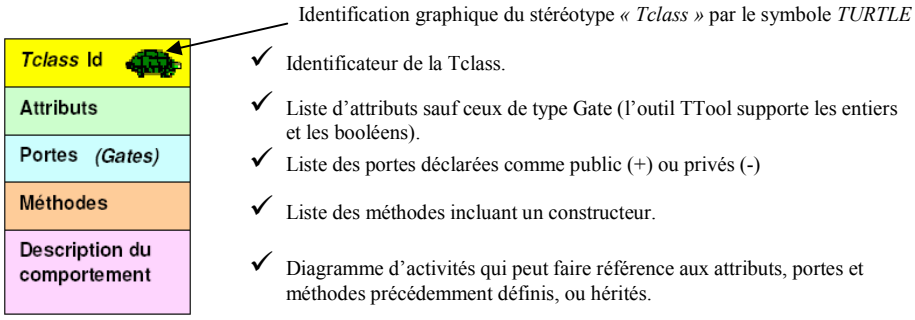

Fig.3. Structure d'une TClass

Constatant le caractère « implicite » du parallélisme entre classes UML et l'absence d'opérateurs de composition qui font la force des langages de modélisation basés sur les algèbres de processus, TURTLE introduit la notion d'*opérateur de composition* par l'entremise de *Tclasses* associatives attachées aux associations (bipoints) entre deux portes de communication (*Gates*) des *Tclasses* (cf. Fig.4). Notons bien que ce sont les instances de classes que l'on compose et non les classes ellles-mêmes. Cependant, par souci d'allègement des diagrammes, dans le cas fréquent où une classe n'est instanciée qu'une fois, le diagramme « confond » la classe et l'instance de celle-ci.

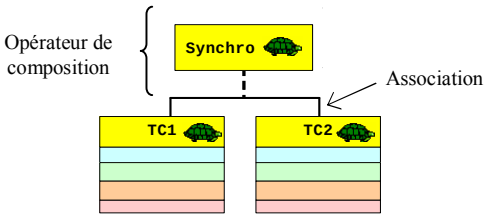

Fig.4. Composition entre deux instances de classes

TURTLE dispose d'opérateurs de composition inspirés de ceux de RT-LOTOS: *Parallel* pour deux *Tclasses* qui s'exécutent en parallèle sans aucune communication ; *Sequence* pour deux instances de *Tclasses* qui s'exécutent séquentiellement; *Synchro* pour que les instances de *Tclasses* puissent se synchroniser et éventuellement échanger des données à cette occasion ; enfin, *Preemption* pour permettre à une instance de *Tclass* d'interrompre une autre instance de *Tclass*. Pour éviter tout risque de confusion qui pourrait apparaître lors de l'utilisation conjointe de ces associations entre *Tclasses*, des priorités sont données. L'ordre est le suivant [LOH 02] :

Priorité (*Preemption*) > Priorité (*Sequence*) > Priorité (*Synchro*) > Priorité (*Parallel*)

b) Le diagramme d'activités TURTLE

Les diagrammes d'activités sont utilisés pour définir les comportements internes aux classes. TURTLE étend les diagrammes d'activités UML avec un retard déterministe, un retard non déterministe et une offre limitée dans le temps. Ces opérateurs temporels sont d'autant d'opérateurs empruntés à RT-LOTOS (cf. section 3.1.1) qui expriment des besoins de décrire le temps [LOH 02] :

- Expression d'un délai temporel après lequel la réalisation d'une action est possible (opérateur *Delay(T)* dans le Tab.2).

- Expression d'un système de déroutement lorsqu'une action n'est pas réalisée dans l'intervalle attendu (opérateur *TLO (t, a, <AD1>, <AD2>)* dans le Tab.2).
- Expression du non déterminisme temporel, à savoir le fait qu'une action réalisée par une activité se produise après un délai variable, non connu a priori (opérateur *Latency(t)* dans Tab.2).

Le Tab.2 présente les différents éléments du langage graphique exprimables dans un diagramme d'activités (AD).

Label	Symbole Graphique	Description informelle
Begin	**AD**	Début d'un diagramme d'activités.
G !x ?y	g!x?y **AD**	Appel sur la gate g avec émission de x et réception de y.
Choice (([g1],<AD1>)+ ([g2], <AD2>) + ([g3], <AD3>))	[g1] [g3] **AD1** [g2] **AD3** **AD2**	Sélection de la première activité prête à s'exécuter et dont la garde est vraie, parmi les 3 activités décrites par AD1, AD2 et AD3.
Parallel ([g1, …, gk], <AD1>, …, <AND>)	[g1,g2] **AD1** **ADN**	Synchronisation sur les portes g1, g2 entre les sous activités AD1, …,ADN.
Preemption (<AD1>, <AD2>)	**AD1** **AD2**	Interruption de l'activité AD1 si l'activité AD2 est prête à s'exécuter.
Sequence (<AD1>, <AD2>)	**AD1** **ADN**	L'activité ADN est effectuée en séquence après que les activités précédentes AD1,…, ADN-1 soient terminées.
Delay (T)	T **AD**	Retard déterministe de T unités de temps.
Latency (t)	t **AD**	Retard non déterministe de t unités de temps.
TLO (t, a, <AD1>, <AD2>)	a t **AD1** **AD2**	Offre limitée dans le temps. L'offre sur la Gate a est offerte pendant une période inférieure ou égale à t. Si l'offre sur a est réalisée alors AD1 est exécutée sinon AD2 est interprétée.

TLO_L (t, l, a, <AD1>, <AD2>)	l a t AD1 AD2	Offre limitée dans le temps avec latence. L'offre sur la Gate a est offerte pendant une période inférieure ou égale à t en considérant une latence de l. Si l'offre sur a est réalisée alors AD1 est exécutée sinon AD2 est interprétée.
Stop		Terminaison d'une activité.
Loop (<AD>)	AD	AD est interprétée à chaque fois que l'on entre dans la boucle.

Tab.2. Sémantique des éléments du diagramme d'activité TURTLE

3.1.2.2 Les diagrammes d'analyse TURTLE

a) Le diagramme global d'interactions TURTLE

Le diagramme global d'interactions (noté par la suite IOD) permet de conduire une analyse de comportement d'un système à un haut niveau d'abstraction. Apparus avec la norme UML 2.0 [UML], les IOD sont relativement similaires à des diagrammes d'activités. Ils supportent des opérateurs de choix, de parallélisme, de synchronisation et de boucle en y ajoutant des références à des scénarii. Un opérateur de préemption a été ajouté dans le profil TURTLE pour décrire des scénarii susceptibles d'en interrompre d'autres à tout instant. Par exemple, la Fig.5 montre un scénario de transfert de données et une libération de connexion (décrit par le choix gardé entre les scenarii *DataTransfer* et *ConnectionRelease*) qui peuvent tous deux être interrompus à tout moment par le scénario de coupure de connexion (*ConnectionBroken*).

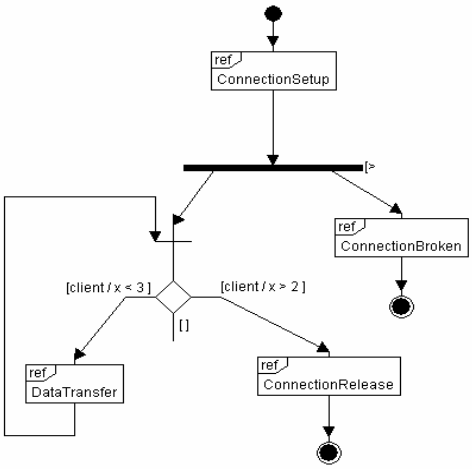

Fig.5. Exemple de diagramme global d'interactions TURTLE

b) Le diagramme de séquences TURTLE

TURTLE reprend la majorité des opérateurs logiques du diagramme de séquences UML 2.1 (synchronisation et envoi réception de messages asynchrone), les opérateurs temps réel (opérateurs de temps absolu et relatif, notion de « *timers* », intervalles temporels). La Fig.6 montre un exemple de

diagramme de séquences décrivant le cas nominal d'un protocole de connexion à distance du type *ping*. L'application envoie un message *Ping_Command* transmis via un réseau dont le temps de transmission est compris entre 6 et 12 unités de temps (rectangle avec un ressort). L'application attend un message Print_Ping clôturant la commande *Ping_Command*.

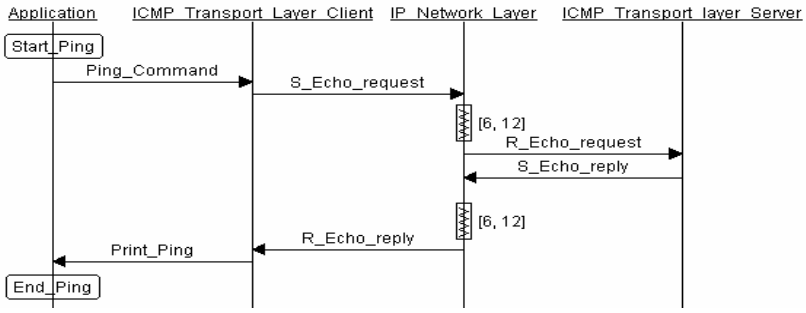

Fig.6. Exemple de diagramme de séquences

3.1.3. Un profil outillé

Comme le montre la Fig.7, la chaîne d'outil TURTLE est composée de :

- TTool (*TURTLE Toolkit*) [TTOOL], présenté dans le rectangle en pointillé de gauche de la Fig.7, est un outil *open source* qui inclut un éditeur graphique de diagrammes TURTLE archivés dans un format XML, d'un analyseur syntaxique, un générateur de code RT-LOTOS et, enfin des outils de visualisation et d'analyse des résultats de simulation et de validation retournés par l'outil RTL introduit dans le prochain infra. Les différents diagrammes TURTLE (analyse et conception) sont toujours traduits dans un format intermédiaire appelé TIF (*TURTLE Intermediate Format*). Ce format intermédiaire est très proche des diagrammes de conception TURTLE. Il sert de point de départ de la génération de spécifications formelles en RT-LOTOS (cf. Fig.7).
- RTL (*RT-LOTOS Laboratory*) [RTL], présenté dans le rectangle en pointillé en haut à droite de la Fig.7, admet en entrée une spécification RT-LOTOS et propose deux approches de validation complémentaires : la simulation intensive qui explore partiellement l'espace d'état du système modélisé par les diagrammes TURTLE et l'analyse d'accessibilité. L'analyse d'accessibilité s'applique aux systèmes bornés et de taille raisonnable. Si l'analyse est possible, un graphe d'accessibilité (*Système de Transitions Etiqueté*) est produit en sortie.
- CADP (*Construction and Analysis of Distributed Processes*) [CADP], présenté dans le rectangle en pointillé en bas à droite de la Fig.7, propose entre autre une boîte à outils d'analyse pour les STE (en termes d'état puits, de statistique de chemins atteignables). Nous utilisons en TURTLE les outils BCG_MIN et SIMULATOR permettant respectivement la minimisation d'un STE (produisant un *automate quotient* par rapport à une relation d'équivalence) et la comparaison de deux STE par rapport à une équivalence donnée.

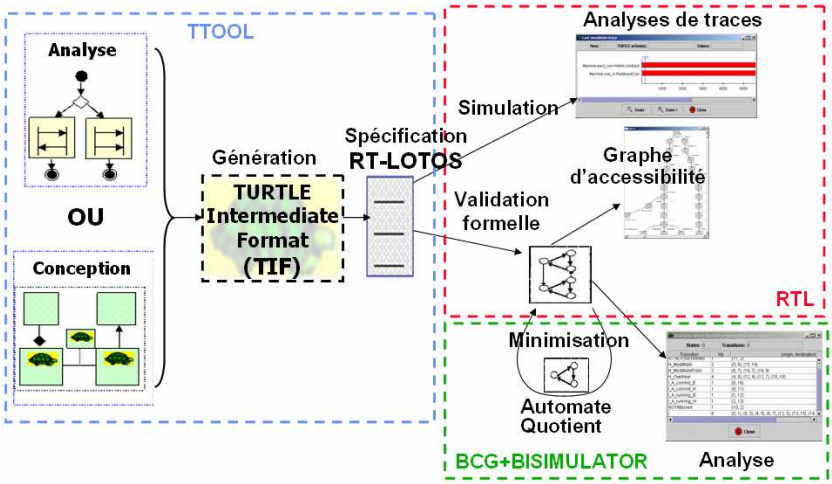

Fig.7. La chaîne d'outils du profil TURTLE

L'interface entre les outils TTool et RTL permet de générer des traces de simulation ou un graphe d'accessibilité de façon transparente depuis TTool, c'est-à-dire sans écrire ou examiner une seule ligne de code RT-LOTOS.

3.1.4. Positionnement du profil TURTLE

Pour la modélisation des systèmes critiques, deux approches ont été proposées dans la communauté UML temps réel : l'introduction d'opérateurs logiques (principalement des opérateurs pour composer des classes UML) et des opérateurs temporels au niveau de la description comportementales des classes. Dans la majorité des profils UML temps réel, la composition logique repose sur une communication asynchrone [CLA 00] [DOU 02] [GTT 02] [SEL 01] [TAU]. Cette composition est généralement masquée et implicite [SEL 01] [TAU]. A l'opposé de ces compositions asynchrones qui anticipent sur des choix d'implantation (communication par file), TURTLE intègre une composition explicite et formellement définie, basée sur une communication synchrone (par rendez-vous à la LOTOS). Du point de vue temporel, les profils décrits en [CLA 00] [DOU 02] [GER 02] [SEL 01] utilisent de très classiques temporisateurs ; ainsi, ils ne permettent ni d'exprimer une latence ni de traiter explicitement l'indéterminisme temporel auquel le système est confronté. Au contraire, TURTLE permet grâce à ses opérateurs temporels de base d'exprimer des contraintes temporelles et des phénomènes tels que la variation temporelle de certains processus (phénomène de gigue) fréquemment constatée dans les systèmes communicants.

Parmi les projets qui placent la validation a priori basée sur les modèles au centre du processus de conception de systèmes temps réel et embarqués, nous pouvons mentionner le projet OMEGA [OMEGA] et son profil UML temps réel : OMEGA-RT [GRA 05]. Les modèles édités dans les outils externes au projet OMEGA (*ROSE-RT*, *Rhapsody* et *Argo*) sont traduits dans le formalisme IF supporté par des outils de validation formelle. La démarche est similaire dans le cas du profil TURTLE ; les modèles sont traduits en TIF puis dans le formalisme RT-LOTOS [COU 00]. Au niveau des diagrammes de classes, OMEGA-RT supporte l'orientation objet d'UML en termes d'héritage. Par contre, la notion d'opérateur de composition que TURTLE hérite des algèbres de processus n'est pas représentée dans les diagrammes de classes d'OMEGA-RT. Par ailleurs

contrairement à TURTLE, OMEGA-RT dote les objets de ports de communication asynchrones. Au niveau comportemental, les contraintes temporelles sont traitées au niveau des événements alors que TURTLE offre trois opérateurs temporels génériques dont un opérateur de latence pour exprimer l'indéterminisme temporel qui semble difficile à représenter dans le profil OMEGA-RT.

Parmi les profils adoptés par l'OMG dans le domaine des systèmes temps réel, le profil UML SPT [OMG 05] (*Schedulability, Performance and Time*) ne considère qu'un temps métrique qui fait implicitement référence au temps physique (expression quantitative du temps). UML SPT introduit les concepts d'instant (*instant*) et de durée (*duration*). Il modélise également les mécanismes temporels classiques dans UML (*clock* et *timer*). Successeur de SPT à l'OMG, le profil MARTE (*Modeling and Analysis of Real-Time and Embedded systems*) [OMG 07] voit ses aspects temporels regroupés dans le sous profil *Time* [AND 07]. L'intérêt de MARTE repose sur la distinction entre plusieurs référentiel de temps (par l'ajout d'un stéréotype « clock »). MARTE modélise aussi bien le temps physique (continu) que le temps *logique*. Une autre caractéristique de MARTE est de pouvoir lier directement et explicitement des éléments comportementaux au temps par les stéréotypes « TimedEvent » (événement dont l'occurrence est directement liée à une horloge) et « TimedProcessing » (activité liée à une horloge). Il en est de même avec les contraintes par le stéréotype « TimedConstraint » (activité liée à une contrainte temporelle) et les observations par « TimedObservation » (observation du temps d'une activité). Doté de beaucoup moins de stéréotypes pour décrire le temps, TURTLE permet par le concept de latence de représenter explicitement l'indéterminisme temporel relatif à l'occurrence d'un événement contrairement à MARTE.

3.2. Méthodologie présentée en [APV 06]

[APV 06] présente une méthodologie de développement de systèmes temps réel et/ou distribués (cf. Fig.8) basée sur le profil TURTLE. Cette méthodologie inclut une phase de déploiement qui ne sera pas traitée dans ce mémoire. [APV 06] définit trois étapes :

1. L'analyse comprend la réalisation d'un diagramme de cas d'utilisation pour définir les différentes fonctionnalités du système (ce diagramme n'ayant pas de sémantique formelle, il n'est produit qu'à titre documentaire) et d'un ou de plusieurs diagramme(s) global d'interaction et de diagrammes de séquences (ces deux derniers peuvent être validés formellement). Cette d'analyse a pour objectif la réalisation d'un sous-ensemble de scénarii possibles au regard des fonctionnalités décrites dans le cahier des charges d'un système.
2. La conception permet de produire une première définition de l'architecture sous la forme d'un diagramme de classes décrivant l'aspect architectural du système. L'architecture globale du système étant définie, le comportement de chaque entité est décrit par des diagrammes d'activités. Le concepteur peut alors ensuite faire évoluer cette conception et vérifier formellement, par la génération à partir du modèle TURTLE d'un graphe d'accessibilité, le bon comportement du système.
3. La validation formelle du modèle TURTLE (soit par les diagrammes d'analyse soit par les diagrammes de conception) qui va guider l'évolution du modèle TURTLE par utilisation de techniques de comparaisons comme la bisimulation prouvant que la conception (ou l'analyse) de l'étape n+1 possède les mêmes ensemble de traces qu'à l'étape n, lorsque bien entendu les traces de l'étape n+1 sont épurées des actions non présentes dans l'étape n. Cette bisimulation s'effectue ainsi entre le graphe d'accessibilité de l'étape n, et le graphe d'accessibilité de l'étape n+1 minimisé par rapport aux actions de l'étape n.

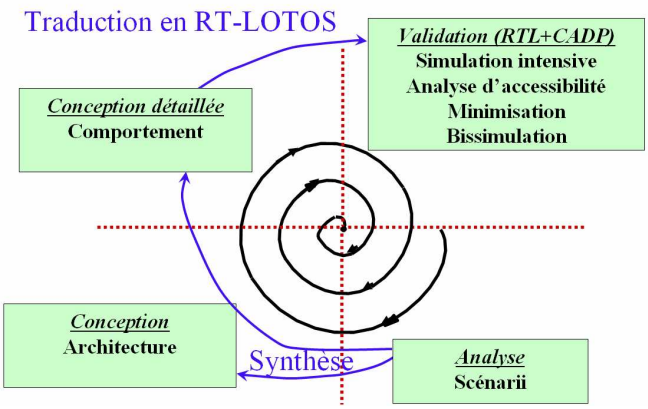

Fig.8. Méthodologie TURTLE pour la conception de systèmes temps réel

Il faut noter que la méthodologie présentée en [APV 06] repose sur une approche incrémentale (cf. spirale de la Fig.8). A chaque étape, la sémantique formelle de TURTLE permet de réaliser des validations formelles, et éventuellement de faire des preuves de conservation de propriétés par utilisation de techniques de bisimulation entre modèles de base et raffiné. Ceci garantit une conception non-régressive d'un système pour atteindre finalement le centre de la spirale que montre la Fig.8.

A l'instar des autres méthodologies UML présentées dans les sections précédentes, celle proposée en [APV 06] ne couvre pas la phase de recueil d'exigences (UML disposant uniquement de diagrammes de *cas d'utilisation*). Outre une description plus exhaustive qu'en [APV 06], la méthodologie présentée dans ce mémoire (cf. chapitre III) repose sur l'ajout de la phase de recueil des exigences et la formalisation des exigences non-fonctionnelles (cf. chapitre IV), point de départ de la vérification formelle et de la traçabilité de ce types d'exigences dans le cycle de développement d'un modèle dédié à la vérification (cf. Fig.8).

4. Conclusion

Le survol des méthodologies de conception de systèmes temps réel utilisées dans l'industrie nous amène à faire le constat suivant : il existe très peu de méthodologies axées sur le langage UML et prenant en compte l'aspect « vérification formelle ». La première idée défendue dans ce mémoire repose sur l'idée de coupler l'aspect conception par construction d'un modèle UML avec la vérification formelle.

Les méthodologies de conception de systèmes temps réel prenant en compte l'aspect validation et le langage UML [PHA 04] [APV 06], ne permettent pas de distinguer explicitement les *exigences fonctionnelles* (représentés par des fonctionnalités ou *Uses Cases*) et les *exigences non-fonctionnelles* très souvent décrites dans les fonctionnalités (par exemple par des *Uses Cases Maps* [MUS 07]) voire même occultées. Notons qu'il en est de même pour les méthodologies de conception de systèmes temps réel et distribués. La deuxième idée défendue dans ce mémoire est donc d'ajouter une étape supplémentaire de *traitement des exigences* (cf. chapitre III) en amont du cycle de conception pour pouvoir faire la distinction explicite entre exigences fonctionnelles et non-fonctionnelles. Les *diagrammes d'exigences* SysML [SysML] autorisent l'expression de ces deux

types d'exigences par opposition à UML dont les *cas d'utilisation* traitent uniquement les *exigences fonctionnelles*.

En s'inspirant de la méthodologie MeMVaTEx [ALB 07], nous avons cherché à lier la définition des *exigences non-fonctionnelles* et la validation de ces dernières dans une *matrice de traçabilité* construite automatiquement à partir de construction de références entre modèles du système et d'exigences présentées de manière informelle. Notre approche se distingue de [ALB 07] dans le fait que nous présentons aussi un langage graphique permettant de décrire de manière formelle les *exigences non-fonctionnelles temporelles* en exprimant le temps de manière continue (cf. chapitre IV), alors que [ALB 07] ne traite pas de cet aspect de formalisation des exigences.

Enfin la différence principale entre les méthodologies de validation formelle d'exigences présentées dans ce chapitre et les contributions des chapitres à venir, repose sur la vérification d'*exigences non-fonctionnelles temporelles* par un processus automatique. Ce mémoire présente dans son chapitre V un dispositif de génération automatique d'observateurs permettant de tracer les *exigences non-fonctionnelles temporelles*, alors que dans [PHA 04] les observateurs sont construits « à la main ».

Chapitre III. Proposition d'une méthodologie

Bien que critiquable en certains points (sémantique ou expression de contraintes temporelles par exemple), le langage UML demeure une référence : il résulte en effet d'un consensus entre membres de l'OMG [OMG] (Object Management Group) qui se sont efforcés d'intégrer les « *meilleures pratiques* » du génie logiciel. Ce travail de normalisation a abouti à une notation et en aucun cas une méthodologie. Par ailleurs, UML 2.1 [UML] n'inclut pas moins de 13 diagrammes. La difficulté réside donc à trouver un compromis entre les différentes constructions de diagrammes UML possibles dans un domaine d'application donné (nous nous concentrons dans ce mémoire sur les systèmes à temps contraints et en particulier les protocoles). L'objectif de ce chapitre est donc de choisir judicieusement des diagrammes UML et SysML à utiliser dans le domaine des systèmes temps réel et des protocoles.

Les méthodologies recensées dans le chapitre II sont majoritairement décrites de manière textuelle, hors de tout langage dédié à la description de méthodologie. De plus dans la plupart des articles orientés autour de la conception de modèles UML basés sur les méthodes formelles, les éléments d'ordre méthodologique concernent l'aspect « outillage » et la manière de mettre en œuvre ces outils. Les aspects méthodologiques orientés « utilisateur » sont très souvent occultés ou exprimés sous forme textuelle.

Au contraire, ce chapitre définit formellement un langage de description de méthodologie, avant d'élaborer une méthodologie de conception à base d'UML, SysML et méthodes formelles.

Une originalité réside dans le traitement de la *formalisation des exigences non-fonctionnelles temporelles* qui n'est pas abordé dans les méthodologies orientées UML exposées en [BOO 00] [ROQ 04] [DOU 04] [DUB 06] [APV 06].

Ce chapitre est organisé de la manière suivante. La section 1 définit le langage de description de processus méthodologique utilisé dans la suite de ce chapitre. La section 2 présente le domaine d'application de la méthodologie dans un tableau qui sert de guide de lecture pour ce chapitre. La section 3 expose le cœur de la contribution du chapitre à savoir une méthodologie de conception de systèmes temps réel en contexte UML/SysML, en se dégageant d'outils UML particuliers dans un premier temps puis en l'instanciant sur le profil UML TURTLE. Enfin, la section 4 traite de la spécialisation de cette méthodologie aux protocoles.

1. Vue d'ensemble de la méthodologie

1.1. Définition informelle

La définition d'une méthodologie repose sur plusieurs éléments [DOU 04] :

- Un *périmètre*. Il s'inscrit dans les phases préliminaires du cycle de développement des systèmes. Ceci correspond aux phases de *préétude* et d'*élaboration* d'une méthode UP

(cf. chapitre II section 2.2). Orientée autour de la vérification, la méthodologie présentée dans ce mémoire repose sur l'idée que la vérification reste un filtre d'erreurs mis en œuvre *a priori* et permet d'évaluer la faisabilité du système à concevoir. Certaines modifications ou ajouts de mécanismes dans le système à concevoir qui peuvent survenir après la phase de conception entrainent un retour à la modélisation et à la vérification en prenant les précautions requises en termes de non-régression vis-à-vis des exigences considérées dans le cycle d'itération précédent.

- Un *langage de modélisation*. Il est composé d'un *cadre sémantique* [DOU 04] (concept et définition du langage) et d'une *notation visuelle* [DOU 04] ayant une syntaxe prédéfinie. Ici les langages utilisés sont UML 2.1 [UML] et SysML 1.0 [SysML], tous deux pourvus du même *cadre sémantique* (SysML étant vu comme une extension d'UML couvrant entre autre la formalisation des exigences) et de *notations visuelles* équivalentes. En complément à UML, nous empruntons à SysML les *diagrammes d'exigences* et les *matrices de traçabilité*.
- *Un processus de développement*. Ce processus est assimilé au vocable de méthodologie : il correspond à la description des successions d'*activités de travail* qui gouvernent la *production de « livrables »*. Le processus, que nous appelons par la suite méthodologie, sera décrit par le formalisme introduit dans la partie 1.2.

1.2. Langage de description d'une méthodologie

Un *processus de développement* méthodologique peut être vu comme un diagramme formé d'un ensemble d'*étapes méthodologiques* liées entre elles par des *transitions*, ayant comme entrée/sortie des *documents*, et des opérateurs de choix et parallélisme respectant la syntaxe des diagrammes d'activités UML.

1.2.1. Principe

Les organigrammes utilisés dans ce chapitre pour décrire les *processus de développement* étendent les *diagrammes d'activités* UML 2.1 [UML]. Les étapes méthodologiques sont présentées par des activités qui peuvent être raffinées en sous activités. Les activités des diagrammes présentés dans ce chapitre sont numérotées en conformité avec le Tab.3 de la section 2 de ce chapitre.

La Fig.9 montre une partie d'un processus de développement décrit dans le diagramme de description de méthodologie (DP pour *Development Process)* introduit dans cette section. Un diagramme d'activités débute obligatoirement avec un disque noir représentant le début de la première activité. Chaque étape est reliée par une flèche de transition montrant les relations de précédence entre activités (par exemple l'activité *sous étape méthodologique suivante* doit se produire après l'activité *sous étape méthodologique*). Les activités sont représentées par des rectangles aux coins arrondis. Une activité peut être décomposée en plusieurs sous-activités comme le montre l'activité *étape méthodologique* décomposée en deux sous-activités sur la Fig.9.

La *production* (création/modification) et la *consommation* (consultation) sont respectivement représentées par des flèches pleines (par opposition aux flèches de transitions entre activités) sortantes et entrantes de l'activité sur les côtés droit et gauche de la Fig.9. Cette dernière représente une étape méthodologique composée de deux sous-activités qui sont respectivement associées à la *consommation* et à la *production* de documents. La Fig.9 montre que l'on envisage un seul type

d'entrée (les documents consultables représentés par des flèches pleines). Il est possible de représenter deux types de sorties :

- les documents à créer représentés par des flèches sortantes pleines.
- les documents à modifier représentés par des flèches sortantes pointillées.

Un document sortant (produit ou modifié) est visible et donc consultable par toutes les autres activités. Par contre, la production est uniquement autorisée dans les activités qui possèdent une flèche sortante.

Au niveau des entrées et sorties, il est possible de distinguer le caractère obligatoire ou optionnel de la *production/consommation* de documents ; ceci apparaît également sur la Fig.9. La distinction entre les deux se fait par le texte :

- si le texte est écrit entre crochets, alors le document est optionnel (l'activité peut s'achever sans la *production/consommation* de ce document)
- si le texte n'est pas placé entre crochets, alors le document est obligatoire (l'activité ne peut s'achever sans la *production/consommation* de ce document)

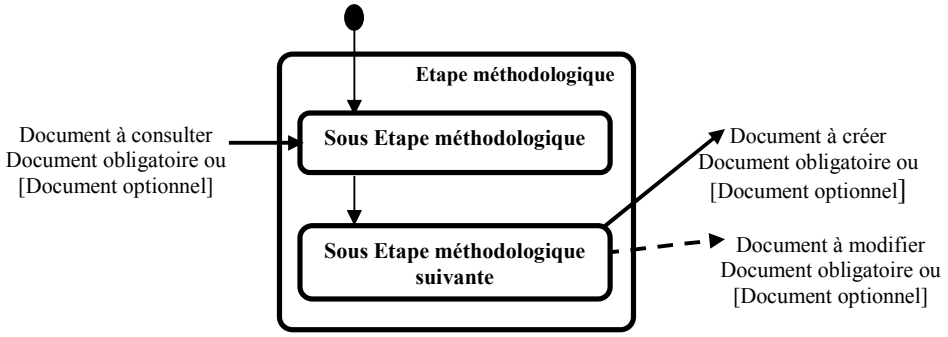

Fig.9. Langage de description de la méthodologie (DP)

1.2.2. Définition formelle

Ceci nous amène donc à définir de manière formelle le *processus de développement* d'une méthodologie.

Définition 1 : Processus de développement

DP = (MS, DOC, OP, transition, prod, modif, cons)

où

MS est un ensemble d'étapes méthodologiques ms pouvant être raffinées en un sous ensemble dp_ms

avec dp_ms = (MS, DOC, OP, transition, prod, modif, cons)

DOC est un ensemble de documents

OP est un ensemble d'opérateurs (choix, parallélisme, …) qui correspond aux opérateurs logiques des diagrammes d'activités UML

transition: MS → MS est une fonction de transition entre deux étapes méthodologiques

prod: MS → DOC est une fonction de production de bien livrable à partir d'une étape méthodologique

modif: MS→ DOC est une fonction de modification de bien livrable à partir d'une étape méthodologique

cons: DOC→ MS est une fonction de consommation de bien livrable à partir d'une étape méthodologique

1.2.3. Méta-modèle

Le diagramme de description de méthodologie (DP) que nous proposons étend donc les diagrammes d'activités (DA) UML 2.1 [UML]. Un méta-modèle décrit ce langage en UML par référence au méta-modèle des DA UML défini dans la norme UML [UML] (cf. Fig.10).

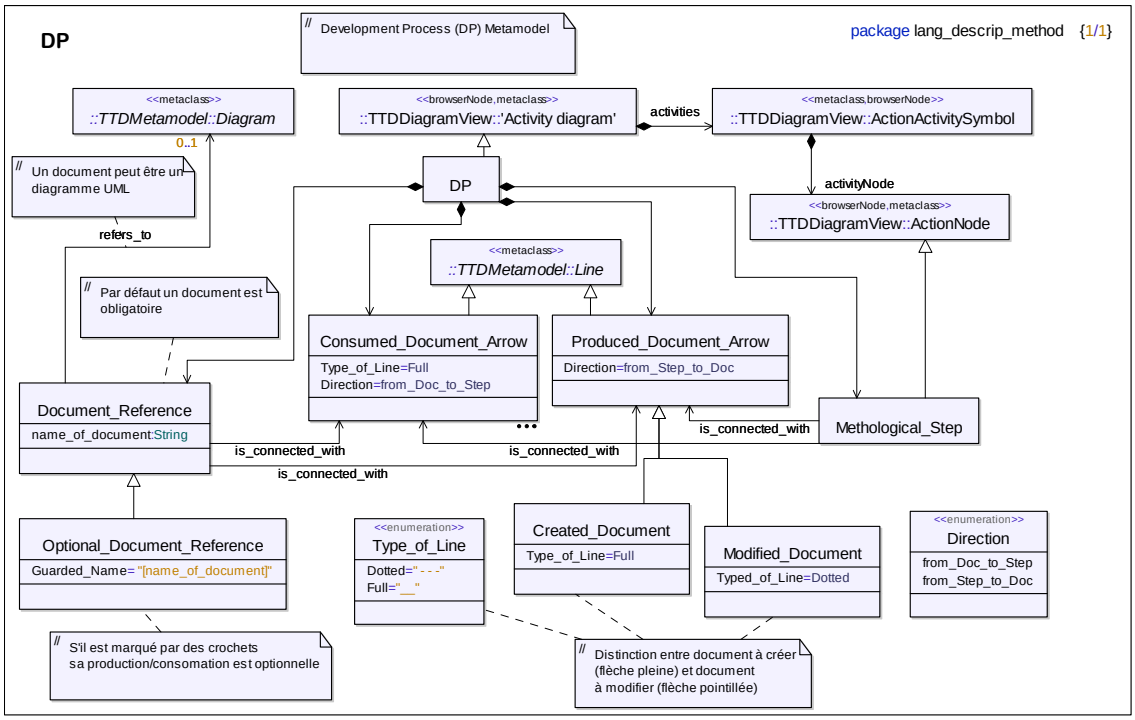

Fig.10. Méta-modèle du DP

Le diagramme de description de la méthodologie (DP) correspond à la classe *DP* qui hérite de la classe *ActivityDiagram*. DP hérite donc de toutes les caractéristiques d'un diagramme d'activités. Il est composé d'étapes méthodologiques (classe *Methodological_Step)*, de références de documents (classe *Document_Reference*) et de flèches reliant ces deux dernières (classes *Consumed_Document_Arrow* et *Produced_Document_Arrow*).

L'extension du DA en DP se fait par la spécialisation des nœuds représentant les actions (classe *ActionNode*). La classe *Methodological_Step* correspondant aux étapes méthodologiques hérite donc de la classe *ActionNode*. Les relations de *production/consommation* de documents (représentés par les classes *Consumed_Document_Arrow* et *Produced_Document_Arrow*) sont une spécialisation de la classe *Line* (provenant du méta-modèle UML représentant une relation qui peut être une flèche). On distingue deux types de relation entre documents et étapes méthodologiques :

- Une relation pour les documents consultés avec la classe *Consumed_Document_Arrow* représentant les flèches entrantes du DP. Le sens de la flèche va du document vers l'étape méthodologique, comme l'indique la valeur de l'attribut *Direction=from_Doc_to_Step* et les associations *is_connected_with* venant des classes *Document_Reference* et *Methodological_Step*.

- Une relation pour les documents produits avec la classe *Produced_Document_Arrow* représentant les flèches sortantes. Le sens de la flèche va de l'étape méthodologique vers le document, comme l'indique la valeur de l'attribut *Direction=from_Step_to_Doc* et les associations *is_connected_with* venant des classes *Document_Reference* et *Methodological_Step*. Cette relation se spécialise en deux classes *Created_Document* (cf. la flèche pleine que montre la valeur de l'attribut *Type_of_Line*) et *Modified_Document* (cf. la flèche pointillée que montre la valeur de l'attribut *Type_of_Line*).

Les documents associés à chaque étape méthodologique (en entrée ou en sortie) sont représentés par la classe *Document_Reference* contenant comme attribut le nom du diagramme (*name_of_document*). Cette classe peut faire référence à un diagramme UML ou SysML ; comme le montre l'association entre *refers_to* et la classe *Diagram*. Par défaut, le méta-modèle définit un document produit ou consommé comme obligatoire. On représente le caractère optionnel du document par la spécialisation de la classe *Document* en *Optional_Document_Reference* où le texte apparaît entre crochets, (cf. attribut *Guarded_Name*).

2. Guide de lecture

La méthodologie définie dans ce chapitre présente les relations entre les différentes étapes de travail et la livraison de diagrammes UML 2.1 [UML] et/ou SysML 1.1 [SysML]. Le Tab.3 confirme le caractère général de la méthodologie proposée et la possibilité de la spécialiser pour les protocoles.

<table>
<tr><th colspan="3">1
Vérification formelle d'exigences temporelles de modèles en contexte UML/SysML</th><th>2
Spécialisation dans les protocoles</th><th>Documents produits</th></tr>
<tr><td rowspan="5">1.1
Traitement des exigences</td><td colspan="2">1.1.0
Expression des besoins</td><td>2.1.0
Expression des besoins</td><td>Non spécifiques</td></tr>
<tr><td rowspan="2">1.1.1
Distinguer exigences et les raffiner en vue de les formaliser</td><td>Fonctionnelles</td><td rowspan="2">2.1.1
Exigences liées aux modes de connexion et à la qualité de service (QoS)</td><td rowspan="4">Base de données</td></tr>
<tr><td>Non-fonctionnelles temporelles</td></tr>
<tr><td rowspan="2">1.1.2
Formuler hypothèses de modélisation</td><td>Intangibles</td><td>Aspect données</td></tr>
<tr><td>Appelées à être levées : définition des aléas</td><td>Aspect contrôle
Qualité du médium :
- Délai
- Segmentation
- Fiabilité
Complexité de l'application :
- Scenarii unitaires
- Interaction de scénarii</td></tr>
<tr><td colspan="2">1.1.3
Spécification des exigences et formalisation des exigences non fonctionnelles temporelles selon classification de [ALU 93]</td><td>2.1.3
On reprend la classification de [ALU 93] sur laquelle on ajoute des règles pour les échanges de messages</td><td>Diagramme d'exigences</td></tr>
<tr><td>1.2
Analyse</td><td colspan="2">1.2.1
Périmètre du système : description de l'environnement et des acteurs</td><td>2.2.1
Application et médium (services sous jacents)</td><td>Diagramme de Cas d'utilisation</td></tr>
</table>

	1.2.2 Fonctionnalités : définition des cas d'utilisation	2.2.2 Définition des primitives de services	
	1.2.3 Scénarii : formalisation des scénarios	2.2.3 Description des PDUs échangés entre entités de protocoles correspondantes aux différentes primitives de services	Diagramme de séquences et Diagramme global d'interaction
1.3 Conception	1.3.1 Architecture : construction et association des entités	2.3.1 Architecture du protocole et pattern à trois couches inspiré du modèle OSI	Diagramme de Classes
	1.3.2 Comportement	2.3.2 Comportement des entités de protocoles à spécifier Application des dégradations sur le modèle abstrait du médium Construction d'un séquenceur de scénarii pour faire abstraction de l'application	Diagramme d'activités
1.4 Vérification des exigences non-fonctionnelles temporelles		2.4 Vérification des exigences temporelles liées au type de connexion et à la QoS	Matrice de traçabilité

Tab.3. Guide de lecture du chapitre III

Les méthodologies présentées dans les sections 3 et 4 sont, de plus, instanciées sur le profil UML temps réel TURTLE (présenté dans le chapitre II section 4.). La dernière colonne du tableau correspond au type de documents (qui peuvent être des diagrammes UML ou SysML) produits à chaque phase.

Le périmètre de la méthodologie étant fixé, nous allons nous concentrer sur le *processus de développement* proposé.

3. Vérification formelle d'exigences temporelles de modèles en contexte UML/SysML

La méthodologie présentée ici est dédiée à la vérification de modèle et ne couvre donc pas la phase d'implémentation et de déploiement. Nous supposons dans un premier temps que la responsabilité de la rédaction du cahier des charges incombe au client.

Cette section est structurée de la manière suivante. Le paragraphe 3.1 présente l'étape de *traitement des exigences* incluant les phases de *recueil des exigences*, de définition des *hypothèses de modélisation* et de *spécification des exigences*. La section 3.2 présente la phase d'*analyse*. Le paragraphe 3.3 présente la phase de *conception*. Enfin la partie 3.4 définit la dernière étape de *vérification des exigences*.

3.1. Traitement des exigences

La méthodologie démarre par la levée des ambiguïtés, manques et incohérences contenus dans les besoins exprimés par le client dans le cahier des charges : ceci correspond à la phase d'*expression des besoins* (partie 1.1.0 de la Fig.11). Dans la phase de *recueil d'exigences*, les besoins sont classifiés pour être transformés en exigences. Cette étape aboutit à la construction d'une base de données répertoriant les exigences à traiter (cf. section 3.1.1) où la distinction est faite entre exigences *fonctionnelles* qui seront le point de départ pour définir les *fonctionnalités* [HUL 04] (cf. chapitre II section 1) et exigences *non fonctionnelles temporelles* auxquelles le modèle sera confronté dans la

phase de *vérification* (cf. section 3.4). Cette étape est effectuée en parallèle avec la définition des *Hypothèse de modélisation* (étape 1.1.2 de la Fig.11, cf. section 3.1.2). Ensuite, la phase de *spécification des exigences* (étape 1.1.3 de la Fig.11) permet de procéder au raffinement des exigences pour obtenir des exigences précises et bien formulées en utilisant le *diagramme d'exigences* de SysML (cf. section 3.1.3). Après avoir traité les exigences, on peut spécifier les exigences *fonctionnelles* dans la phase d'*analyse* du modèle à concevoir.

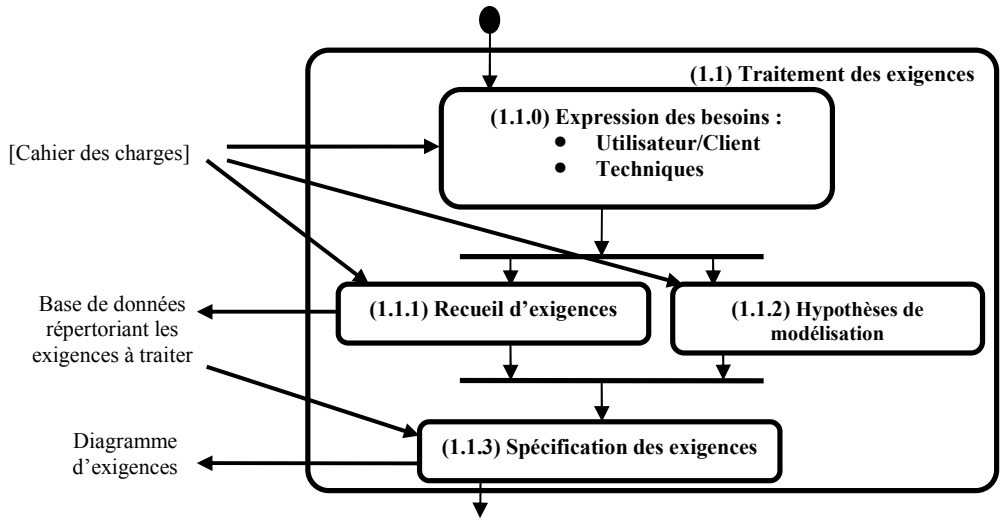

Fig.11. DP de la phase de traitement des exigences

Nous considérons que la phase d'*expression des besoins* (partie 1.1.0 de la Fig.11) a déjà été effectuée. L'utilisateur de la méthode, conjointement avec le client, a en premier lieu levé les ambiguïtés, les manques et les incohérences du cahier des charges.

3.1.1. Recueil des exigences

Dans un premier temps, les exigences sont inventoriées dans une *base de données répertoriant les exigences à traiter dans le système*. Comme mentionné dans le chapitre II section 1, nous distinguons [HUL 04] les exigences *fonctionnelles* (représentant les différentes fonctionnalités que devra satisfaire le système) et les exigences *non fonctionnelles* (représentant les différents objectifs que devra atteindre le système).

Le Tab.4 correspond à la base de données qui sera remplie durant cette étape; une exigence sera définie dans la base de données par :

- Un identifiant (champ *Requirement ID*) qui représente le numéro d'exigence, hiérarchisé en fonction du degré de raffinement de l'exigence (par exemple l'exigence 200 est d'un niveau plus abstrait que l'exigence 211 qui est raffinée à partir de l'exigence 210 elle-même découlant de l'exigence 200).
- Du texte (champ *Text*) décrivant de manière informelle l'exigence à traiter.
- Un niveau d'objectif (champs *Level of Goal*). Ceci correspond à la distinction entre niveau utilisateur et technique. On considère dans un premier temps les *exigences du client* (équivalent à la *branche fonctionnelle* de la méthode 2TUP présenté dans le

chapitre II section 1.2.2) à partir du *cahier des charges*. Dans un second temps, les exigences du client étant répertoriées, le recueil des *exigences techniques* (provenant des l'équipe de conception) pourra être formulé. Ceci découle en effet des exigences clients qui vont piloter un choix précis d'architecture et de mécanismes à mettre en œuvre.

- Le type d'exigence (champ *Kind*) qui distingue une exigence *fonctionnelle* d'une exigence *non-fonctionnelle.*
- Un domaine d'hypothèses (champ *Hypothesis Domain*) qui correspond au positionnement de l'exigence par rapport aux hypothèses de modélisation présentées en 3.1.2 et dont l'étape correspondante est effectuée parallèlement à cette étape (cf. DP de la Fig.11 section 3.1).

Requirement ID			Requirement Name	Requirement Text	Level of Goal	Kind	Hypothesis Domain
1	0	0	First Requirement	Informal text of Requirement	User Goal or Technical Goal	Functionnal or Non-Functionnal	Intangible or Released
2	0	0	Second Requirement				
2	1	0	Refined Requirement				
2	1	1	Refined Requirement				

Tab.4. Base de données répertoriant les exigences à traiter dans la phase de recueil d'exigences

Ensuite, pour faciliter la construction de la base de données, les exigences peuvent être formulées selon des *patterns d'exigence* définis en [HUL 04].

- Pour les exigences fonctionnelles, le pattern est du type suivant :

The <stakeholder type> (*shall or must*) be able to <capability>

où *stakeholder type* représente l'objet concerné par l'exigence, les verbes *shall/ must* représentent le degré de criticité de l'exigence (haut pour l'emploi de *must* et bas pour l'emploi de *shall*) et *capability* présente la fonctionnalité à construire.

- Pour les exigences non fonctionnelles temporelle, le pattern est le suivant :

The <stakeholder type> (*shall or must*) <function> (*within, after, in, every or between*) <performances> <units> of <events>

où *stakeholder type* représente le module concerné par l'exigence, les verbes *shall/ must* représentent le degré de criticité de l'exigence (haut pour l'emploi de *must* et bas pour l'emploi de *shall*), *function* présente la fonctionnalité considérée dans l'exigence ; les prépositions *within/ after/ in/ every/ between* seront équivalents respectivement aux types d'exigences temporelles à vérifier *Promptness/ Minimal Delay/ Punctuality/ Periodicity/ Interval Delay* [ALU 93] (Cf. chapitre II section 1), *performance* et *units* correspondent à la (aux) valeur(s) temporelle(s) de l'exigence et à l'unité temporelle employée et *event* correspond à l'(aux) événement(s) déclenchant l'exigence, que nous appellerons par la suite *point(s) d'observation*.

Le Tab.5 montre la structure de la base de données répertoriant les exigences non fonctionnelles temporelles, étendues après formulation des pattern d'exigences [HUL 04], (les « ,,, » représentent les champs montrés dans le Tab.4). Nous avons donc ajouté les champs suivants :

- Un niveau de risque (champ *Risk Level*) définit le niveau de criticité de l'exigence (haut ou bas).

- La liste des acteurs concernés (champ *Concerned Actors*) définit les acteurs[4] impliqués par l'exigence.
- Le type d'exigence temporelle (champ *Kind of Temporal Requirement*) formalise l'exigence selon la classification de [ALU 93] (cf. chapitre II section 1.).
- Les cas d'utilisation (champ *Use Case*) qui correspondent à l'exigence *non-fonctionnelle*.

Requirement ID			,,,	Risk Level	Concerned Actors	Kind of Temporal Requirement	Use Case
1	0	0	,,,	High or Low	List of Concerned Actors	Promptness	List of corresponding use cases
2	0	0	,,,			Minimal Delay	
2	1	0	,,,			Punctuality	
2	1	1	,,,			Periodicity	
2	1	2	,,,			Interval Delay	

Tab.5. Base de données répertoriant les exigences enrichies dans la phase de spécification d'exigences

3.1.2. Hypothèses de modélisation

Intégré à l'activité de modélisation système, le travail sur les exigences va de pair avec la définition d'hypothèses de modélisation. Là encore nous pouvons distinguer :

- Les hypothèses intangibles qui dépendent principalement du type d'outil de vérification envisagé.
- Les hypothèses appelées à être levées qui justifient des retours successifs dans la phase de traitement d'exigences.

Développé pour satisfaire les exigences et sous la contrainte des hypothèses, le modèle du système comprend la description du système proprement dit et de l'environnement.

Nous supposons ici que la description du seul système ne permet pas de construire un modèle exécutable dans un outil de vérification et qu'il est nécessaire de modéliser l'environnement du système. Cet environnement est formé d'un ou plusieurs « *aléas* ». Par exemple, dans le cadre d'une modélisation de couche de protocole, le médium sous-jacent à celle-ci est considéré comme un ensemble d'aléas dans la mesure où l'on peut démarrer la vérification avec un médium parfait pour intégrer ensuite les pertes, délais de transmissions et autres aléas qu'un tel médium peut introduire.

Sur la base des définitions précédentes, nous menons de front la construction d'une base de données d'exigences et la définition des hypothèses. Chaque hypothèse entraîne son lot de simplifications dans le modèle. Lorsqu'il s'agit d'hypothèses intangibles, l'on sait que les simplifications correspondantes s'appliqueront à toutes les versions du modèle qui pourront être élaborées. Par essence, les hypothèses appelées à être levées seront la base d'une construction incrémentale du modèle et leur granularité aura une incidence directe sur le nombre d'incréments de construction/vérification du modèle.

3.1.3. Spécification des exigences

Sur la base de la norme SysML [SysML] et de pratiques constatées dans le domaine de l'ingénierie des exigences [HUL 04] et des méthodes formelles, nous proposons le pattern de

[4] Dans le sens UML, c'est-à-dire extérieurs au système.

diagramme d'exigences de la Fig.12. Dans ce diagramme, l'exigence *Informal_Requirement* est une exigence informelle *non-fonctionnelle* qui est dérivée en une exigence formelle *Formal_Requirement*[5] (qui est obligatoirement *non-fonctionnelle*) ; ceci est indiqué par la relation de dépendance *« derive »*. Cette exigence formelle est décrite non pas par un texte informel mais par une équation de logique ou un langage visuel formel (cf. chapitre IV section 2.). Cette exigence est vérifiée (cf. la relation de dépendance *« verify »*) par des moyens de vérification (expliquées dans le chapitre V section 2.) qui peuvent être par exemple un *model checker* et/ou un *observateur* (les deux approches étant très souvent complémentaires). En SysML, on assimile ces dispositifs à des classes stéréotypées par *« test case »*. Enfin cette exigence formelle concerne une fonctionnalité particulière (représenté par une exigence de type *fonctionnelle*), comme le montre la relation de dépendance *« satisfy »*.

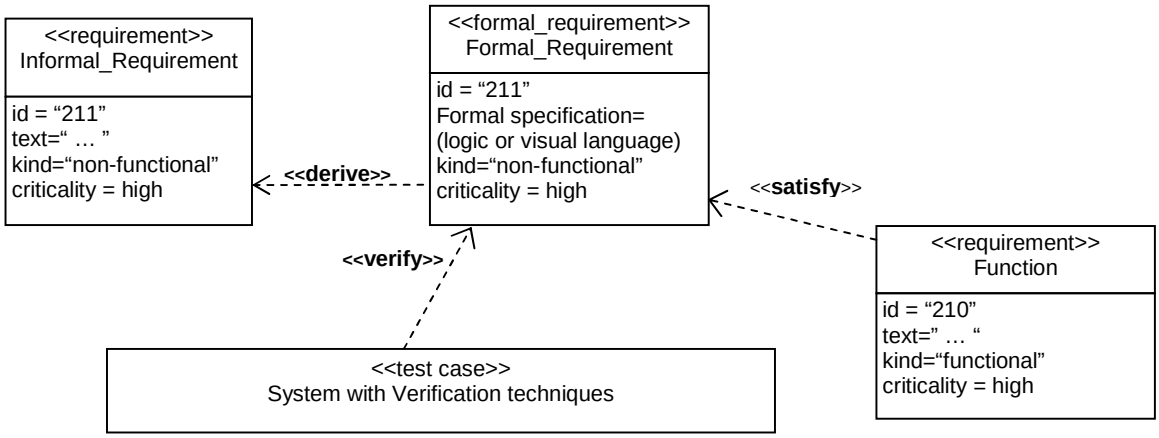

Fig.12. Pattern du diagramme d'exigence incluant une exigence formelle

<u>*Instanciation sur le profil TURTLE*</u>

Les *diagrammes d'exigences* de SysML [SysML] sont étendus pour séparer les exigences informelles des exigences formelles (conformément au pattern présenté en Fig.12) et en particulier des exigences non-fonctionnelles temporelles. Ces dernières sont définies non pas, par des formules de logique [HUT 04], mais par un langage visuel basé sur la représentation en chronogrammes et appelé *diagramme de description d'exigences temporelles* (*Timing Requirement Description Diagram = TRDD*). Cette instanciation est présentée en détail dans le chapitre IV sections 2. et 3.

Les exigences *non-fonctionnelles temporelles* sont spécifiées dans un diagramme d'exigences préfigurant la spécification formelle de ces dernières. Nous passons à la phase d'analyse qui va permettre de spécifier les exigences *fonctionnelles* puis de construire les premiers éléments du modèle formel orienté vérification.

3.2. Analyse

Cette étape correspond à la première description de l'aspect opérationnel du système. Nous nous concentrons donc sur la spécification des exigences fonctionnelles qui conduiront à l'établissement des *cas d'utilisation* et à leur description par des scénarii. La Fig.13 montre le

[5] Le stéréotype « formal_requirement » est une extension de SysML. Le méta-modèle introduisant ce stéréotype est présenté dans le chapitre IV section 2.1.

diagramme d'activités étendu de l'étape d'analyse complet. Il est composé de trois sous-activités qui sont : la *description de l'environnement*, la *construction des fonctionnalités* et la *définition des scénarii* décrivant dans un premier temps chaque *fonctionnalité* individuellement. Par la suite, ces scénarii élémentaires seront structurés dans des scénarii de plus haut-niveaux permettant de définir des scénarii d'utilisation du système incluant plusieurs fonctionnalités.

Dans un premier temps, il faut *définir le périmètre du système* et séparer celui-ci de ses *utilisateurs* et de son *environnement*. La base de données construite dans l'étape 1.1.2 (cf. § 3.1.1) sert de point de départ à la définition des acteurs qui rentrent en jeux dans les exigences *fonctionnelles*. Les acteurs extérieurs au système sont répartis en deux catégories : les acteurs qui utilisent le système et l'environnement (matériel/logiciel) sur lequel repose le système. La seconde étape correspond à la *définition du périmètre du système*.

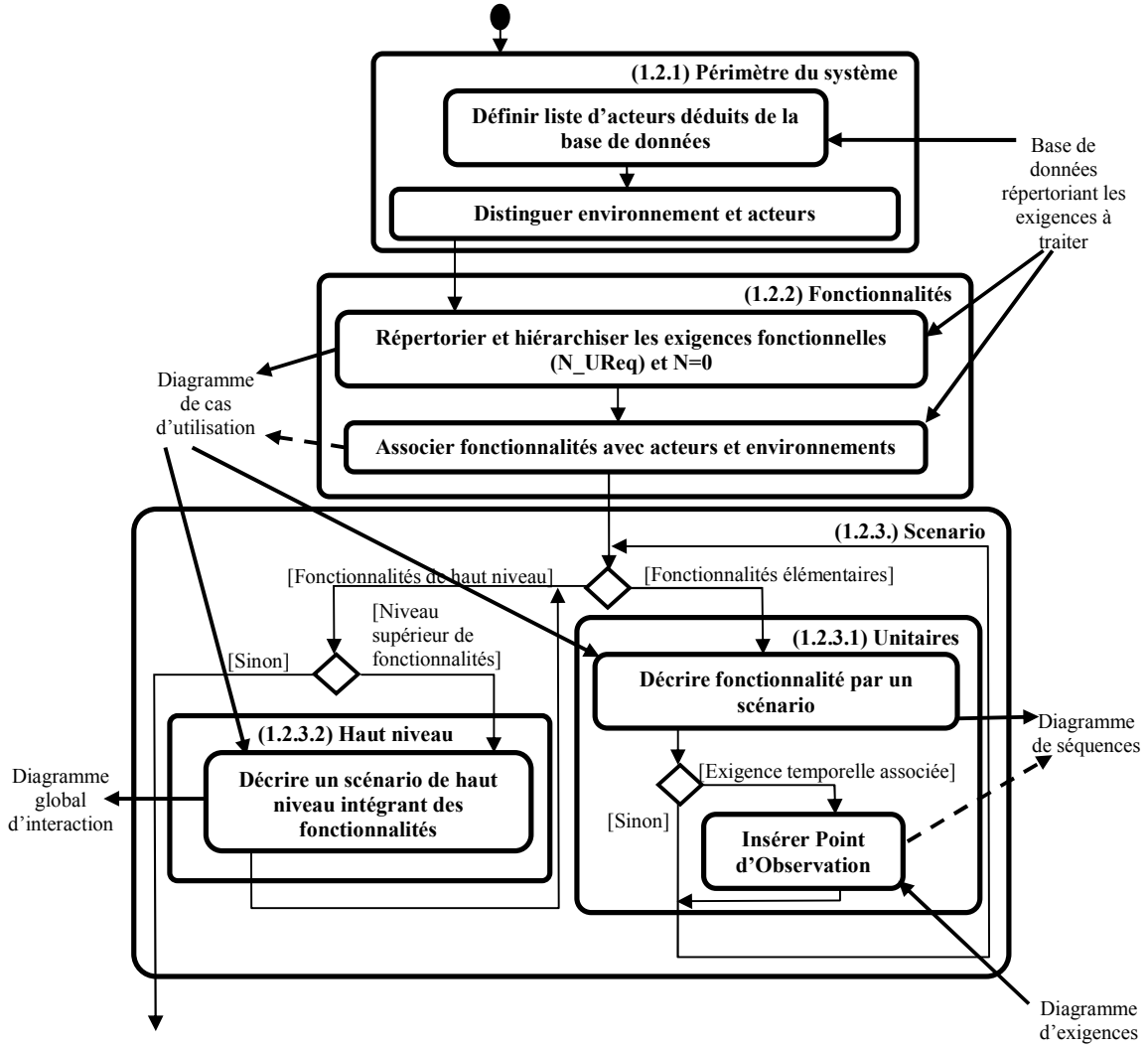

Fig.13. DP de l'étape d'analyse

Les fonctionnalités du système qui décrivent l'aspect opérationnel du système sont ensuite exprimées. Comme mentionné dans [HUL 04] [ROQ 04], les exigences *fonctionnelles* du système sont équivalentes aux *fonctionnalités* du système et donc en termes UML aux *cas d'utilisation*. Ceux-ci peuvent être reliés entre eux pour produire des inclusions ou des extensions entre fonctionnalités. Les stéréotypes de relation de dépendance entre *cas d'utilisation* sont respectivement « *include* » et « *extend* ». Les *cas d'utilisation* peuvent aussi se spécialiser et se généraliser par des *relations d'héritage*. Ces différentes *fonctionnalités* sont ensuite associées aux *acteurs* adéquats. Le *diagramme de cas d'utilisation* s'enrichit, comme le montre la flèche en pointillé sortant de l'activité *Associer fonctionnalités avec acteur et environnement* sur la Fig.13).

Les fonctionnalités du système ayant été définies (activité (1.2.2) dans un *diagramme de cas d'utilisation*, nous passons à l'étape de *définition des scénarii*. Chaque *fonctionnalité* est, dans un premier temps, décrite par un scénario élémentaire où l'on fait intervenir les différentes entités et acteurs concernés par cette fonctionnalité (activité 1.2.3.1). Ceci est fait en construisant un *diagramme de séquences*. On peut alors relier les *fonctionnalités* aux exigences *non-fonctionnelles* produites en section 3.1.1, en insérant des *points d'observations* présentés dans la section 3.1.1 de ce chapitre (étape *Insérer points d'observations* de la Fig.13). Ces *points d'observations* correspondent aux événements définissant l'exigence (cf. § 3.1.1) et leur placement est un point très délicat. En effet, si ces points ne sont pas correctement placés alors l'exigence ne sera pas correctement vérifiée. Le placement correct des points d'observation fait l'objet de la section 3 du chapitre V.

Une fois construits, les scénarii élémentaires sont structurés par des scénarii de plus haut niveau. Ces derniers sont composés de fonctionnalités élémentaires incluses dans la fonctionnalité décrite par un *diagramme global d'interaction* (activité 1.2.3.2.).

<u>*Instanciation sur le profil TURTLE*</u>

La Fig.14 illustre le placement des points d'observation dans un *diagramme de séquences* générique non tributaire d'un processus spécifique. Les *points d'observations* sont représentés par des activités internes étiquetées selon les choix de l'utilisateur, par exemple, par *PO_Start* pour la première occurrence décrivant l'exigence et *PO_End* pour la seconde occurrence décrivant la fin de l'exigence. Il est très important de placer les *points d'observation* immédiatement après les messages qui vont conduire d'une part à l'exécution du processus (*Start_Process*) et d'autre part à la fin du processus (*End_Process*). Les règles définies dans le chapitre V section 3 (éléments de preuves), montrent que si les points d'observations étaient placés avant l'événement déclenchant une exigence temporelle alors l'observateur délivrerait un mauvais diagnostic. Notons ici que les *points d'observation* peuvent être placés dans des objets différents.

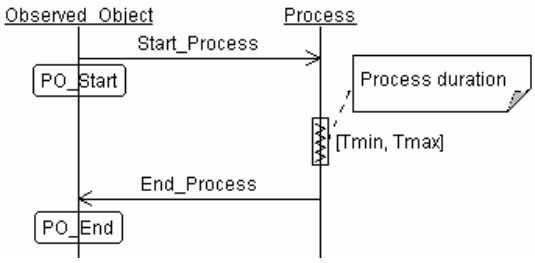

Fig.14. Placement des points d'observation dans un diagramme de séquences TURTLE

A partir de ces informations, une ébauche de modèle peut être construite du point de vue architectural (en représentant les interactions définies dans le *diagramme de séquences* entre *objets du système*, les *acteurs* et l'*environnement*) et comportemental (déduits de la description des scénarii sous forme de *diagrammes de séquences* et de *diagrammes globaux d'interactions*).

3.3. Conception

Cette activité prépare l'implantation pour structurer le système en le décomposant en sous-systèmes qui correspondent aux différentes entités du système et en définissant le comportement interne de chaque classe. Faisant suite à l'analyse, la conception est décomposée en deux étapes successives (cf. Fig.15).

La première étape est la spécification au niveau architectural du modèle (activité 2.3.1 de la Fig.15) par un *diagramme de classes* et *diagramme de structure composite* permettant de décrire l'architecture d'un module complexe composé de modules internes. Ceux-ci sont utilisés pour décomposer le système en sous-systèmes définissant les différentes classes/objets. Quand tous les modules sont définis, le concepteur les associe entre eux en fonctions des scénarii définis en phase d'analyse.

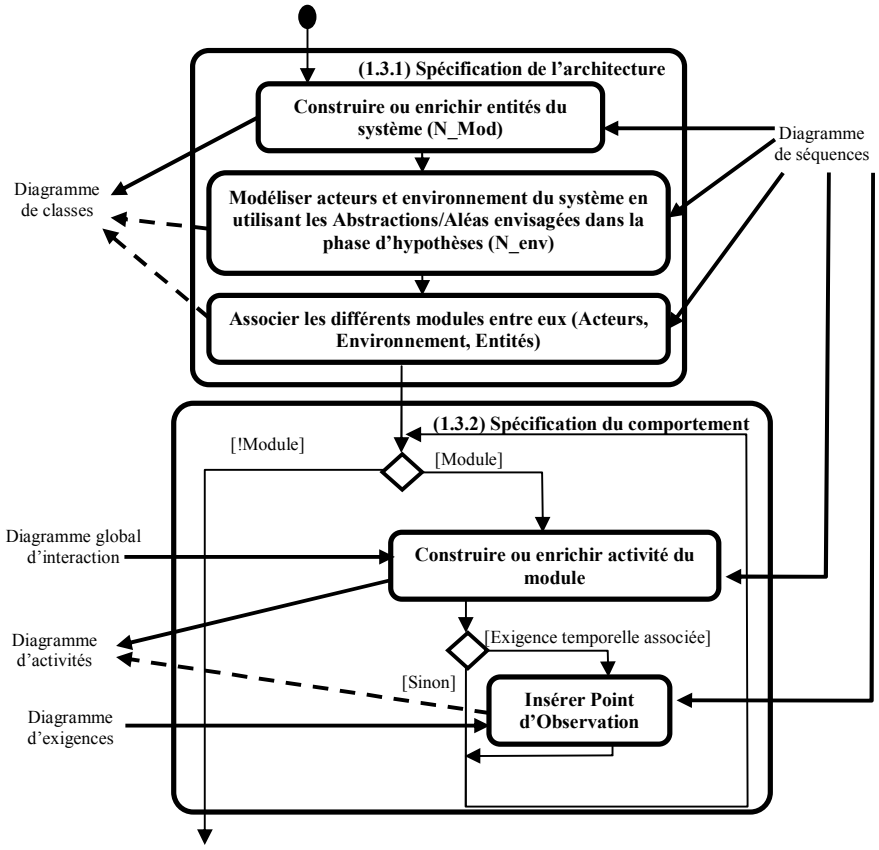

Fig.15. DP de l'étape de conception

Ensuite, le comportement interne de chaque classe est construit avec un *diagramme d'activités* (activité 2.3.2 de la Fig.15), à partir des scénarii définis dans l'analyse (dans un premier temps les scénarii élémentaires décrits dans les *diagrammes de séquences* des fonctionnalités puis les scénarii de plus haut niveau définis dans les *diagrammes globaux d'interaction*). Pour les classes qui interviennent dans une exigence temporelle, il faut ajouter des points d'observations, décrits dans la section précédente, dans le *diagramme d'activités* en respectant le placement de ces points décrits dans des diagrammes d'analyse (*diagrammes de séquences et diagrammes globaux d'interactions*).

Instanciation sur le profil UML TURTLE :

Le placement des points d'observation est illustré sur la Fig.16. Tout comme sur la Fig.14, les *points d'observations* seront placés après les événements à observer pour respecter les règles de la section 3 du chapitre V. A nouveau, les points d'observation peuvent être placés dans des objets différents.

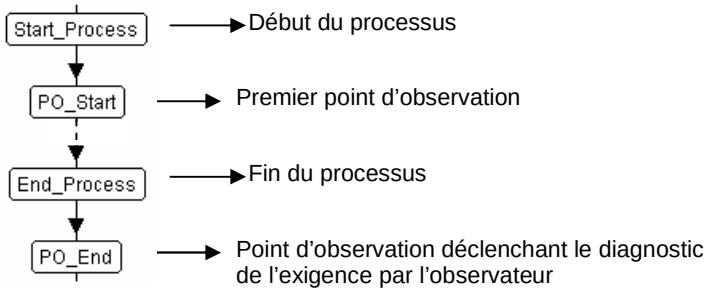

Fig.16. Placement des points d'observations dans la classe observée

En amorce de la *vérification formelle d'exigences*, le modèle de *conception* ainsi spécifié est ensuite traduit dans le langage formel associé au profil UML considéré.

3.4. Vérification formelle des exigences

La Fig.17 présente le diagramme d'activités étendu (DP) de l'étape de vérification des exigences. Nous limitons notre approche aux techniques de modélisation pour lesquelles il existe des outils d'analyse d'accessibilité. A partir de la spécification formelle issue du modèle en UML on construit un *graphe d'accessibilité*. Ceci s'applique par exemple aux *systèmes de transitions étiquetés* LTS [ALU 94] ou aux *réseaux de Petri temporels* [BER 04]. Deux problèmes principaux se posent alors : d'une part, peut-on construire le graphe d'accessibilité ? D'autre part, lorsque ce *graphe d'accessibilité* est construit, comment l'exploiter correctement pour vérifier les exigences du système ? La non-construction du graphe d'accessibilité est souvent due à la définition de « mauvaises » abstractions : un retour dans la phase de *définition des hypothèses de modélisation* s'impose donc (cf. Fig.17). Pour construire dans un premier temps un modèle formel qui puisse être exploitable, il faut considérer des hypothèses très simples (voire triviales) dans un premier temps. Au fur et à mesure des cycles d'itérations on construit un modèle de plus en plus complet jusqu'à obtenir une représentation du système aussi proche que possible de la réalité pour pouvoir vérifier les exigences.

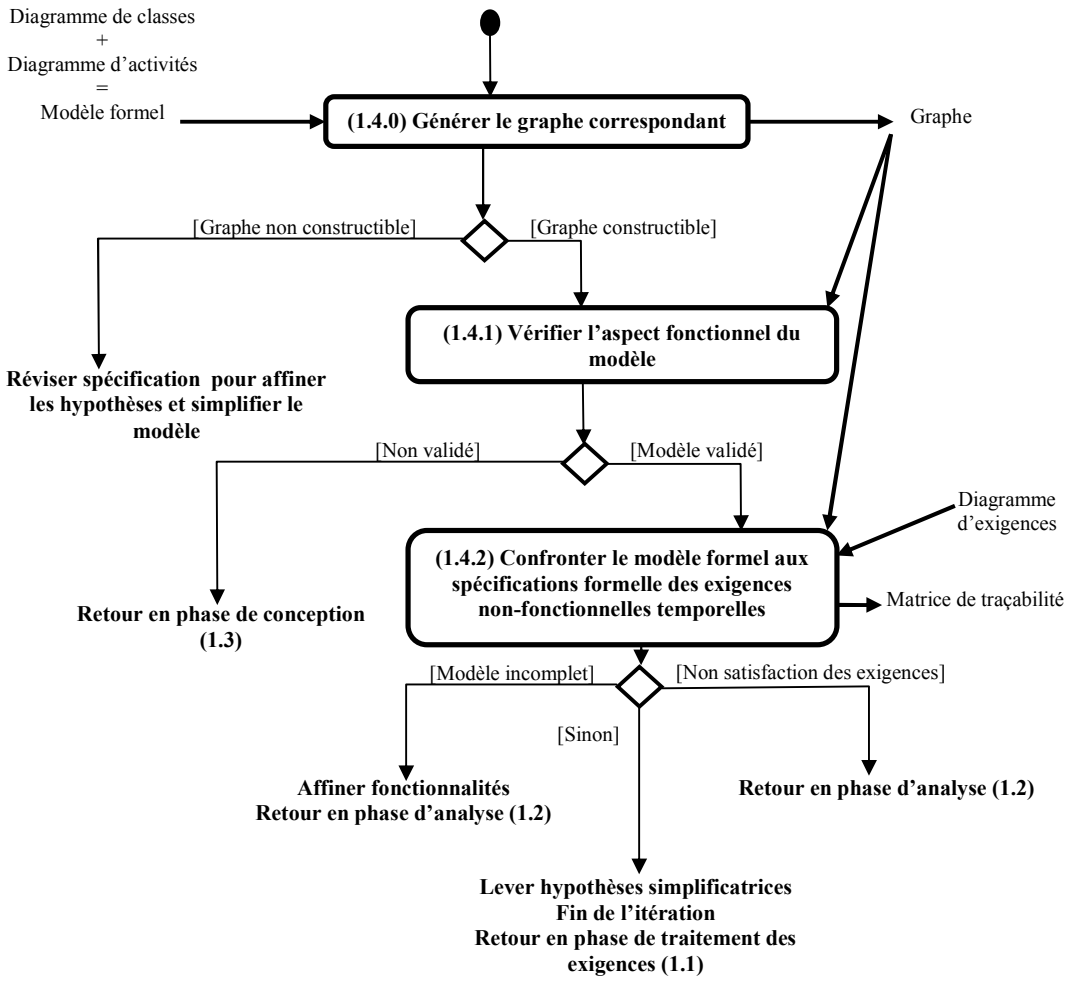

Fig.17. DP de l'étape de vérification

Nous nous plaçons ensuite dans l'hypothèse favorable où le graphe d'accessibilité est construit[6] dans un temps acceptable. Il est possible d'exploiter ce graphe en utilisant différentes techniques de vérification ; soit la vérification par contrôle de modèle (*model checking*), soit la construction d'observateurs dédiés à la vérification des exigences. Ces techniques sont décrites plus amplement dans le chapitre V section 2. On peut alors envisager la vérification des exigences en deux étapes successives :

- Dans un premier temps, la vérification est envisagée sur l'aspect *fonctionnel* du système (activité *Vérifier l'aspect fonctionnel du modèle* de la Fig.17) à savoir l'absence de blocage (« deadlock »), l'absence de fonctionnement cyclique infini (« livelock »), la possibilité de revenir à l'état initial, ou encore les équivalences entre les diagrammes d'*analyse* et de *conception*. La non-vérification de l'aspect *fonctionnel*

[6] Nous nous plaçons dans l'hypothèse ou la vérification s'effectue sur le graphe d'accessibilité complet. Pour l'utilisation de *model-checker* à la volée [CLA 99] il suffit de commencer le processus de développement à partir de l'étape 1.4.1.

entraine un retour dans la phase de *conception* comme le montre la Fig.17 (les fonctionnalités du système ayant été mal spécifiées dans les diagrammes de *conception*).

- Dans un second temps, la vérification de l'aspect *non-fonctionnel* du système repose sur la confrontation du modèle formel aux spécifications formelles des exigences (activité *Confronter le modèle formel aux spécifications formelle des exigences non-fonctionnelles temporelles* de la Fig.17). Ces dernières, pour pouvoir être vérifiées, doivent être décrites formellement (par formule de logique ou par définition d'observateurs). Le résultat de la vérification apparaît dans une *matrice de traçabilité* assimilable à une base de données qui lie chaque exigence formalisée à un résultat de la vérification formelle. Si ces exigences ne sont pas validées, un retour dans la phase d'*analyse* s'impose, les mécanismes mis en œuvre ne permettant pas de satisfaire les exigences *non-fonctionnelles* du système.

La vérification du modèle relève d'un processus incrémental qui démarre en traitant pas à pas les *cas d'utilisation*. On se limite dans la première itération à une situation nominale où l'environnement est supposé parfait. Si le modèle est incomplet, il devra être enrichi au niveau de l'*analyse* (par raffinement des *cas d'utilisation*). Notre approche s'inscrit dans le contexte d'une modélisation AGILE (cf. chapitre II. section 2.1). Si le modèle est complet et validé, l'itération est finie. Une présentation au client est faite alors pour lui montrer le modèle et lui rendre compte de la vérification des exigences dans un mode de fonctionnement sans *aléa* (cf. section 3.1.2). On rend ensuite l'environnement plus réaliste en levant les hypothèses simplificatrices et en introduisant pas à pas des fonctionnements dégradés, sans omettre des tests de non régression (par comparaison entre le modèle de l'itération n-1 et le modèle de l'itération n) vis-à-vis du modèle qui fonctionnait correctement au préalable et on recommence une itération si les exigences sont satisfaites. Toutes ces alternatives sont décrites dans la Fig.17 par les différents choix situés après les étapes 1.4.1 et 1.4.2.

Instanciation sur le profil TURTLE

Nous suivons donc le DP de Fig.17. La vérification des exigences se distingue en deux étapes :

- La première étape repose sur le principe général de vérification d'un modèle TURTLE décrit dans la méthodologie de [APV 06] présentée dans le chapitre II section 4.2. Si le graphe d'accessibilité est constructible, le modèle est vérifié du point de vue *fonctionnel*. On cherche à voir si le système est borné temporellement et s'il n'y a pas de situation d'interblocage ou de comportement cyclique infini non souhaité. Enfin, on mesure les équivalences en termes de graphes (par exemple par minimisation des actions non mentionnées dans la phase d'analyse) entre les diagrammes d'*analyse* et de *conception*. La chaîne d'outils du profil TURTLE (cf. chapitre II section 4) permet de vérifier formellement les diagrammes d'analyse (*diagrammes de séquence et diagrammes globaux d'interactions*), ce qui nous amène à établir des équivalences entre les diagrammes *d'analyse* et de *conception*. Pour chaque graphe obtenu après génération du modèle UML dans le langage formel RT-LOTOS [COU 00] (cf. chapitre II section 4.1.1) on effectue des minimisations basées d'une part sur des relations d'équivalences (par exemple avec l'équivalence observationnelle de Milner [MIL 89]) et d'autre part sur un ensemble d'événements qui doivent être préservés par la

minimisation (en l'occurrence des éléments communs aux deux modèles). Il résulte en sortie des *automates quotients*, qui sont ensuite comparés par le biais d'une *bisimulation* [MIL 89]. Si les deux automates quotients sont équivalents alors le modèle de conception correspond bien au modèle d'analyse : les exigences *fonctionnelles* définies dans la phase d'analyse correspondent au modèle de conception.

- La seconde étape correspond aux contributions présentées dans le chapitre V de ce mémoire. Les exigences *non-fonctionnelles temporelles* sont vérifiées dans le modèle de conception à l'aide d'observateurs déduits à partir de l'étape de *spécification des exigences* (cf. chapitre IV). Le concept d'*observateurs* et la vérification formelle des observateurs automatiquement générés à partir de l'étape de *spécification des exigences* sont expliqués en détail dans le chapitre V. Le résultat de la minimisation, est un automate quotient dans lequel nous pouvons – entre autres informations – rechercher les étiquettes de violation d'exigences, associées aux exigences temporelles. L'introduction d'étiquettes de violation d'exigences dans les automates quotients est liée au pilotage de la vérification par les observateurs. Ceci conduit à la production d'une matrice de traçabilité (cf. Fig.18) contenant les résultats de la vérification formelle guidée par les observateurs, en l'occurrence la présence ou non de l'étiquette de violation de l'exigence qui conduira à la présence respective de KO ou OK dans la colonne *satisfiability*. La non-satisfaction de ces exigences produira donc un retour dans la phase d'*analyse* comme l'indique la Fig.17. Si celles-ci sont satisfaites alors nous passons à une nouvelle itération en prenant en compte de nouvelles fonctionnalités ou de nouvelles dégradations de l'*environnement*.

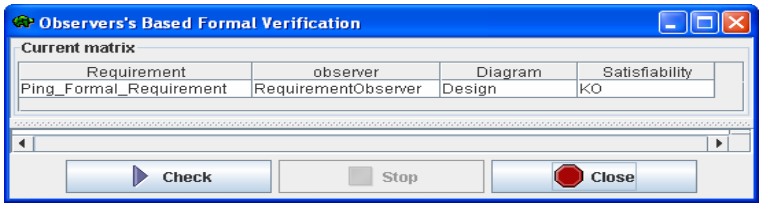

Fig.18. Matrice de traçabilité de TURTLE pour la vérification d'exigences non-fonctionnelles

4. Spécialisation dans le domaine des protocoles

Cette section aborde la spécialisation de la méthodologie précédemment exposée vers la conception de systèmes temps réel et distribués en définissant un ensemble de règles pour les différentes étapes méthodologiques proposées dans le Tab.3 de la section 2 et les DP de la section 3. En premier lieu vient le traitement des exigences liées au concept de Qualité de Service et au type de protocoles envisagés.

4.1. Les règles de traitement des exigences

4.1.1. Exigences liées au type de protocoles envisagés et à la qualité de service

Cette étape dépend principalement du type de service envisagé, et des exigences de *qualité de service* (Cf. chapitre II section 1).

E 1. Si le protocole à concevoir est en mode non-connecté, alors le protocole devra satisfaire une exigence fonctionnelle de transfert de données.

E 2. Si le protocole à concevoir est en mode connecté, alors le protocole devra satisfaire le trio d'exigences fonctionnelles (de phases) présentées ici dans leur ordre d'apparition dans le fonctionnement du protocole :
- *E 2.1. L'établissement de connexion.*
- *E 2.2. Le transfert de données.*
- *E 2.3. La libération de connexion.*

E 3. Si le protocole à concevoir est en mode connecté, alors il devra satisfaire trois exigences non-fonctionnelles qui sont à respecter dans les toutes les phases du protocole :
- *E 3.1. Pas de perte de données et de messages.*
- *E 3.2. Pas de déséquencement de messages.*
- *E 3.3. Pas de duplication de messages.*

E 4. On distingue deux types d'exigences temporelles liée à la qualité de service (QoS) dans les systèmes temps réel [STE 93] sur lesquelles nous pouvons appliquer les différentes classes d'exigences temporelles (promptness, minimal delay, punctuality, periodicity, interval delay) définies dans le chapitre II section 1. Nous identifions alors :
- *E 4.1. La variation de délai entre la réception de deux paquets (Jitter) ; ceci correspond à l'intervalle temporel de réception de deux paquets par une même machine de protocole.*
- *E 4.2. Le délai de bout en bout pour la réception d'un paquet (End to end delay) ; ceci est équivalent au délai de transmission d'un paquet entre deux machines de protocoles.*

Ce sont là quelques exigences types à satisfaire en ingénierie des protocoles. Leur formalisation correspond au pattern de *diagramme d'exigences* de la Fig.12 présenté en 3.1.3. Les exigences *non-fonctionnelles temporelles* doivent être vérifiées au niveau de la couche *application* utilisatrice de la couche de protocole en cours de conception.

4.1.2. Hypothèses de modélisation

La section 3.1.2 a identifié deux types d'hypothèses à prendre en compte dans la vérification formelle d'exigences :

- Les *hypothèses intangibles* dépendent généralement du type d'outils de vérification et de ce que l'on prétend vérifier. En termes de vérification formelle, les hypothèses intangibles concernent l'*aspect données*. Par exemple, il est souvent impossible de représenter des équations de contrôle de congestion du type TFRC [TFRC] (faisant intervenir des flottants) ; cet aspect dépendant des données est très difficilement représentable dans un modèle formel. Une « bonne » méthode d'abstraction consiste à distinguer les différents modes de fonctionnement liés à cette équation et à discrétiser cette équation en fonction de ces modes de fonctionnement (cf. *HI 1*).

HI 1. Une équation faisant intervenir des flottants doit être discrétisée en fonction des modes de fonctionnement déduits de l'équation.

- Les *hypothèses appelées à être levées* déterminent le nombre d'itérations à prendre en compte dans la vérification. Quel que soit le type de protocole, la première hypothèse appelée à être levée a trait à la *segmentation du médium* (service sous-jacent voir *HL 1*). On considère uniquement les PDU échangées entre entités de protocoles dans un premier temps, puis on les encapsule par les primitives de services du médium.
  Les autres hypothèses appelées à être levées sont définies en fonction du type de service sous-jacent retenu.
  - Si le service est de type *fiable*, alors l'hypothèse *HL 2* sera prise en compte durant toutes les itérations pour la conception du modèle dédié à la vérification. Le *délai de transmission* des messages peut être vu comme un délai fixe avec une latence négligeable qui correspond au fait qu'un message prend le même chemin durant toute la session protocolaire (voir *HL 3*).
  - Si le service sous-jacent est de type *non-fiable*, alors l'hypothèse *HL 2* sera levée. Il faudra prendre en compte successivement comme *Aléas* : le *déni de service* (les *pertes*), le *déséquencement* et la *duplication* de messages (comme le montre *HL.4*). Le *délai de transmission* devra ensuite tenir compte du fait qu'un service non fiable transmet des messages sur des chemins différents durant une session. Ce délai sera donc représenté par un intervalle temporel compris entre les temps de transmissions sur chemin le court et sur chemin le plus long.

HL 1. Le médium est capable de transmettre des données de taille arbitraire. Ce qui veut dire que l'on ne considère pas la segmentation de données.

HL 2. Le médium est considéré comme fiable (aucune perte, aucune duplication, pas de déséquencement).

HL 3. Si le médium est considéré comme fiable, le temps de transmission peut être assimilé à un délai (ou un intervalle temporel de très faible latence).

HL 4. Si le médium est considéré comme non-fiable, trois aléas seront à considérer durant les itérations :
HL 4.1. Le déni de service.
HL 4.2. La duplication.
HL 4.3. Le déséquencement.

HL 5. Si le médium est considéré comme non-fiable, le temps de transmission est modélisé par un intervalle temporel (entre les temps de transmission du plus court chemin et du plus long chemin)

Les itérations successives pour la construction du modèle respecteront le cycle décrit en [COU 92] (voir chapitre II section 5). La conception du protocole est caractérisée par les étapes suivantes :

- La première étape stipule que les propriétés *HL 1* et *HL 2* doivent être satisfaites par le médium qui sera donc vu comme *fiable* et *sans segmentation de données*. Les temps de transmission sont modélisés dans le *médium*.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

- Dans la deuxième étape, le médium satisfait seulement la propriété *HL 2*. Cette étape de conception correspond donc à l'intégration des mécanismes de *segmentation/réassemblements* des données.
- Si le médium est *non-fiable*, la dernière étape correspond à l'introduction des mécanismes de reconnexion. Le médium satisfait *HL 5* et ne satisfait plus les propriétés *HL 1* et *HL 2*. Ceci correspond aux itérations suivantes :
    - *HL 4.1* est satisfaite.
    - *HL 4.2 puis HL 4.1 et HL 4.2* sont satisfaites.
    - *HL 4.3 puis HL 4.1, HL 4.2 et HL 4.3* sont satisfaites.
- *Si le médium est fiable,* le médium satisfait *HL 3* et ne satisfait pas *HL 1*.

4.1.3. Les règles liées à la phase d'analyse

On se concentre dans un premier temps sur la description de l'environnement et ensuite sur la spécification des exigences fonctionnelles qui conduiront à l'établissement des *phases* du protocole.

Comme le montre la règle *A 1*, l'environnement considéré dans la vérification de protocoles est composé de (cf. chapitre II section 5.) :

- L'abstraction du service (par exemple la couche application) qui utilise le protocole et qui représente les acteurs.
- Le médium utilisé par le protocole qui représente l'environnement du protocole.

A 1. Le périmètre d'un protocole est composé d'une abstraction du service qui utilise les services du protocole et d'un médium qui est utilisé par le protocole à concevoir.

Le périmètre du système étant construit, la définition des exigences fonctionnelles conduit donc à l'établissement des *cas d'utilisation* qui sont considérés dans un protocole comme les *phases* et des *primitives de services*. Les différentes fonctionnalités de haut niveau correspondent donc aux différentes phases du protocole, soit :

A 2. Un protocole en mode non-connecté est défini au minimum par un cas d'utilisation : le transfert de données.

A 3. Un protocole en mode connecté est défini au moins par trois cas d'utilisation : l'établissement de connexion, le transfert de données et la libération de connexion [EXP 04].

La Fig.19 présente le *diagramme de cas d'utilisation* générique pour les différents modes de protocole.

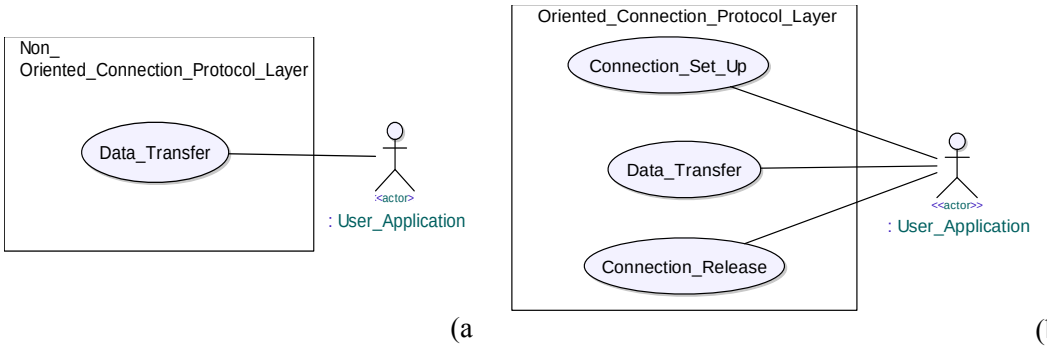

Fig.19. Diagrammes de cas d'utilisation générique pour un protocole en mode connecté et non-connecté

Après avoir défini les différents *cas d'utilisation* (*phases* et *primitives de services*) du protocole, on peut donc définir les scénarii de fonctionnement des phases du protocole en construisant des *diagrammes de séquences* et des *diagrammes globaux d'interaction*. En s'inspirant de la méthodologie Estelle* [COU 91], nous spécifions le protocole (N) en utilisant les deux premiers niveaux d'abstraction :

- La *spécification abstraite du service (N)*, ayant pour objet de représenter l'ordonnancement temporel des primitives de services, telles qu'elles sont visibles par les utilisateurs de la couche (N). Ceci correspond à la première étape de la Fig.20.
- La *spécification abstraite du protocole (N)*, ayant pour objet de raffiner la spécification précédente en introduisant les fonctionnalités des entités de protocoles de la couche (N), ainsi que le mappage des (N) PDUs échangés entre les entités sur les primitives de services de la couche (N-1). Cette phase correspond à la seconde étape de la Fig.20.

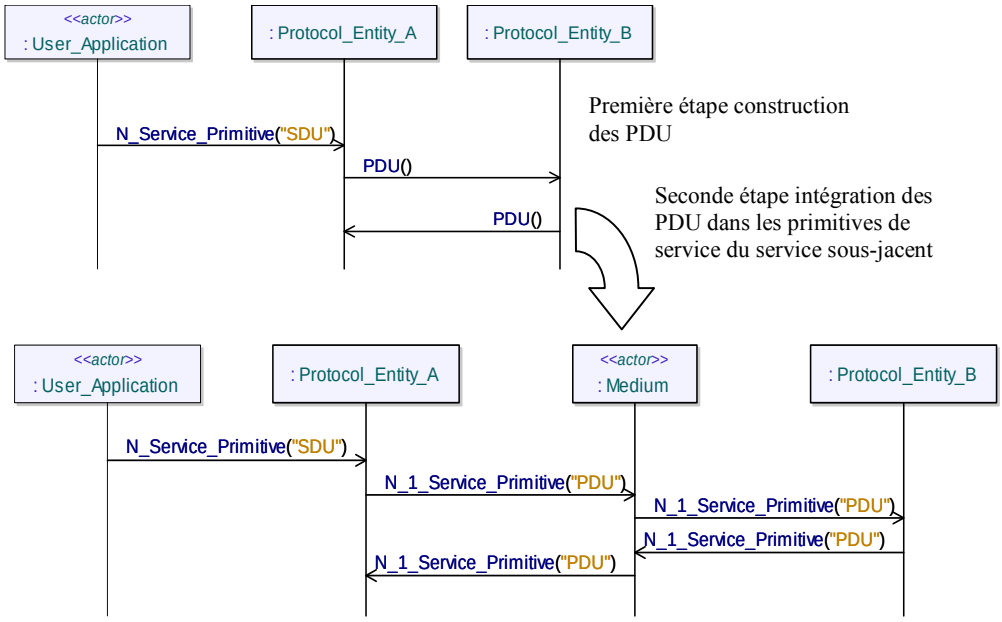

Fig.20. Diagramme de séquences générique pour un protocole

Instanciation sur le profil TURTLE

La Fig.21 montre un placement des points d'observation caractéristique dans le *diagramme de séquences* générique de la Fig.20 pour vérifier des exigences temporelles liées à durée d'un service de la couche de protocole. Les points d'observations sont représentés par des activités internes étiquetées selon les choix de l'utilisateur, par exemple, par *PO_Start* pour la première occurrence marquant le début de l'exigence et *PO_End* pour la seconde occurrence marquant la fin de l'exigence. Il est très important de placer les *points d'observation* immédiatement après les messages qui vont conduire d'une part à l'exécution du service et d'autre part à la fin du service (cf. chapitre V section 3). Pour vérifier des exigences liées à la durée d'un service N, il faut se placer au dessus du service qui va appeler les différents services du protocole et donc ici dans la couche application. Ceci permet de ne pas modifier la spécification des entités de protocole à concevoir : on modifie seulement le service qui utilise les services fournis par le protocole à concevoir ; ce service est à l'origine une abstraction et donc facilement modifiable. Les points d'observation peuvent être placés dans des objets différents, par exemple deux applications différentes.

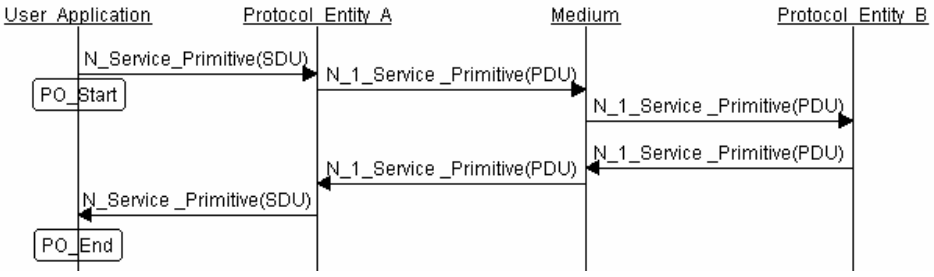

Fig.21. Placement des points d'observation dans un diagramme de séquence TURTLE

L'exigence de *qualité de service* (QoS) de *« jitter »* d'un service N correspond à la variation de délai entre la réception de deux paquets. La Fig.22 montre le placement des points d'observation pour pouvoir vérifier ce type d'exigence (en ayant préalablement spécifié celle-ci par un diagramme de description d'exigences temporelles montré dans le chapitre IV), dans un diagramme de séquences (DS) TURTLE. Les points d'observation (*PO_Start* et *PO_End*) sont donc placés dans la classe de l'application qui reçoit les messages du service N et après réception de ces derniers.

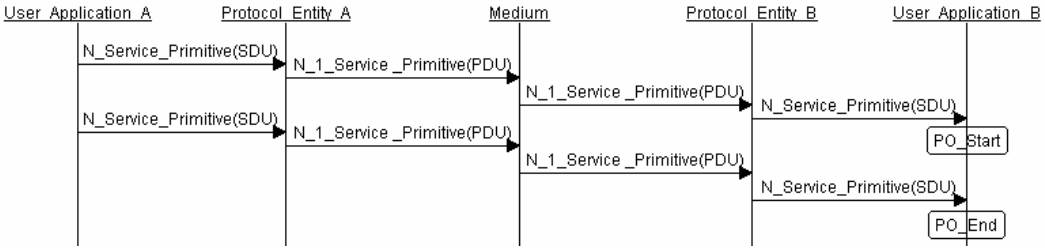

Fig.22. Placement des points d'observation dans un DS pour vérifier une exigence de QoS de *« jitter »*

L'exigence de QoS de *« end to end delay »* d'un service N correspond au délai de bout en bout de transmission d'un paquet sur le réseau entre deux applications utilisant le service N. La Fig.23 montre le placement des points d'observation pour pouvoir vérifier ce type d'exigence (en ayant préalablement spécifié celle-ci par un diagramme de description d'exigences temporelle montré dans

le chapitre IV), dans un diagramme de séquence TURTLE. Les points d'observation (*PO_Start* et *PO_End*) sont placés de part et d'autre dans les classes des applications utilisant le service N respectivement émettrice et réceptrice du message.

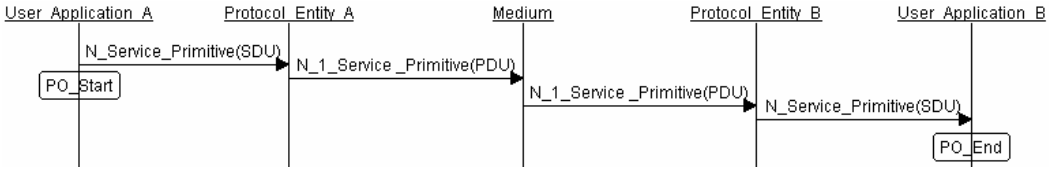

Fig.23. Placement des points d'observation dans un DS pour vérifier une exigence de QoS de *« end to end delay »*

A partir de la définition des différentes *phases* du protocole, des *primitives de services*, des *PDU* échangées entre entités de protocole et de leur intégration dans les primitives du service sous-jacent, une ébauche de modèle peut alors être construite à partir du pattern à trois couches défini dans le chapitre II section 5. Nous passons donc à la phase de conception.

4.2. Les règles liées à la phase de conception

La première étape consiste à définir l'architecture du protocole en construisant un *diagramme de classes*. Pour respecter la règle *C 1*, ce diagramme repose sur le pattern à trois couches défini dans le chapitre II. Ce pattern est enrichi pour prendre en compte règles *C 2* et *C 3*.

C 1. Le diagramme de classes du modèle du protocole à concevoir repose sur le modèle de référence à trois couches défini dans le chapitre II section 5.

C 2. L'abstraction de l'application ou des services utilisant le protocole N prend en compte l'aspect interaction de fonctionnalités. Pour cela elle doit être représentée par trois niveaux qui sont le séquenceur de scénarii, les appels de fonctionnalités et les rôles correspondants aux scénarii (Fig.24).

Comme [GOT 02], qui définit un protocole de haut niveau par un ensemble de micro-protocoles, nous souhaitons représenter l'application comme un scénario appelant des classes de fonctionnalités assimilable aux micro-protocoles définis en [GOT 02]. Il arrive très souvent qu'un même utilisateur doive remplir différents rôles dans une session et appeler différentes fonctions. Nous avons représenté l'abstraction de l'application de la manière suivante :

- *Séquenceur de scenarii* : les utilisateurs du service sont décrits de la manière la plus abstraite possible et dans tous les cas en préjugeant le moins possible d'une application particulière. C'est dans ce niveau, que sont spécifiés dans le *diagramme d'activités* les scénarii incluant toutes les fonctionnalités décrites par les *diagrammes globaux d'interactions*. Ces fonctionnalités apparaissent dans les scénarii via des *appels de fonctionnalités*.
- *Appels de fonctionnalités* : on représente chaque appel de fonctionnalité par une classe. Le niveau au-dessus (Séquenceur de scénarii) invoque les différentes classes fonctionnalités pour que celles-ci puissent être exécutées.
- *Rôles correspondants aux scénarii* : à chaque fonctionnalité correspondent des rôles d'utilisateurs différents. Les objets correspondant aux rôles d'utilisateurs sont ensuite

connectées aux différentes entités de protocoles correspondant aux utilisateurs qui, ensemble, exécutent les services nécessaires.

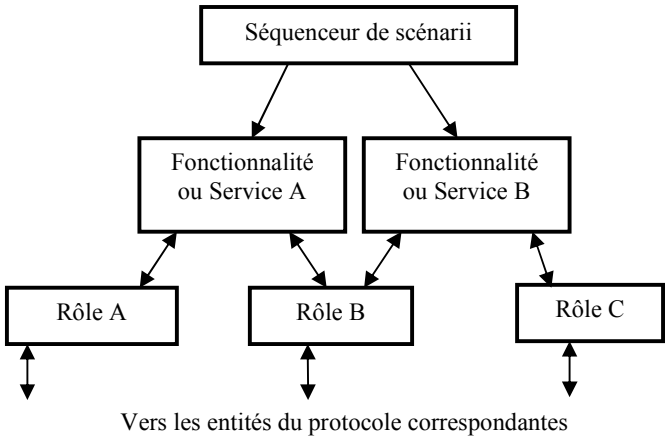

Fig.24. Description de l'abstraction des couches supérieures

C 3. Le modèle du service sous jacent ou médium doit pouvoir prendre en compte les aléas définis dans la section 4.1.2. Pour cela il doit intégrer des modules d'Aléas (cf. Fig.25).

Comme [GIN 05], qui définit n modules (appelés *impairements* en [GIN 05]) pour représenter n routeurs dans les simulations avec leurs caractéristiques propres, nous souhaitons modéliser les différents *Aléas* par différents modules d'*Aléas* pour caractériser le médium et les dégradations qui lui sont associées. Par exemple, pour modéliser un médium de type *non fiable,* on représente quatre modules d'*Aléas* : un module de *transmission* prenant en compte les délais et les latences de transmission, un module de *pertes,* un module de *duplication* et un module de *déséquencement*. Ces modules (correspondant aux *hypothèses appelées à être levées* en section 4.1.2) sont ajoutés au fur et à mesure des itérations. Ainsi les différents *Aléas* sont représentés et vérifiés indépendamment dans un premier temps puis ajoutés dans un second temps aux précédent *Aleas* : ceci garantit donc les tests de non-régression vis-à-vis du modèle considéré dans l'itération précédente. La Fig.25 décrit la manière de brancher ces différents modules *d'Aléas* (ou *impairements*) dans le modèle du *service sous-jacent*.

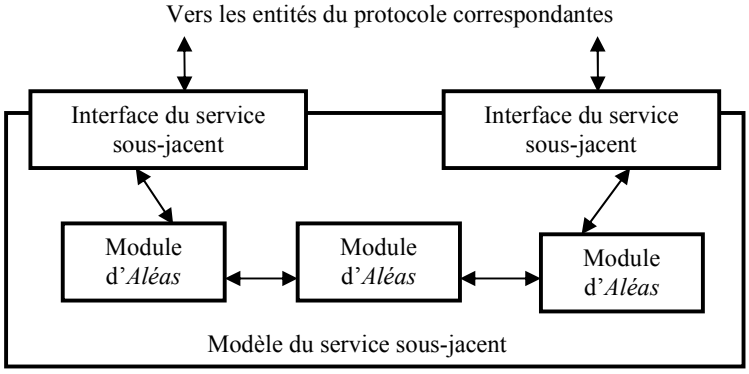

Fig.25. Description du modèle du service sous-jacent

Le comportement des entités de protocoles peut alors être spécifié par le concepteur. Le comportement de chaque module est donc construit, à partir des scénarii définis dans l'analyse (dans un premier temps les scénarii élémentaires décrit dans les *diagrammes de séquences* des fonctionnalités puis les scénarii de plus haut niveau définis dans les *diagrammes globaux d'interaction*).

Le processus de *vérification* ne diffère pas de celui présenté en section 3.4. Les itérations sont ici levées en fonction de l'insertion des différents *aléas* (cf. section 4.1.2). Il faut noter que l'insertion de l'*aléa* de déséquencement dans le modèle est très délicat ; en effet celui-ci augmente la combinatoire de l'entrelacement des états de façon significative. Cela peut être à l'origine du phénomène d'*explosion combinatoire* qui conduit à la non-construction du graphe d'accessibilité. C'est dans ce sens que seules les dernières itérations comportent l'*Aléas* de déséquencement.

Instanciation sur le profil TURTLE :

La Fig.26 décrit le pattern du *diagramme de classes* TURTLE d'un modèle de protocole en respectant les règles *C 1*, *C 2* et *C 3* énoncées dans la partie 4.2. Conformément à la règle *C 1*, l'architecture du modèle repose donc sur trois couches : *l'abstraction de l'application*, les *entités de protocoles à vérifier* et l'*abstraction du médium*.

L'*abstraction de l'application,* comme définie dans la règle *C 2*, est scindée en trois niveaux : le *séquenceur de scénarii* dont le *diagramme d'activités* sera conforme au *diagramme global d'interactions*, les modules d'*appels de fonctionnalités* qui exécuteront la fonctionnalité correspondante et les *rôles correspondant aux fonctionnalités* qui sont attribués aux entités de protocole durant l'exécution d'une fonctionnalité. C'est dans cette couche et plus précisément dans la sous-couche *d'appel de fonctionnalités* que les observateurs vont se greffer au modèle. Ceci permet de définir les points d'observations facilement et directement au niveau des appels de fonctionnalités. Notons que toutes les actions des modules d'*appels de fonctionnalités* et de *rôles correspondant aux services* doivent être exécutées de manière atomique (c'est-à-dire instantané), ces modules ne devant pas influer sur le modèle.

Enfin le *modèle du médium* doit être conforme à la règle *C 3.* Il est donc constitué d'*interfaces* (qui représentent, par exemple pour un protocole de niveau transport, les contraintes liées à la segmentation des données) et de modules d'Aléas qui sont distingués eux-mêmes par un module temporel qui représente les temps de transmission (dans un premier temps le modèle sera un délai puis s'affinera durant les itérations suivantes pour modéliser le type de service employé par le *médium* (cf. section 4.1.2).

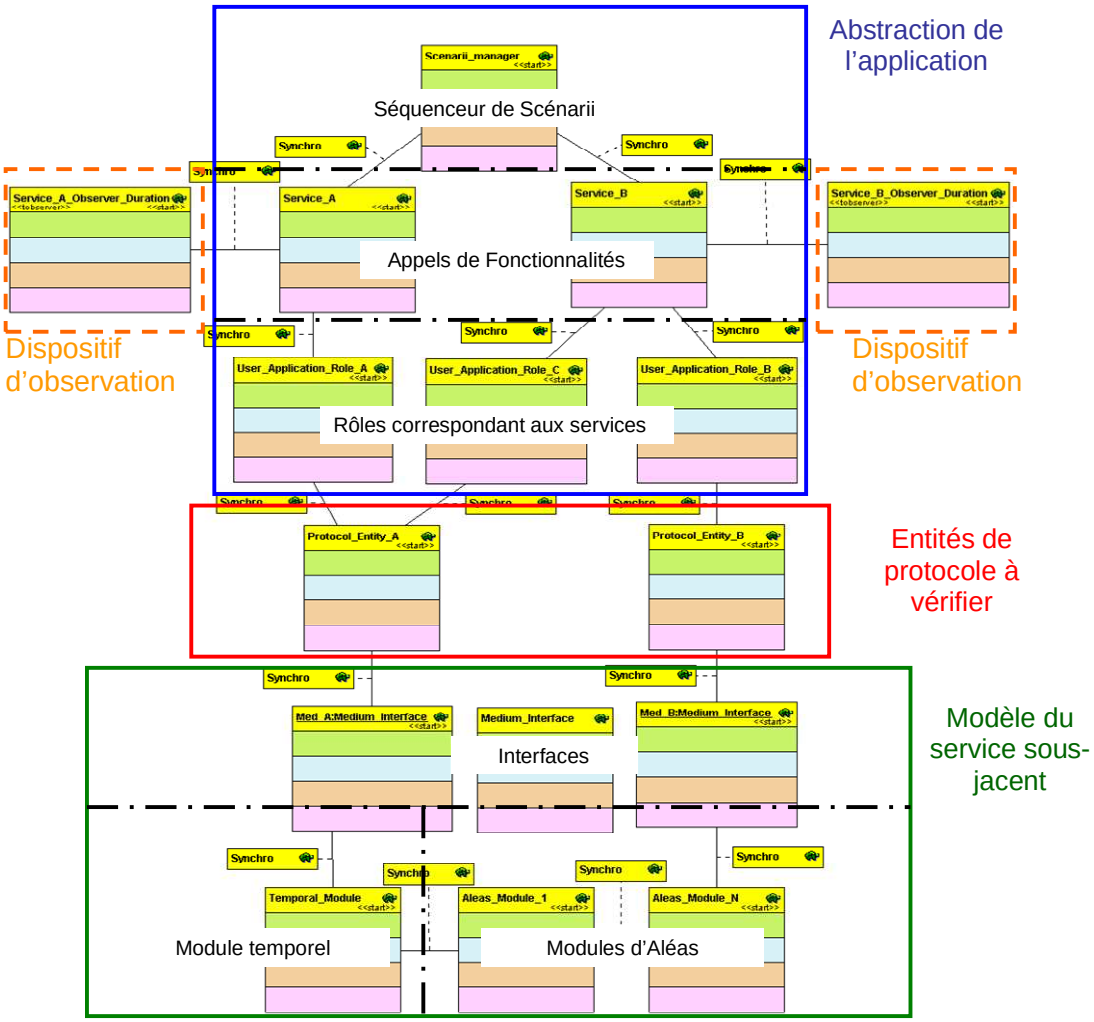

Fig.26. Pattern de diagramme de classes d'un protocole à vérifier

Le comportement de la couche *abstraction de l'application* est décrit par le *séquenceur de scénarii*. Ce dernier est construit de manière identique au *diagramme global d'interactions*. Les modules d'*appels de fonctionnalités* intégreront les *points d'observations* pour la vérification des exigences *non-fonctionnelles temporelles* liées à la durée d'un service (dans la Fig.26 pour les services A et B), comme l'indique la Fig.27. Tout comme sur la Fig.21, les *points d'observations* seront placés après les événements à observer pour respecter les règles de la section 3 du chapitre V. Notons que les modules de *rôles correspondant aux services* sont triviaux à mettre en œuvre et doivent être construits de manière à s'exécuter de façon *atomique* (instantanée) pour ne pas fausser le diagnostic des observateurs.

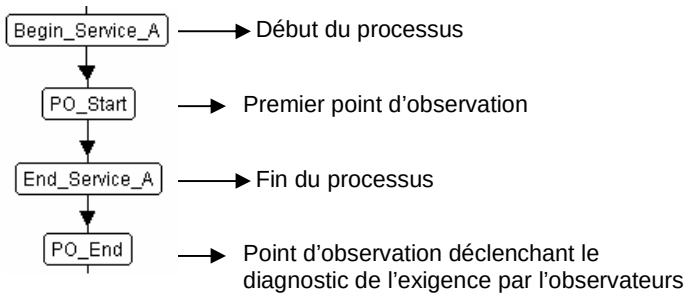

Fig.27. Placement des points d'observations dans la classe d'appel de fonctionnalité Service_A

Pour une exigence temporelle de type *jitter* pour un service N, le placement des points d'observations s'effectue conformément à la Fig.22, c'est dire sur l'objet concerné par l'exigence à savoir la classe *User_Application_Role* qui demande le service N. La Fig.28 montre le placement de ces points d'observations dans le diagramme de classes et dans le diagramme d'activités de la classe concernée. Dans ce dernier, les points d'observation (*PO_Start* et *PO_End*) sont placés respectivement après chaque réception de message.

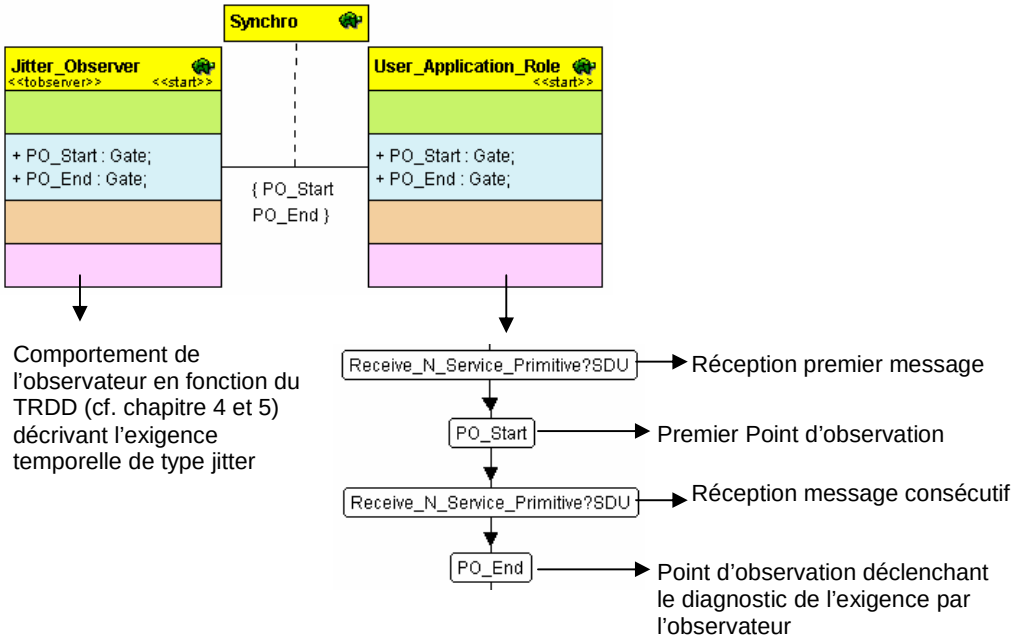

Fig.28. Placement des points d'observations pour vérifier une exigence de QoS de *« jitter »*

Pour une exigence temporelle de type *end to end delay* pour un service N, le placement des points d'observations s'effectue conformément à la Fig.23, c'est dire sur les objets concernés par l'exigence c'est-à-dire les classes *User_Application_Role_Sender* envoyant le paquet et *User_Application_Role_Receiver* recevant le paquet. La Fig.29 montre le placement de ces points d'observations dans le diagramme de classes et dans les diagrammes d'activités des classes

concernées. Dans ce dernier, les points d'observation (*PO_Start* et *PO_End*) sont placés respectivement après chaque réception de message.

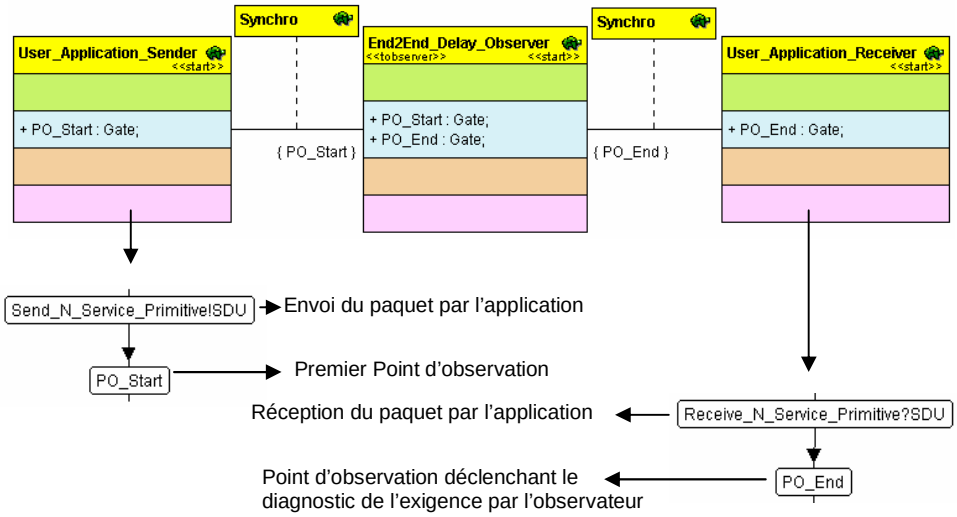

Fig.29. Placement des points d'observations dans les classes applications pour vérifier une exigence de QoS de *« end to end delay »*

Ensuite, il faut construire les *Aléas* définis dans le *médium* à partir de la définition des hypothèses appelées à être levées (cf. section 4.1.2). Très souvent, un protocole de type Transport dépend d'un service sous-jacent de type non-connecté ; c'est le cas du protocole réseaux IP (*Internet Protocol*).. Il faut donc pouvoir modéliser ce réseau par des modules *d'aléas* (*impairements* [GIN 05]) comme stipulé par la règle *C 3*. En ajoutant la règle *HL 4* il va falloir considérer trois types d'*Aléas* : le *déni de service* (DoS), la *duplication* et le *déséquencement*. La Fig.31 montre les *diagrammes d'activités* des trois modules d'Aléas cités précédemment ; en dessous des diagrammes d'activités figurent les graphes d'accessibilité issus du modèle présenté dans la Fig.30. Ce modèle correspond à l'envoi successif borné[7] de deux messages par l'objet A ayant pour paramètre un numéro de séquence et reçus par B via le module d'Aléas correspondant présenté dans la Fig.31.

[7] Les modules d'*Aleas* présentés dans cette section, sont définis à partir de l'hypothèse de l'envoi d'un nombre de messages borné ; dans le cas contraire (envoi infini de messages) ces modules sont « infinis » et provoquent le phénomène d'explosion combinatoire.

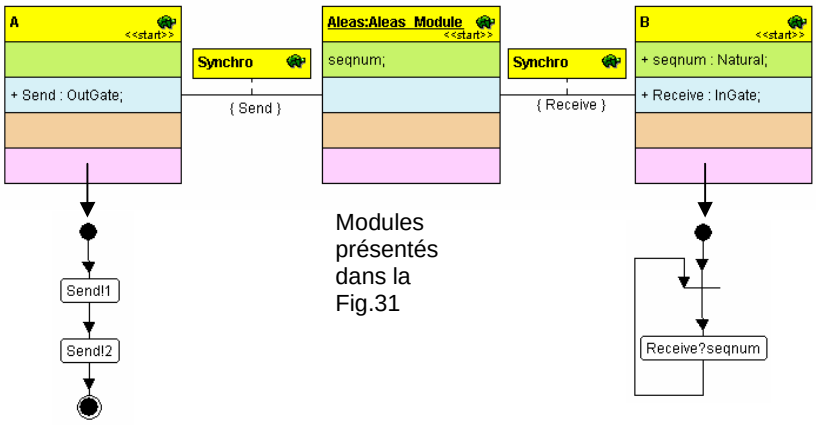

Fig.30. Exemple pour introduire les modules d'Aléas

Le module d'*aléa* de pertes (*DoS*) correspond à la non émission d'un message vers la classe B. Ceci est indiqué par le choix non-déterministe entre l'émission du message activité *Receive !Seqnum* et l'envoi du message interne *Lost*. Comme le montre le graphe d'accessibilité correspondant, chaque envoi de message peut conduire soit à l'émission du message, soit à la perte du message représenté par l'étiquette *Lost*.

Le module d'*aléa* de *duplication* correspond à la répétition du message à émettre vers B. Le choix non-déterministe est donc effectué après l'envoi du message vers B entre la réémission du message vers B (*Receive !Seqnum)* ou la réception du message provenant de A (*Send ?Seqnum*). Ceci apparaît sur le graphe d'accessibilité au travers des états numéro 2 et 4 bouclant sur eux-mêmes et représentant la duplication des messages.

Enfin le module d'*aléa* de *déséquencement* est moins trivial que ses prédécesseurs. Il correspond à la réception du message provenant de A (*Send ?Seqnum*) et aux tâches effectuées en parallèle, comme le montre l'opérateur de parallélisme, entre la réception et l'envoi du message. Le graphe d'accessibilité contient plusieurs chemins : d'une part, les deux messages peuvent être envoyés et reçus séquentiellement (chemin 0-1-3-5-7) ; d'autre part, les deux messages sont reçus consécutivement dans le médium puis envoyés vers B soit dans l'ordre (chemin 0-1-2-5-7) soit dans le désordre (chemin 0-1-2-4-6).

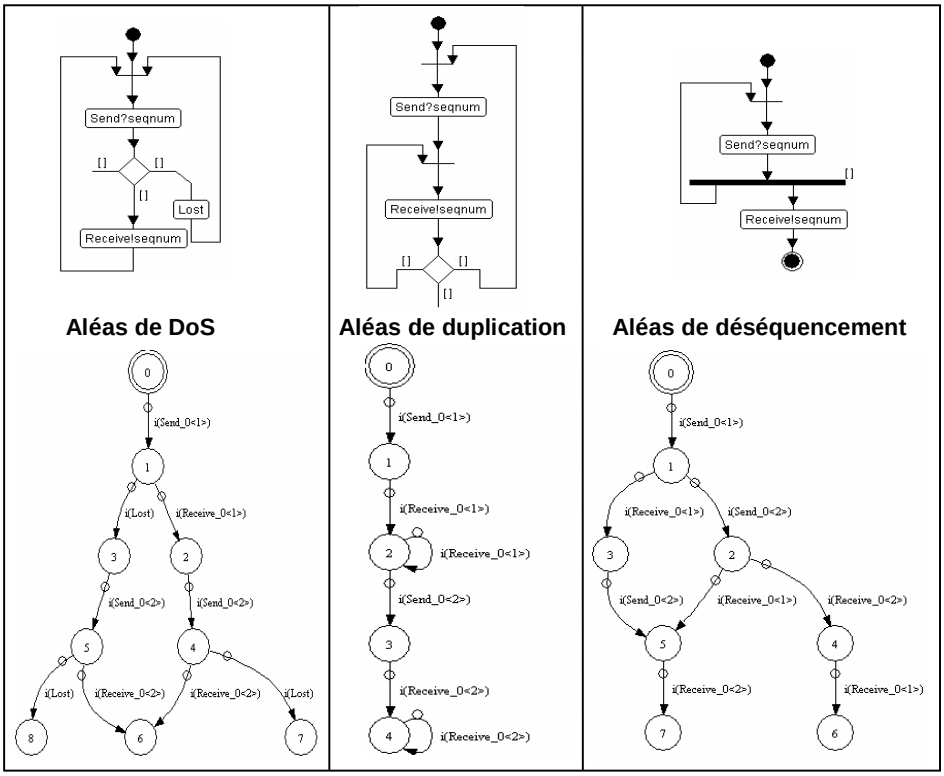

Fig.31. Diagrammes d'activités et graphes d'accessibilité correspondant aux Aléas liés à un service non-fiable

5. Conclusion

La Fig.32 fait une synthèse rapide de la méthodologie instanciée dans le profil TURTLE présentée dans ce chapitre. Les contributions, par rapport à la méthodologie présentée dans [APV 06] (restreinte ici aux phases d'analyse, conception et vérification[8]), concernent l'ajout de la phase de traitement des exigences et de la traçabilité des exigences non-fonctionnelles temporelles termes de vérification dans le cycle méthodologique TURTLE de conception et vérification de systèmes temps réel et protocoles.

Les contributions, présentées dans ce mémoire, reposent de plus sur la définition de diagrammes d'exigences et de TRDD permettant de décrire formellement les exigences non-fonctionnelles temporelles qui sont présentées dans le chapitre IV (cf. Fig.32). La définition de ces diagrammes de spécification des exigences permet d'effectuer un processus automatique de guidage de vérification des exigences non-fonctionnelles temporelles par construction d'observateur (cf. Fig.32). Cette contribution fait l'objet du chapitre V.

[8] Les phases de déploiement et de codage sont présentées dans [APV 06] mais ne sont pas traitées dans ce mémoire.

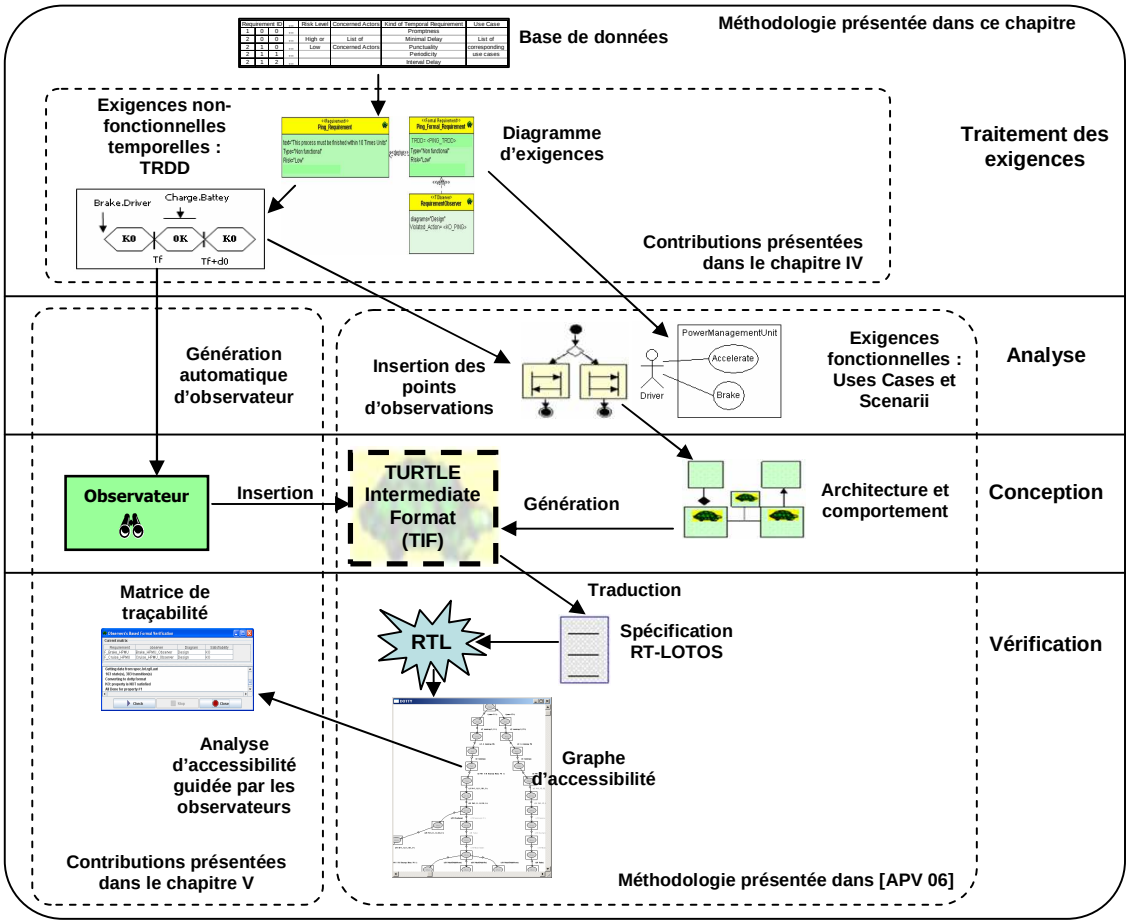

Fig.32. Récapitulatif de la méthodologie et présentation des contributions

Ceci conclut la définition d'une méthodologie de conception de systèmes et protocoles temps réel en contexte UML / SysML. Une application de cette méthodologie instanciée sur le profil TURTLE sur un exemple concret est proposée dans le chapitre VI concernant un protocole de communication de groupe sécurisé (projet SAFECAST [SFC]).

Chapitre IV. Langage de description d'exigences non-fonctionnelles temporelles

La vérification formelle des *exigences non-fonctionnelles temporelles* (que nous appelerons *exigences temporelles* par la suite) est au cœur de la méthodologie présentée dans le chapitre III. Vérifier ce type exigences nécessite donc de pouvoir les spécifier formellement. L'écriture de *formules de logiques temporelles* [HUT 04] (par exemple CTL ou LTL) présente un bon moyen de formalisation. Cependant la manipulation de ces formules nécessite un certain niveau d'expertise. L'utilisation d'un langage graphique est plus aisée pour une majorité d'utilisateurs ne connaissant pas ce type de techniques.

L'approche défendue dans ce chapitre privilégie donc les langages graphiques. Elle consiste à faire correspondre à chaque exigence temporelle définie dans un diagramme d'exigence SysML, un chronogramme appelé « diagramme de description d'exigence temporelle » ou TRDD (*Timing Requirement Description Diagram*). Ce TRDD décrit formellement une exigence temporelle présentant une occurrence de début et de fin afin de mesurer leur distance temporelle et permet de tracer ce type d'exigence au niveau de la vérification formelle (chapitre V).

Ce chapitre est structuré de la manière suivante. La section 1 présente un état de l'art des travaux dédiés aux langages de description formelle d'exigences temporelles. La section 2 introduit les diagrammes utilisés en TURTLE pour spécifier les exigences non-fonctionnelles temporelles. Il s'agit des diagrammes d'exigences SysML d'une part et des diagrammes de description d'exigences temporelles (TRDD) d'autre part. La section 3 montre les extensions proposées dans les diagrammes d'exigences TURTLE pour décrire l'aspect compositionnel des exigences temporelles en introduisant les notions de raffinement et de satisfaction d'exigences fonctionnelles. Enfin la section 4 conclut ce chapitre.

1. Etat de l'art et positionnement de nos travaux

Le chapitre II (section 2.2) a rappelé que la norme UML ne dispose pas de notation pour la phase de traitement des exigences. En pratique, celles-ci sont traitées hors du modèle UML, typiquement par la gestion d'une base de données d'exigences [HUL 04] [RAS 03] et au moyen d'outils tels que DOORS [BUC 05].

La normalisation de SysML par l'OMG, présente une solution pour pouvoir exprimer les exigences avec une notation proche d'UML. Un diagramme d'exigences SysML permet en effet d'exprimer des interactions entre exigences en intégrant les notions de raffinement, satisfaction et dérivation. Nous avons donc opté pour la construction d'un diagramme d'exigences à la SysML pour représenter les interactions entre exigences.

Néanmoins, SysML manque d'un support méthodologique pour recueillir correctement les exigences. De même, la pratique des premiers outils disposant de plug-ins SysML (par exemple, TAU G 2.3.1 [TAU]) montre que les exigences sont décrites de manière totalement informelle et sans lien automatisé avec les fonctionnalités de simulation de modèles. A contrario, les travaux présentés dans ce chapitre proposent d'étendre les diagrammes d'exigences SysML en formalisant des exigences temporelles et de les tracer dans le processus de vérification (cf. chapitre V).

En termes de méthodologie, KAOS (*Keep All Objective Satisfied* [LAM 06]) fournit un langage basé sur des formules de logique et sur la construction de méthodes pour une conception orientée but. L'outil correspondant, Objectiver [OBJ], permet de spécifier des exigences d'une façon systématique (par construction d'arbre d'exigences) permettant la traçabilité des exigences jusqu'à la définition des buts du système. L'intérêt de la méthodologie KAOS est de formaliser et de tracer de nombreux types d'exigences (fonctionnelles et non-fonctionnelles incluant la sécurité (face aux intrus), la sûreté de fonctionnement, le coût et la performance). Ce chapitre emprunte à KAOS [LAM 06] l'idée de formaliser des exigences temporelles par un processus de raffinement d'exigences basé sur la construction d'un arbre d'exigences. Les contributions apportées ici au profil TURTLE offrent une interface graphique pour la spécification formelle d'exigences temporelles (diagramme d'exigences et TRDD) contrairement à la logique supportée par KAOS (CTL* voir chapitre V section 1).

Avec l'approche orientée scénario, le processus de vérification consiste à faire un lien (notion de *matching* [BRA 05]) entre les scénarii et le modèle du système. Ainsi, *Timed Use Case Maps* [HAS 06] (voir TUCM dans le Tab.6) décrit les interactions entre cas d'utilisation en définisant le temps absolu au moyen d'un « *master clock* » mais aussi le temps relatif (Durée, Temporisateur). En décrivant les cas d'utilisations individuellement, *Context eXtended Use Case Chart* [DHA 07] (voir CxUCC dans le Tab.6) décrit à la fois le contexte de vérification et l'exigence à vérifier en construisant un diagramme d'observation (diagramme d'activités UML 2.0 étendu). *Visual Timed Events Scenario* [BRA 05] (voir VTS dans le Tab.6) représente les interactions entre événements. Un événement est vu comme une action qui apparaît potentiellement dans le système. VTS inclut les représentations temporelles d'ordre partiel entre événements et de contraintes temporelles relatives entre événements. *Live Sequence Charts* (LSC dans le Tab.6) étend les *Messages Sequence Charts* (MSC [ITU 96]) pour décrire les scénarios. LSC permet de faire la distinction entre scénarii possibles et nécessaires.

	Langages visuels basés sur les scenarii			
Nom	TUCM	CxUCC	VTS	LSC
Référence	[HAS 06]	[DHA 07]	[BRA 05]	[DAM 01]
Langage Formel	Clocked Transition System	Langage IF	Timed Computation Tree Logic	Automate de Bücchi
Type de vérification	Model Checking	Observateurs	Model Checking (UPPAAL/Kronos)	Model Checking

Tab.6. Langages visuels basés sur la description de scenarii

L'approche orientée scénarii, nous parait de trop bas niveau pour décrire les exigences. En effet, elle implique une « bonne » connaissance du comportement du système. De plus, la spécification d'exigences doit être effectuée durant la phase d'analyse. Au contraire, la méthodologie

présentée dans le chapitre précédent opère une distinction entre les scénarii définis dans la phase d'analyse et la phase de spécification des exigences.

Afin de réduire le fossé entre le recueil des exigences et leur formalisation, les exigences temporelles peuvent être représentées par un formalisme de type « chronogrammes » (*Timing Diagrams* [UML 07]). Les chronogrammes permettent de représenter les exigences temporelles d'une manière facile à lire. [CHO 05] donne un modèle formel, en formules LTL (*Linear Time Logic*) garantissant la sémantique formelle des *Timing Diagram* UML. Cependant, l'auteur souligne que les relations d'ordre partiel ne sont pas représentées dans les chronogrammes. L'outil ICOS [FRA 01] utilise le formalisme RT-STD (*Real Time Symbolic Timing Diagram*, voir RT-STD dans le Tab.2) pour spécifier les systèmes « hardware » (*System on Chip SoC*) en fonctions des entrées/sorties (un système est vu comme une boîte noire). L'outil ICOS couvre les phases de conception, vérification, simulation et prototypage. Pour leur part, les *Regular Timing Diagrams* [AML 99] (voir RTD dans le Tab.2) étendent les chronogrammes pour représenter les ordres partiels entre diagrammes. Cependant les langages visuels présentés dans Tab.2, sont utilisés pour spécifier l'aspect « conception du système » et non les exigences liées à ce système. C'est pourquoi nous avons défini les *Timing Requirement Description Diagram* (TRDD) pour exprimer les exigences temporelles présentant une occurrence de début et de fin afin de mesurer leur distance temporelle d'une manière graphique et formelle et qui s'intègre dans notre cycle méthodologique.

	Langages visuels basés sur les chronogrammes		
Nom	RT-STD	RTD	TRDD
Référence	[FRA 01]	[AML 06]	[FON 06b]
Langage Formel	Automate de Bücchi	Valeurs symboliques	RT-LOTOS
Type de vérification	Model Checking	Model Checking	Observateurs

Tab.7. Langages visuels basés sur les chronogrammes

Il faut souligner que les travaux de recherche dans le domaine de la vérification d'exigences temporelles restent encore ouverts. Par exemple, le projet TopCAseD [TCAD] cherche à convertir les diagrammes SysML en langages formels supportés par des outils de vérifications [BOD 06]. Par ailleurs, [JAN 99] définit un langage de modélisation graphique basé sur des canevas d'exigences (inspiré des définitions de [ALU 93] cités dans le chapitre II). Ces derniers sont traduits en formule LTL et vérifiés en utilisant le model-checker SPIN [HOL 03]. En [GRA 05], les exigences temporelles sont exprimées par des machines à états temporisées définissant le « modèle de contexte et d'exigences ». Le système modélisé et les machines à états des exigences sont traduits dans le langage IF défini dans le cadre du projet OMEGA et supporté par des outils de vérification [GRA 05]. Les auteurs construisent les diagrammes de contexte (automates temporels) à la main. Dans l'approche défendue dans ce mémoire, les observateurs (équivalents aux automates de contextes de [GRA 05]) sont générés automatiquement à partir des diagrammes de spécification d'exigences (diagrammes d'exigences et diagramme de description d'exigence temporelle) et les résultats de la vérification des exigences temporelles sont inclus dans une matrice de traçabilité.

2. Définition d'un langage de description d'exigences temporelles

Le travail présenté dans cette section vise à formaliser les exigences non-fonctionnelles – tout au moins lorsqu'il s'agit d'exigences temporelles quantitatives – et à établir de manière automatique un lien entre « exigences non-fonctionnelles temporelles » et « vérification formelle ». Nous prenons pour langage de modélisation non pas SysML, mais une version du profil UML TURTLE enrichie de diagrammes d'exigences SysML.

Fonctionnelles ou non, les exigences SysML demeurent informelles. En TURTLE, il devient possible de dériver d'une première exigence informelle une deuxième exigence, formelle cette fois, de nature temporelle et exprimée dans le langage graphique qu'est TRDD.

2.1. Formalisation d'exigences temporelles SysML

Le langage SysML [SysML 06] ne permet pas de distinguer clairement exigences informelles et formelles (point de départ du processus de vérification). C'est pourquoi nous avons étendu le langage SysML en introduisant un nouveau stéréotype *« Formal_Requirement »* permettant de faire la distinction entre les deux types d'exigences.

2.1.1. Exemple de diagramme d'exigences

Prenons pour exemple, un processus temps réel pour lequel on souhaite vérifier l'exigence suivante : « *le processus doit s'exécuter avant 10 unités de temps* ».

L'exigence informelle est exprimée par du texte. L'exigence formelle temporelle est exprimée dans le langage TRDD présenté en section 2.2. Dans l'exemple de la Fig.33, le TRDD a pour nom Process_TRDD.

La Fig.33 décrit un diagramme d'exigences composé d'une exigence informelle dont on dérive une exigence formelle (par la relation <<derive>>).

Une exigence informelle (stéréotypée par <<Requirement>>) possède quatre attributs : un *identifiant*, du *texte* (une description informelle de l'exigence), un *type* (fonctionnelle, non-fonctionnelle) et un *niveau de criticité* (haut ou bas).

Une exigence formelle temporelle (stéréotypée par <<Formal Requirement>>) possède également un type et un niveau de criticité. En complément, un attribut TRDD décrit l'exigence formelle sous la forme d'un chronogramme exprimé dans le langage TRDD. Dans cet exemple, ce chronogramme s'appelle P_TRDD.

L'exigence exprimée par un TRDD est vérifiée formellement par un observateur (relation <<verify>>). Un observateur (stéréotypé par <<TObserver>>) possède un nom, une référence aux type de diagrammes qui seront pris en compte dans le dispositif de vérification (attribut *diagrams*) et un attribut « Violated_Action » qui spécifie l'identificateur utilisé dans le graphe d'accessibilité pour caractériser (lorsqu'elle existe) la violation d'exigence (Dans notre exemple, l'identificateur est en question est KO_Process). Le dispositif de vérification des exigences temporelles décrites par un TRDD est présenté dans le chapitre V.

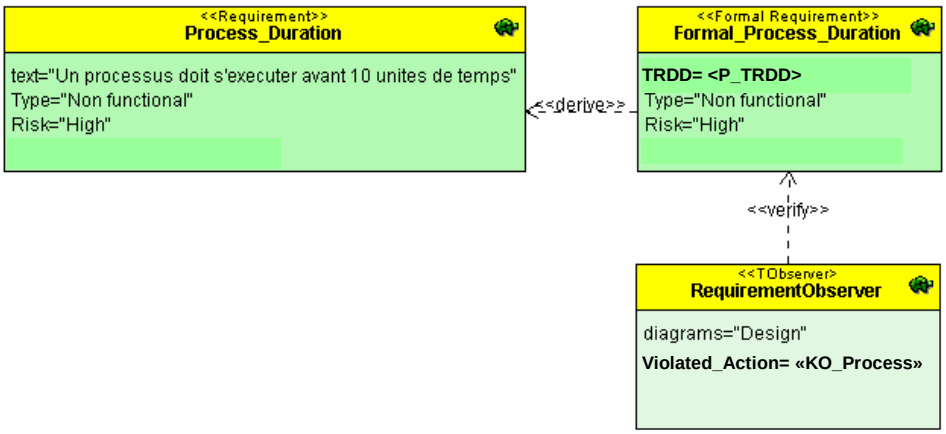

Fig.33. Diagramme d'exigences TURTLE

2.1.2. Définitions formelles

Un diagramme d'exigences (noté par $\mathcal{RD}$) est un graphe 5-parties composé d'exigences informelles (IR) et temporelles formelles (FTR), d'observateurs (OBS) et de fonctions « derive » (fonction deriv) et « verify » (fonction verif). La relation « derive » relie une exigence temporelle formelle à une exigence informelle, la deuxième étant dérivée de la première. La relation « verify » lie un observateur à une exigence temporelle formelle, pour représenter l'opération de vérification de cette exigence temporelle formelle par un observateur. La vérification de l'exigence par d'autres moyens que les observateurs va au-delà du champ de mise en œuvre de cette thèse, mais peut être exprimée en SysML (cf. pattern de diagramme d'exigences du chapitre III section 3.1.3) et dans les diagrammes d'exigences TURTLE.

Définition 2 : Diagramme d'exigences

$\mathcal{RD}$ = (IR, FTR, OBS, deriv, verif)

où

IR est un ensemble d'exigences informelles

FTR est un ensemble d'exigences formelles

OBS est un ensemble d'observateurs

deriv: FTR → IR est une fonction décrite par <<derive>>

verif: OBS → FTR est une fonction décrite par <<verify>>

Un nœud IR, stéréotypé par <<Requirement>>, est défini par un nom (chaîne de caractères), une description textuelle de l'exigence (chaîne de caractères), un type (fonctionnel, non fonctionnel) et un niveau de criticité (haut ou bas).

Définition 3 : Exigence informelle

ir = {Nom, Texte, Type, Risque}
où
Nom: string
Texte string
Type Є {Fonctionnelle, Non-Fonctionnelle}
Risque Є {Haut, Bas}

Un nœud FTR, stéréotypé par <<Formal_Requirement>>, est défini par un nom (chaîne de caractères). Un diagramme de description d'exigence temporelle (TRDD défini en section 3.2), un type non fonctionnel et un niveau de criticité (haut ou bas). Un TRDD définit une exigence temporelle quantitative qui est une exigence non fonctionnelle.

Définition 4 : Exigence temporelle formelle

ftr = {Nom, TRDD, Type, Risque}
où
Nom: string
TRDD = < TRDD > voir définition 5
Type Є {Non-Fonctionnelle}
Risque Є {Haut, Bas}

Un observateur, stéréotypé par <<TObserver>>, est défini par un label (son nom), le type de *package* (le *diagramme de composants* TURTLE [APV 06] (contenant un diagramme de classes et une collection de diagrammes d'activités), auquel il sera greffé pour guider la vérification), et un label de violation d'exigence (*Violated_Action*). En cas de violation d'exigence, une transition du graphe d'accessibilité sera étiquetée par ce label. La violation de l'exigence sera ainsi repérable par recherche sur les étiquettes du graphe (rappelons ici que l'environnement de vérification de TURTLE repose en autres sur la construction d'un graphe d'accessibilité).

Définition 5 : Observateur

obs = (Nom, Package, Violated_Action)
où
Nom: string
Package: Package_ref où Package_ref fait de référence à un package (Package_ref)
Violated_Action: Gate_ref où Gate_ref fait référence à une porte

2.1.3. Méta-modèle UML

En TURTLE étendu, un diagramme d'exigences (classe *TRequirement_Diagram*[9] de la Fig.34), améliore le diagramme d'exigences SysML (classe *::SysML ::'Requirement diagram'*) et comporte trois types de nœuds : des d'exigences informelles (classe stéréotype *TInformal_Requirement*), des exigences temporelles formelles (classe stéréotype *TFormal_Requirement*) et des observateurs (classe stéréotype *TObserver*).

[9] L'indicatif T désigne TURTLE

Le méta-modèle du diagramme d'exigences TURTLE est présenté en Fig.34.

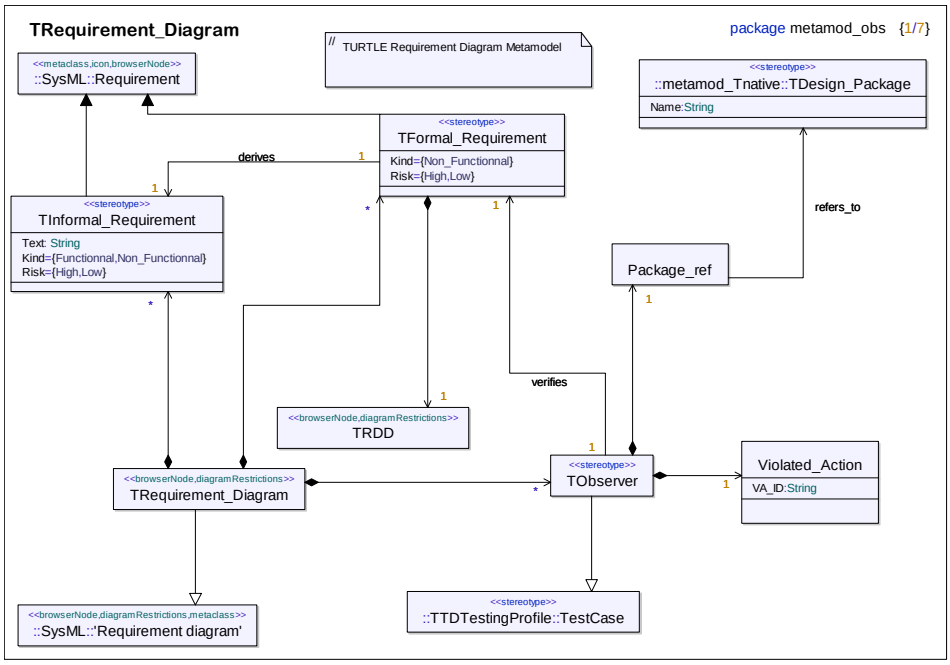

Fig.34. Méta-modèle du diagramme d'exigences TURTLE

Les exigences informelles et temporelles formelles (respectivement classes stéréotypées *TInformal_Requirement* et *TFormal_Requirement*) sont des nouveaux stéréotypes définis à partir des exigences SysML (classe *::SysML ::Requirement*). Une exigence temporelle formelle dérive d'une exigence informelle (relation d'association étiquetée par *derive*).

Les observateurs (classes stéréotypées *TObserver*) sont des nouveaux stéréotypes issus entre autres du stéréotype SysML « *TestCase* » (classe stéréotype *::TTDTestingProfile ::TestCase*) qui correspond à un dispositif de « vérification » de l'exigence. Dans notre cas, la Tclass de l'observateur est greffée au modèle de conception TIF (cf. chapitre III section 2.2). L'observateur est une spécialisation de la classe stéréotypée *Tclass*. Nous considérons donc qu'un observateur vérifie une et une seule exigence temporelle formelle ; ceci est représenté par la relation d'association étiquetée par *verify* liant un « *TestCase* », ici un observateur, à une exigence [SysML 06] temporelle formelle.

Comme l'indique la définition 5, un observateur est composé :

- D'une action de violation d'exigence (classe *Violated_Action*) qui caractérise sur le graphe d'accessibilité la violation d'exigence par la présence de l'étiquette VA_ID (attribut *VA_ID*).
- D'une référence d'un package de conception (classe *Package_Ref*) qui se réfère à un package TURTLE (classe *::metamod_T_native ::TPackage*).

2.2. Timing Requirement Description Diagram (TRDD)

Après avoir spécifié le processus de formalisation des exigences temporelles, nous avons choisi de formaliser ces dernières par un langage graphique permettant de les décrire dans un langage

simple et qui s'intègre dans la phase de traitement des exigences de la méthodologie présentée dans le chapitre III.

Nous avons donc choisi de spécifier ces exigences non-fonctionnelles temporelles par des *Timing Diagrams* UML 2.1 [UML 07]. Il s'agit à l'origine de « chronogrammes » qui peuvent être utilisés en substitut de machines à états pour décrire le comportement interne d'objets. Ici, les *Timing Diagrams* [UML 07] sont étendu en TRDD (*Timing Requirement Description Diagram*) utilisés non pas pour décrire le comportement d'objets mais pour exprimer graphiquement des exigences temporelles présentant une occurrence de début et de fin afin de mesurer leur distance temporelle.

2.2.1. Exemple

Pour l'exigence formelle de la Fig.33 « *le processus* (décrit par les *points d'observations* pour la vérification de l'exigence Start_P et End_P) *doit s'exécuter avant 10 unités de temps* », le TRDD de la Fig.35 est construit en incluant les *points d'observations* (première occurrence *Start_P* marquant le début de l'exigence et seconde occurrence *End_P* marquant la fin de l'exigence) qui sont définis comme des *points d'observations* (*Observation_Points* sur la Fig.35) d'une part et les frontières temporelles de satisfaction et violation d'exigences (respectivement OK et KO) (*ligne de vie de l'exigence*) d'autre part. La ligne de vie de l'exigence est elle-même composée d'un élément de *début*, d'éléments de *description d'exigences* et d'un élément de *fin*.

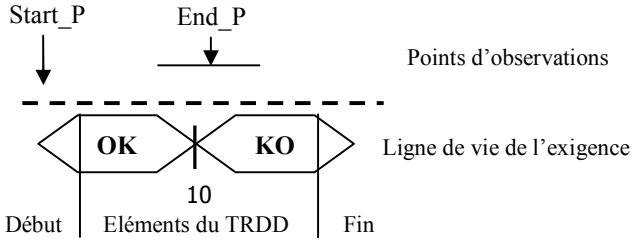

Fig.35. Diagramme de description d'exigence temporelle de P_TRDD

2.2.2. Définition formelle

Un diagramme de description d'exigence temporelle (noté par TRDD) est un graphe 2-parties composé d'une ligne de vie d'exigence (*Requirement_Lifeline*) et de points d'observations (*Observations_Points*). La ligne de vie de l'exigence est décrite par un élément de début (*Begin*), d'éléments de descriptions temporelles (*TRDD_Element*) et d'un élément de fin (End). Les éléments de descriptions temporelles sont constitués de Frontières Temporelles (*Temporal_Frontier*) et d'état d'exigences (*Requirement_State*) correspondant à la violation/satisfaction d'exigences respectivement représentées par KO (*KO_Part*) et OK (*OK_Part*).

Les TRDD mesurent l'écart temporel entre deux occurrences d'événements qui sont appelés *points d'observations*. Ces derniers sont définis dans le TRDD d'une part par l'action présentant le l'occurrence marquant le début de description d'exigence (*Start_Action*) et, d'autre part par la seconde occurrence déclenchant le diagnostic de l'observateur (*Capture_Action*).

Définition 6: TRDD

TRDD = (Requirement_Lifeline, Observation_Points)

où

Requirement_Lifeline= <Begin, TRRD_Element, End>

où

TRDD_Elements est un ensemble de TRDD_Element

où TRDD_Element Є {Temporal_Frontier, Requirement_State}

et Requirement_State Є {OK_Part,KO_Part}

Observations_points = <Start_Action, Capture_Action>

Le Tab.8 présente les différents éléments du langage graphique exprimables dans un TRDD.

Label du TRDD	Symbole Graphique	Description informelle
Begin	***TRDD*** <	**Début du TRDD**
T_F(Tin)	***TRDD1*** >|< ***TRDD2*** T_{in}	**Frontière temporelle entre violation et satisfaction d'exigence**
OK_Part	***TRDD*** **OK**	**Partie temporelle où l'exigence temporelle est satisfaite**
KO_Part	***TRDD*** **KO**	**Partie temporelle où l'exigence temporelle est violée**
End	***TRDD*** >	**Fin du TRDD**
Start_Action	« Start_Action» ↓	**Première occurrence présentant le début de description de l'exigence**
Capture_Action	« Capture_Action» ↓	**Seconde occurrence déclenchant le diagnostic de l'exigence par les observateurs**

Tab.8. Sémantique des éléments du TRDD

La Fig.36 illustre les règles de grammaire pour la construction d'un diagramme de description d'exigence temporelle.

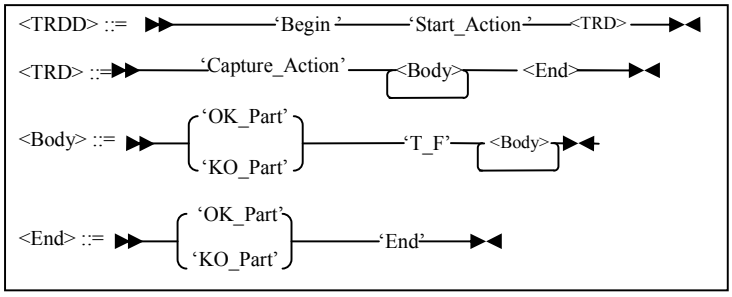

Fig.36. Grammaire pour la construction de description d'exigences temporelles

2.2.3. Méta-modèles UML

Le méta-modèle d'un TRDD est présenté en Fig.37. Un diagramme de description d'exigences (classe *TRDD*) étend les Timing Diagrams UML 2.1 en notation concise (Value Lifeline Timing Diagram) (classe *::TTDMetamodel ::Diagram*). Il contient un attribut *n_TRD* (nombre d'éléments de description d'exigence figurant dans le diagramme de description d'exigence).

Comme l'indique la définition 6, un TRDD est composé :

- D'une ligne de vie d'une exigence (classe *Requirement_Lifeline*) définie à partir de la méta-classe *::TTDMetamodel ::LifeLine*.
- De deux points d'observations (classe *Observation_Points*) qui correspondent individuellement à des portes de communication (classe *TGate*) des objets TURTLE appartenant au système observé. Ces portes seront ensuite synchronisées avec les portes d'observation de l'observateur (voir section 4.).

La ligne de vie de l'exigence temporelle formelle (classe *Requirement_Lifeline*) est composée des symboles (classes stéréotypées « icon » correspondant à un label) suivants :

- Un symbole *Begin* qui illustre le début du TRDD.
- Un symbole *End* qui termine le TRDD.
- De un à N *Requirement_State* dérivés de la méta-classe *::TTDMetamodel::State* qui correspondent à la zone temporelle de violation/satisfaction de l'exigence. Les *Requirement_States* sont étiquetée par OK ou KO.
- N-1 *Temporal_Frontier* qui correspond à un changement d'état de l'exigence (montré par l'association *switches_the_next* décrite par la formule OCL en commentaire sur la Fig.38.). Cette classe contient comme attribut la date de la frontière temporelle.

NB : N est égal au nombre total d'états d'exigences (*Requirement_State*) dans le TRDD.

Les points d'observations (classe *Observation_Points*) correspondent à des portes à observer sur des Tobjects du système. Ils sont composés des symboles (classes stéréotypées par « icon ») suivants :

- *Start_Action* est le premier point d'observation déclenchant la capture de l'exigence ; il est associé à la porte de communication *Start* de l'observateur (voir section 3).
- *Capture_Action* est le second point d'observation qui déclenche la fin de capture de l'exigence ; il permettra d'établir un diagnostic sur la satisfaction/violation de l'exigence par l'observateur. Ce point d'observation sera associé avec la porte *Capture* de l'observateur (voir section 3).

Le méta-modèle du TRDD apparaît sur la Fig.37.

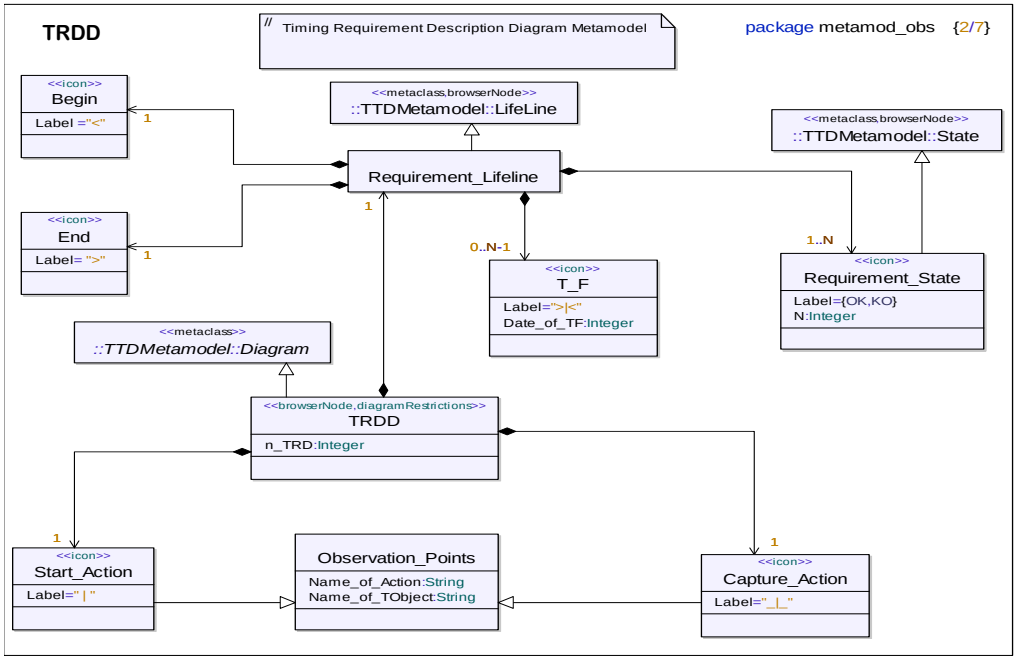

Fig.37. Méta-modèle du TRDD

Les règles de construction d'un TRDD sont identifiées par le méta-modèle de la Fig.38.

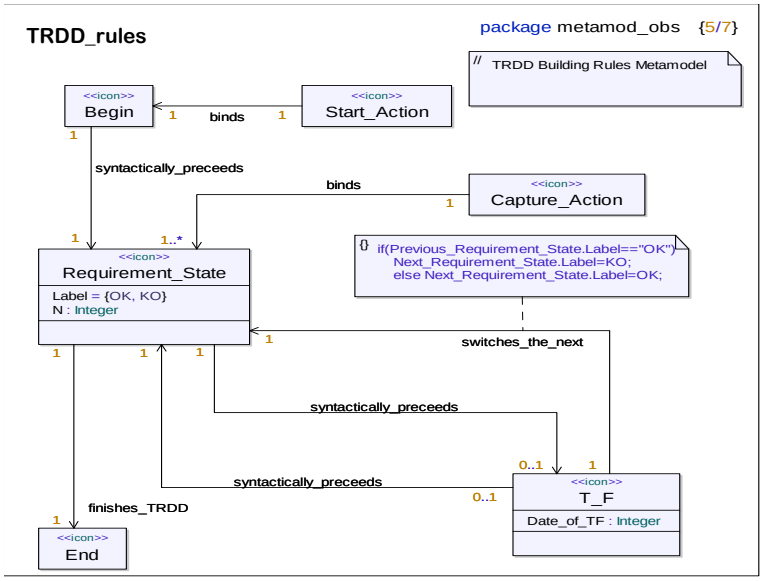

Fig.38. Méta-modèle décrivant les règles de construction du TRDD

2.2.4. Pouvoir d'expression des TRDD

Ce paragraphe a pour objectif d'illustrer ce qui peut être exprimé dans un TRDD. Nous comparons différents patterns de TRDD avec les exigences proposées dans la taxonomie définie dans

le chapitre II section 1 à partir de [ALU 93]. Ces exemples sont illustrés par la Fig.39, un TRDD exprime :

- En a) les exigences contenant une frontière temporelle et qui doivent se terminer/commencer à T unités de temps. Ces exigences correspondent respectivement aux propriétés de *délais maximum/minimum*.
- En b) les exigences contenant deux frontières temporelles et qui doivent être comprises/exclues dans les intervalles respectifs $]T_1 ; T_2[$ et $[T_1 ; T_2]$. Ces exigences correspondent aux propriétés de *délais bornés* et de *ponctualité* (par exemple à date T on aura $T_1 = T-1$ et $T_2 = T+1$).
- En c) une exigence contenant n frontières temporelles et n+1 états d'exigences (OK ou KO), où Ch1 et Ch2 représentent les deux possibilités de représentation (OK ou KO), la frontière temporelle étant décrite par la date T_n. Ces exigences ne sont pas référencées dans la taxonomie énoncée dans le chapitre II. Ceci montre donc l'originalité du formalisme TRDD. Son pouvoir d'expression permet d'exprimer d'autres exigences que celles recensées dans la taxonomie du chapitre II (section 1).

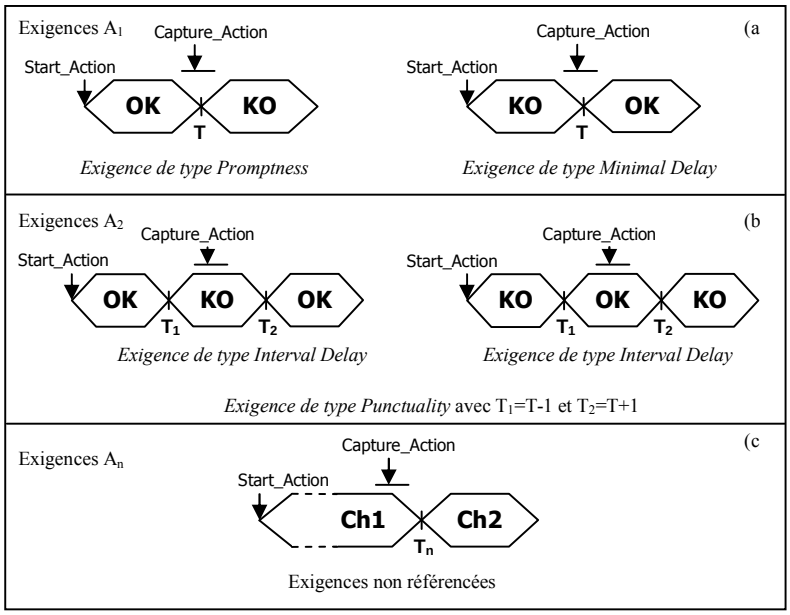

Fig.39. Exemples de TRDD

Toutes les exigences construites dans le formalisme du TRDD sont de nature périodique (*Periodicity Property*). Le processus de vérification, en l'occurrence l'observateur généré à partir du TRDD, présenté dans le chapitre V, permet de reboucler sur le premier point d'observation dès que le second point a été franchi ou que l'exigence temporelle n'est pas satisfaite.

3. Composition d'exigences

Les diagrammes d'exigences TURTLE (cf. section 2.1) ne prennent pas en compte les notions de raffinement et de satisfaction (a quelles fonctionnalités correspondent les exigences temporelles qui doivent être vérifiés ?). Or le langage SysML [SysML 06] permet par les concepts de *composition*

et de *satisfaction* d'exprimer ces deux notions. Cette section formule une proposition d'extension des diagrammes d'exigences TURTLE supportant les notions de *composition* et *satisfaction*.

3.1. Les concepts de composition et satisfaction

Dans le domaine de l'ingénierie des exigences [HUL 04] [LAM 06] [RAS 03], on distingue généralement deux types de relations de composition d'exigences :

- La relation de *composition* vise à décrire un processus de raffinement d'exigence.
- La relation de *satisfaction* permet de faire un lien entre exigences fonctionnelles et exigences non-fonctionnelles décrivant respectivement un mécanisme à construire et le but que ce dernier doit atteindre.

Le concept de *composition* est employé très souvent dans la phase de construction de bases de données. Des outils de base de données comme DOORS [BUC 05] ou ARCaDe [RAS 05], expriment ce concept par une classification des exigences en fonction des numéros (1.0 pour une exigence souche, 1.1 pour l'exigence raffinée produite à partir de 1.0). Dans la méthode KAOS [LAM 05], on construit un arbre d'exigences pour raffiner celles-ci jusqu'à obtenir des exigences « feuilles », point de départ du dispositif de formalisation des exigences. Dans la méthodologie présentée dans le chapitre III, il est possible d'intégrer ce concept dans la construction de la base de données (cf. section 3.1.1), mais pas dans les diagrammes d'exigences TURTLE. Or ce concept est pris en compte dans le langage SysML [SysML 06] par la relation de composition d'exigences qui est exprimée par une association entre exigences notée avec un signe « + » entouré. Nous proposons donc d'inclure cette relation de *composition* dans les diagrammes d'exigences TURTLE.

Avant l'apport de SysML pour la spécification des exigences, le concept de *satisfaction* d'exigence dans les outils de vérification orienté UML était généralement employé dans la phase d'analyse [HAS 06] [DHA 07]. Après avoir identifié les fonctionnalités du système, on définissait des exigences temporelles liées à des exigences fonctionnelles (ou fonctionnalités) telles que la durée d'une fonctionnalité [HAS 06] [DHA 07] [BRA 05] [DAM 01]. Cependant, il est admis dans la communauté de l'ingénierie des exigences [LAM 06] [HUL 04] que les exigences non-fonctionnelles sont très souvent définies dans le cahier des charges avant les exigences fonctionnelles qui décrivent des mécanismes. Pour palier ce fossé entre recueil et spécification d'exigences, [LAM 06] lie les exigences raffinées au concept d'*agent* (qui sont vues comme des entités du système devant satisfaire les exigences). C'est pourquoi nous choisissons de lier chaque exigence non-fonctionnelle temporelle raffinée à une (ou plusieurs) exigence(s) fonctionnelle(s) qui peuvent être vues comme des agents dans [LAM 06]. Nous proposons d'inclure ce concept non seulement dans les bases de données [RAS 03], mais aussi dans les diagrammes d'exigences SysML [SysML 06]. Tout comme le concept de *composition*, le concept de *satisfaction* est pris en compte en SysML par la relation de dépendance stéréotypé par « *satisfy* » liant une exigence fonctionnelle à une exigence non-fonctionnelle.

3.2. Proposition d'extension des diagrammes d'exigences

Reprenons l'exemple courant de la section 2.1.1 (cf. Fig.33). Le processus qui doit s'exécuter en moins de 10 unités de temps est maintenant décomposé en plusieurs sous processus (dont un processus de début *Begin_Process* et un processus de fin *End_Process*). Pour ces deux derniers, on souhaite aussi vérifier des exigences temporelles liées à leurs durées (respectivement en moins de 2 et 3 unités de temps).

Le diagramme d'exigences TURTLE étendu de la Fig.40 décrit les exigences énoncées dans le paragraphe précédent. Les extensions proposées dans le diagramme d'exigence TURTLE intègrent les concepts de *composition* d'exigences et de *satisfaction* d'une exigence non-fonctionnelle temporelle par une exigence fonctionnelle.

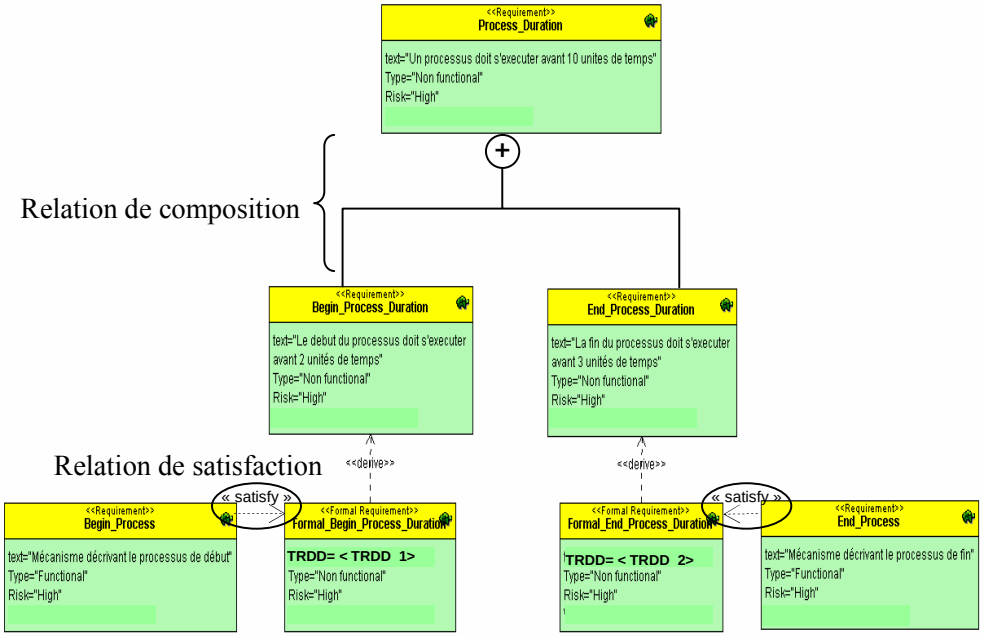

Fig.40. Proposition d'extension des diagrammes d'exigences TURTLE

L'exigence non-fonctionnelle temporelle informelle *Process_Duration* décrite par les Fig.33 et Fig.35 est décomposée en deux « sous » exigences temporelles informelles *Begin_Process_Duration* et *End_Process_Duration*. Ce processus de raffinement d'une exigence se fait à l'aide d'une relation de composition d'exigence [SysML 06] (association avec un signe « + » entouré) entre l'exigence *Process_Duration* et les exigences raffinés *Begin_Process_Duration* et *End_Process_Duration*. La notion de *composition* s'applique uniquement aux exigences informelles, la formalisation des exigences non-fonctionnelles temporelles se faisant avec la relation « *derive* » (cf. Fig.33).

Les exigences formelles *Formal_Begin_Process_Duration* et *Formal_End_Proces_Duration* sont respectivement associées aux exigences fonctionnelles *Begin_Process* et *End_Process* par la relation « *satisfy* ». Cette relation représente la satisfaction d'une exigence non-fonctionnelle temporelle par un mécanisme. L'exigence formelle exprimée par TRDD est vérifiée à l'aide d'un observateur pour un mécanisme particulier du système (décrit par une exigence fonctionnelle qui sera ensuite dérivée en cas d'utilisation puis en scénario). C'est pourquoi nous établissons la relation « *satisfy* » entre une exigence non-fonctionnelle formelle et une exigence fonctionnelle. Dans la mesure où la vérification formelle repose sur une confrontation entre exigences formelle et spécification formelle du système. Ce dernier est construit à partir des exigences fonctionnelles (cas d'utilisation).

3.3. Définition formelle

Un diagramme d'exigences TURTLE étendu (noté $\mathcal{ERD}$) est un graphe 3-parties composé d'un diagramme d'exigences ($\mathcal{RD}$ voir définition 2 dans la section 2.1.2 de ce chapitre) et des fonctions « *composition* » (comp) et « *satisfy* » (sat). La relation de composition relie une exigence informelle à plusieurs exigences informelles qui composent la première. La relation « *satisfy* » lie une exigence fonctionnelle (IR avec type(IR)=fonctionnelle) à une exigence non-fonctionnelle temporelle formelle (FTR), pour représenter la satisfaction d'une exigence non-fonctionnelle par une exigence fonctionnelle.

Définition 7 : Diagramme d'exigences étendu

$\mathcal{ERD}$ = ($\mathcal{RD}$, comp, sat)

où

$\mathcal{RD}$ est diagramme d'exigences (voir définition 1)

comp: IR → IR est une fonction décrite par $\oplus$

sat: IR → FTR est une fonction décrite par <<satisfy>> si et seulement si type(IR)=fonctionnelle

Nous ne présentons pas le méta-modèle du diagramme d'exigences étendu, car les relations de composition et de satisfaction sont déjà définies dans le méta-modèle du langage SysML [SysML 06] sur lequel se base le méta-modèle du diagramme d'exigences.

4. Conclusion

La vérification formelle d'exigences temporelles est au cœur des contributions présentées dans ce mémoire. La vérification de ce type d'exigence nécessite donc de pouvoir les spécifier formellement. L'incorporation d'un langage graphique, en l'occurrence SysML, dans les diagrammes du profil TURTLE, nous a paru inévitable du fait que le cadre sémantique de TURTLE est basé sur UML.

L'approche présentée dans ce chapitre correspond à la phase de spécification des exigences (cf. chapitre III section 3.1.3) du profil TURTLE. Les contributions présentées dans ce chapitre reposent sur :

- La définition du diagramme d'exigences TURTLE, basé sur le diagramme d'exigences SysML, qui permet de définir des exigences formelles par le stéréotype « Formal_Requirement ». Ces dernières sont dérivées d'exigences informelles et vérifiées formellement à l'aide d'observateurs (que nous avons également définis dans ce chapitre).
- La définition d'un langage graphique d'exigences temporelles (TRDD), basé sur les chronogrammes et qui permet de décrire formellement une exigence présentant une occurrence de début et de fin pour qu'elle puisse être vérifiée en termes de distance temporelle. Ceci préfigure la génération automatique d'observateurs permettant de guider la vérification de l'exigence formalisée par un TRDD (cf. chapitre V).
- La définition des relations de *composition* dans les diagrammes d'exigences TURTLE pour raffiner des exigences informelles et de *satisfaction* d'une exigence non-fonctionnelle par une exigence fonctionnelle.

Les contributions présentées dans ce chapitre, permettent donc d'exprimer de manière formelle des exigences non-fonctionnelles temporelles présentant une occurrence de début et de fin et ainsi les confronter à une spécification formelle du système. Cependant, pour des projets pouvant contenir des milliers d'exigences, les langages graphiques du type SysML atteignent leurs limites. Il est donc nécessaire de construire des bases de données d'exigences [HUL 04] [RAS 03] [BUC 05]. Généralement ces dernières ne prennent pas en compte toutes les descriptions d'un diagramme d'exigences, en particulier la formalisation et les moyens mis en œuvre pour vérifier ces dernières. Une contribution future serait de construire des bases de données, contenant des descriptions formelles de type TRDD, pour pouvoir à partir de celles-ci générer automatiquement des diagrammes d'exigences SysML.

La spécification des exigences temporelles, par des diagrammes d'exigences TURTLE et des TRDD, est le point de départ pour la vérification des exigences temporelles guidée par des observateurs qui est présentée dans le chapitre suivant.

Chapitre V. Vérification formelle d'exigences temporelles sur le modèle de conception

Dans le domaine de la vérification basée sur l'analyse d'accessibilité de modèles formels, les exigences peuvent être décrites à l'aide de formules de logique ou d'observateurs greffés au modèle. Notre choix s'est porté sur la *vérification* d'exigences temporelles *guidée par observateur*. Réduisant le coût de l'outillage, cette approche fait seulement appel à des outils d'analyse d'accessibilité. A contrario, la vérification basée sur les techniques de *model-checking* demande l'utilisation d'outils supplémentaires (*model-checker*).

La contribution présentée dans ce chapitre repose sur la génération automatique d'observateurs dans le modèle de conception TURTLE à partir d'une spécification des exigences temporelles en TRDD (cf. chapitre IV section 2.2). Cette génération s'effectue en deux grandes étapes : la construction du comportement de l'observateur (à partir de la spécification de l'exigence en TRDD) et la construction du modèle observable contenant le modèle du système couplé avec l'observateur. Ce modèle observable est dérivé du modèle TURTLE de conception.

Ce chapitre est organisé de la manière suivante. La section 1 recense les différentes méthodes de vérification d'exigences temporelles tout en positionnant notre approche face aux différentes techniques de vérification formelle. La section 2 présente le processus de vérification d'exigences temporelles guidée par observateurs proposé pour le profil TURTLE. La section 3 énonce les hypothèses de fonctionnement de ce processus, caractérise le placement des *points d'observations*, élément crucial dans la *vérification guidée par les observateurs*. Cette même section identifie ce qui reste à implanter dans l'outil TTool [TTOOL]. Enfin, la section 4 conclut ce chapitre.

1. Approches de vérification des exigences temporelles

Dans la littérature associée au domaine de la vérification de systèmes temps réel basée sur l'analyse d'accessibilité de modèles formels, deux grandes approches de vérification[10] d'exigences temporelles cohabitent :

- La vérification par *contrôle de modèles* ou *model-checking*. Cette approche consiste à vérifier un modèle en termes d'assertions, par des formules de logique que l'on applique sur des graphes représentant l'exécution du système [CLA 99]. Le résultat de la vérification est une réponse de type oui/non à chacune des assertions. L'intérêt de cette approche repose sur le fait que la vérification du modèle se fait sans modifier le comportement du système. Cependant ces formules sont souvent très difficiles à manipuler pour un « non-initié ».
- La vérification *guidée par des observateurs*. Les *observateurs*, tels que décrits par exemple en [JAR 88] sont des objets externes à ceux qui composent le système et dédiés à la vérification des exigences. Ils sont construits au sein du modèle. On construit des *observateurs* qui interagissent

[10] L'approche de « *Theorem proving* » n'est pas décrite dans ce mémoire car elle n'appartient pas à cette catégorie.

avec le système sans le bloquer intempestivement (notion d'observateurs « non-intrusifs »). L'intérêt est de construire un observateur exprimé dans le même langage que le modèle du système. Plus souple dans le pouvoir d'expression que les formules de logique, cette approche n'en demeure pas moins difficile à mettre en œuvre dans le modèle par le fait qu'un *observateur* ne doit pas être intrusif durant l'exécution du modèle à observer [JAR 88].

1.1. La vérification par contrôle de modèles

La vérification par *contrôle de modèle* ou *model-checking* consiste à vérifier si un modèle formel donné, le système ou une abstraction du système, satisfait une propriété en parcourant le modèle formel (en l'occurrence un ou plusieurs graphes) représentant l'exécution du système. Ces propriétés (qui correspondent à des exigences) peuvent être formulées sous forme de logique temporelle [HUT 04].

On distingue deux grandes familles de techniques de *model-checking* basées sur deux types d'algorithmes :

- Les algorithmes locaux qui n'explorent que les parties du système contribuant à la solution, mais en évaluant toutes les sous-formules de la propriété. Ces techniques reposent sur la formulation des propriétés en *logique temporelle linéaire*. L'exemple le plus répandu de cette logique est LTL (*Linear Temporal Logic*) [VAR 86] qui est utilisé dans le model-checker SPIN [HOL 03] supporté par le profil UML PROMELA [MAG 02] (cf. chapitre VI) mais aussi dans le model-checker de l'outil UPPAAL [UPPA] sur lequel repose les travaux de [JUR 02] (cf. chapitre VI). Il existe aussi des extensions à LTL. Par exemple, se-LTL [CHA 04] est adapté à la représentation état/événement (réseaux de Petri temporels par exemple) et est utilisé dans l'outil TINA [BER 05] correspondant au volet vérification du profil UML MARTE [OMG 07] (cf. chapitre II).
- Les algorithmes globaux qui évaluent toutes les sous formules sur tous les états du système et opèrent par récurrence sur la structure syntaxique de la propriété. Ces techniques reposent sur la formulation des propriétés en *logique temporelle arborescente*, dont la plus connue est CTL (*Computational Tree Logic*) [CLA 86]. Les logiques modales comme le *μ-calcul propositionnel* [STI 96] [BRA 06] reposent aussi sur les algorithmes locaux. Des extensions temporisées de CTL issues des logiques modales prennent en compte l'ordre entre évènements et leur distance temporelle [CLA 99] comme TCTL (Timed CTL) [ALU 89]. TCTL est utilisé dans les travaux de [BRA 05] sur la description visuelle de scénarii en formules de logique (cf. chapitre IV). Ce type de logique ne nécessite pas de construire des observateurs permettant d'augmenter le pouvoir d'expression des exigences temporelles (nous reviendrons sur ce problème dans le paragraphe suivant). Cependant la formulation des exigences en logique modale nécessite une très bonne connaissance dans le domaine et est particulièrement difficile pour un « non-initié » [ROG 06].

L'approche de vérification basée sur le *model-checking* est la méthode de vérification d'exigences la plus couramment utilisée car c'est une approche prouvée et entièrement automatisable (approche dite « presse bouton »). La construction de formules de logique ne demande pas de preuve et permet d'exprimer formellement l'ordre des événements. Cependant une mauvaise formulation de l'exigence conduit à un mauvais diagnostic du *model-checker*. De plus, ces formules sont généralement plus difficiles à mettre en œuvre quand la définition des exigences fait intervenir des

« distances temporelles » entre événements [TOU 97] (mis à part les logiques modales [STI 96] [BRA 06] et extensions temporisés de CTL [ALU 89]). Pour palier ce problème, on en vient à construire généralement des observateurs temporels dans le système et à vérifier l'ordre entre les événements à observer et ceux déclenchés par l'observateur. Par exemple, les travaux de [MAL 06] concernent la vérification d'exigences non-fonctionnelles temporelles dans le profil MARTE [OMG 07] en surchargeant le réseau de Petri de TINA [BER 05] par des places fantômes, qui n'interfèrent pas sur le comportement du système, pour pouvoir vérifier des exigences temporelles à l'aide de formules se-LTL [CHA 04].

L'utilisation des méthodes de *model-checking* demande aussi de se doter d'outils permettant d'exprimer les exigences sous forme de logique mais aussi d'un *model-checker* pour les vérifier sur les traces d'exécutions. Ce n'est pas encore le cas, par exemple, de l'outil TTool [TTOOL] du profil TURTLE qui permet faire de l'analyse d'accessibilité mais n'a pas de *model-checker*.

Enfin, ces méthodes ne permettent pas généralement de réutiliser des spécifications des exigences (formules de logique) dans les autres phases de validation (simulation, tests). Au contraire, l'approche guidée par les observateurs réside aussi dans le caractère réutilisable de la spécification des exigences dans les autres phases de validation. Les objets observateurs, exprimés dans le même langage que le modèle (définis par exemple en UML), peuvent être prototypés ensuite dans le langage correspondant à la phase de déploiement/mise en œuvre (notion de *portabilité*). Ce dernier aspect sera développé dans les perspectives.

1.2. Vérification guidée par les observateurs

Le concept d'observateur est très largement utilisé dans le processus de validation qu'il s'agisse de vérification [DHA 07] [GRA 05], simulation [JAR 88] [DOL 03] ou de test (cet aspect n'est pas traité dans ce mémoire).

1.2.1. Mise en œuvre d'observateurs dans le simulateur Véda

[JAR 88] présente la mise en œuvre des observateurs en Estelle [IS 9074] de systèmes distribués dans le simulateur *Véda*. Celui-ci simule des systèmes « clos » (sans interactions avec l'extérieur) ce qui implique de modéliser également l'environnement dans le modèle décrit en Estelle.

En [JAR 88], un observateur a la possibilité de collecter toutes les informations du système (les interactions entre les différents modules et les états respectifs de ces derniers). Il peut aussi stopper le programme afin d'effectuer des traitements avancés sur les données observées. Une exigence correspond à un observateur. Ce dernier est spécifié dans une variante d'Estelle car :

- Les machines à états sont bien adaptées au principe de l'observation.
- Le compilateur existe déjà sur Véda ce qui limite le coût d'outillage pour faire de la vérification d'exigences.
- Le langage est connu d'un utilisateur de *Véda*.

Les observateurs sont de bons candidats pour la mesure de performances temporelles mais aussi pour collecter des informations et produire une analyse statistique. Il est aussi possible d'utiliser plusieurs observateurs en parallèle.

La proposition dans [JAR 88] est, à notre connaissance, pionnière dans ce domaine et couvre les aspects conceptuels jusqu'à l'implémentation. Le principe de stopper le programme par l'observateur nous a fortement inspiré pour la conception d'observateurs dont l'analyse est intrusive en interrompant l'exécution du système lorsqu'une exigence temporelle de niveau de criticité haute est violée. Cependant, aucune méthodologie automatisable n'est proposée en [JAR 88], les observateurs étant construits à la « main ». Au contraire, l'approche proposée en section 2 présente la construction d'observateurs telle qu'on peut l'automatiser à partir de la spécification des exigences exprimées au moyen de diagrammes d'exigences et de TRDD (cf. chapitre IV).

1.2.2. Observateurs de machines synchrones

Les travaux de [HAL 93] traitent de manière générale la vérification de propriétés de sûreté, à l'aide d'observateurs sur des programmes réactifs et synchrones. Un programme réactif et synchrone réagit instantanément et de manière déterministe aux événements de son environnement (le temps est donc représenté sous forme *logique*). Ces observateurs, déterministes et synchrones sont capables d'exprimer des propriétés de sûreté et des hypothèses sur l'environnement du système.

Les programmes doivent être décrits par une algèbre de processus synchrone. La communication interprocessus est basée sur le concept de diffusion (un émetteur et plusieurs récepteurs). Elle est non-bloquante pour l'envoi et synchrone sur la réception. Cette communication asymétrique est nécessaire à l'utilisation d'observateurs qui, connectés au système, sont synchronisés sur la réception sans changer la description de ce dernier.

[HAL 93] propose un cadre théorique complet pour les observateurs de machines synchrones. Une exigence est vue comme un ensemble de traces qu'il est difficile d'exprimer à partir de la formulation des exigences. De surcroît, le temps y est exprimé de manière qualitative. A contrario, les travaux présentés dans ce mémoire introduisent un langage « simple » pour formuler ces exigences en représentant le temps de manière quantitative. Notons que dans [HAL 93], les observateurs ne se limitent pas qu'à l'expression des exigences. Ils prennent aussi en compte la réduction de l'explosion combinatoire par réduction des comportements de l'environnement à un sous ensemble complet et cohérent. Au contraire, notre approche consiste plutôt à distinguer les aspects « observations » et « limitation de l'explosion combinatoire » (cf. chapitre III section 3.1.2).

1.2.3. Observateurs pour les systèmes auto-validés

Les observateurs dans [DIA 94] sont vus comme un concept permettant la création de systèmes distribués auto-validés. Un système auto-validé est composé du système à observer et d'un observateur, possédant un comportement redondant par rapport à l'implémentation. L'observateur correspond à une implémentation différente du système, capable d'observer des disfonctionnements par des mécanismes d'espionnages et de contrôle du système. Il est décrit dans le même langage de spécification que le système, afin d'en simplifier la description.

L'observateur est un sous-système formel décrit pour vérifier des comportements distribués. Sa conception est indépendante de celle de l'implémentation. Un système est auto-validé si et seulement si ce dernier est composé d'une spécification formelle du système et d'un modèle formel de l'observateur. Le langage utilisé pour construire un tel système doit être capable de supporter des procédures de vérification et d'inclure des mécanismes d'espionnage. Dans [DIA 94], les

spécifications du système et de l'observateur sont formulées par des réseaux de Petri [MUR 89], formalisme qui permet d'exprimer les caractéristiques précédemment citées.

Cette technique permet de surveiller les défaillances de systèmes pendant leur exécution. Cette approche est orientée test d'implantation. Cependant elle ne garantit pas l'absence d'erreur par le fait que l'observateur est une duplication du système observé (duplication des erreurs dans les deux modèles). [DIA 94] établit un ensemble de concepts sur les observateurs en termes *d'espionnage* et de *contrôle*, qui ont inspiré nos travaux au niveau de l'intrusivité des observateurs dans le projet SAFECAST (cf. chapitre VI).

1.2.4. Le langage GOAL

Le langage d'observation GOAL (*Geode Observation Automata Language*) [OBE 99] [DOL 03] a été défini et mis en œuvre dans l'outil *ObjectGeode*. C'est un outil de conception, simulation et validation de modèles spécifiés en langage SDL [SDL 99].

Les observateurs GOAL fonctionnent sur le principe suivant. Pour une action exécutée dans le système, l'observateur peut en exécuter plusieurs. Ce principe permet d'assurer à l'observateur une surveillance sur le système. Les observateurs sont composés au système lors des simulations aléatoires ou exhaustives.

Les observateurs GOAL sont des automates SDL spécifiques. Ils possèdent une construction supplémentaire : notion de *matching* [DOL 03] pour pouvoir être associés aux objets observés. Des états de succès et/ou de rejet peuvent être déclarés. Suivant la simulation choisie et la présence des états spéciaux, le verdict peut être interprété de différentes façons :

- Pour une simulation aléatoire, l'observateur peut donner un résultat du type "ok", "nok" ou « inconclusive » (dans ce dernier cas, l'observateur n'a pas pu diagnostiquer la satisfaction ou non de l'exigence).
- Pour une simulation exhaustive, la propriété peut être vérifiée ou non. Des outils de diagnostic, de traces, de surveillance de l'état du système accompagnent ce processus de vérification.

Les observateurs GOAL sont un exemple réussi de l'intégration d'un processus de vérification dans un outil industriel [DOL 03]. Le concept de *matching* nous a inspiré pour définir la notion de *points d'observations* (cf. section 2). Cependant [DOL 03] ne propose aucune solution pour la « rédaction des observateurs ». Au contraire, notre approche repose sur la génération automatique d'observateurs à partir d'une spécification des exigences temporelles en TRDD (cf. chapitre IV section 2.2).

1.2.5. Les observateurs du profil UML OMEGA

Les travaux de [GRA 05] sont menés dans le cadre du projet OMEGA, présenté dans le chapitre II, sur la vérification de modèles UML temporisés. L'idée principale est de proposer à l'utilisateur le moyen de spécifier des propriétés sous forme d'observateurs en langage UML. La vérification de modèles UML est effectuée par simulation et vérification, en traduisant des modèles UML en IF (cf. chapitre II section 4.1.4).

Les observateurs sont des classes stéréotypées par « observer » que l'on ajoute au système. Chacune de ces classes possède une machine à états. Certains de ces états sont stéréotypés par *success* ou *reject*. On distingue deux types d'observations :

- de sûreté, avec des observateurs classiques (analyse d'accessibilité d'états de rejet) ;
- de vivacité, avec des automates de *Büchi* [VAR 86] (analyse des composantes connexes).

Les travaux de [GRA 05] permettent d'envisager une méthodologie d'exploitation des observateurs dans un formalisme de haut niveau (tout comme [DHA 07] voir section 1.2.7). Néanmoins ces travaux, à notre connaissance, ne sont pas encore associés à un dispositif de génération automatisée d'observateurs.

1.2.6. Méthodes de surcharge

Les travaux présentés en [TOU 97] ne traitent pas explicitement les observateurs mais l'approche est identique : surcharger le modèle formel du système (des réseaux de Petri temporels [BER 91]) par des places et des transitions supplémentaires exprimant l'exigence à vérifier. Ces « observateurs neutres » ne perturbent pas le comportement du système spécifié. La satisfaction de l'exigence est alors démontrée par les recensements ou non des tirs de certaines transitions exprimées dans la surcharge (qui sont des places notés par OK ou NOK reflétant la satisfaction ou non de l'exigence).

Les exigences vérifiées dans cette approche concernent les dates d'occurrence des événements de l'application qui s'expriment sur le modèle par des dates de tirs d'occurrences dans le réseau de Petri. Dans cette approche on distingue quatre types d'exigences exprimés ensuite sous forme de patterns de réseaux de Petri temporels :

- Date d'occurrence d'événements.
- Intervalle de temps entre les occurrences successives d'un événement.
- Intervalle de temps entre deux événements causaux.
- Simultanéité d'un ensemble d'événements.

L'approche présentée en [TOU 97] permet d'exprimer des exigences temporelles non-fonctionnelles quantitatives qui se rapprochent de nos travaux. Ces travaux nous ont fortement inspiré pour exprimer les notions d'intervalles quantitatifs temporels d'événements. Notre approche ne traite pas des exigences liées à la simultanéité, mais est exprimée dans un langage graphique de haut niveau permettant de générer automatiquement des observateurs alors que [TOU 97] les exprime sous forme d'un motif surchargeant le réseau de Petri représentant le système.

1.2.7. Diagrammes de contexte et d'observation

[DHA 07] présente un travail en cours concernant la définition et l'exploitation d'un concept d'unité de preuve comme élément encapsulant toutes les données nécessaires à la preuve d'une propriété sur un modèle plongé dans un environnement.

Les modèles des unités manipulées dans le processus de développement, peuvent être construits, sous la forme d'automates, à partir des données fournies par les phases d'analyse et de conception du système. Ces modèles intègrent des *contextes de preuve abstraites*. Ceux-ci modélisent formellement une exigence (ou une composition d'exigences) de sûreté ou vivacité bornée qui doit

être vérifiée pour un modèle de système donné dans un contexte comportemental spécifique. Les contextes de preuve sont ensuite traduits par transformation en modèles concrets puis en langage IF par des outils d'analyse formelle. Cette approche est expérimentée sur une implantation des unités de preuve intégrant une description des contextes de preuve dans un langage nommé CxUCC (cf. chapitre IV) et exploités par un outil OBP (Observer-Based Prover) mettant en œuvre le langage IF et une technique de vérification par observateur.

Contrairement aux approches citées dans les paragraphes précédents, cette approche a le mérite de contenir des éléments méthodologique pour construire correctement les observateurs [ROG 06]. Ces derniers servent également à décrire la notion de contexte simulant l'environnement du modèle. Tout comme dans notre approche, un lien est fait entre un langage graphique de description des exigences (CxUCC pour [DHA 07] et TRDD pour notre approche) et génération d'observateurs. Cependant, deux aspects diffèrent : d'une part, dans notre approche, la notion de contexte est dégagée des observateurs (cf. chapitre III section 3.1.2) ; d'autre part, la spécification des exigences non-fonctionnelles temporelles se fait après la définition des fonctionnalités dans [DHA 07] (approche orientée scénarii). Alors que dans notre approche nous spécifions les exigences avant la construction d'un diagramme de cas d'utilisation en charge de définir les fonctionnalités. Notons que l'approche présentée en [DHA 07] et principalement les travaux antérieurs [ROG 06] nous ont servi de référence pour la définition de la taxonomie des exigences temporelles et des dispositifs d'intégration des observateurs dans le modèle.

2. Génération d'observateurs TURTLE à partir d'exigences temporelles

Cette section présente le dispositif de génération automatique d'observateurs à partir d'une spécification des exigences temporelle en TRDD (cf. chapitre IV section 2.2). Cette génération s'effectue en deux grandes étapes :

- La construction du comportement de l'observateur définie à partir de la spécification de l'exigence temporelle en TRDD.
- L'insertion de l'observateur dans le modèle TIF (d'analyse ou de conception) déduit à partir des informations du diagramme d'exigences.

2.1. Construction d'un observateur TURTLE

2.1.1. Définition d'un observateur TURTLE

Un observateur est une classe TURTLE stéréotypée par « *TObserver* » ayant les mêmes caractéristiques qu'une classe TURTLE. Le nom de l'observateur est défini dans le diagramme d'exigences (*Name_Obs*). Les attributs d'un observateur correspondent aux dates définies dans les frontières temporelles du TRDD (*Time_Temporal_Frontier*). On distingue trois types de portes de communication :

- *Connexions des actions figurant dans le TRDD au modèle*. Ces portes de synchronisation sont appelées par la suite *points d'observations* (*OP*). Les synchronisations entre observateurs et objets observés sont unidirectionnelles : elles vont des objets observés vers l'observateur pour prélever dans le système les différentes informations logiques correspondant aux exigences temporelles présentes dans le TRDD. Il existe deux types de *points d'observations* (pouvant être synchronisées avec différents objets observés) : l'action « *Start* » déclenchant la capture d'exigence et l'action « *Capture* » déclenchant le diagnostic de l'exigence par l'observateur.

- *Traçabilité des exigences.* Cette porte sert à l'exploitation de la vérification de l'exigence décrite par le TRDD. On génère la porte de violation d'exigence (VA) par une étiquette définie dans le diagramme d'exigence présenté dans le chapitre IV (attribut *Violated_Action*). Si l'exigence est satisfaite, l'observateur ne se manifeste pas pour ne pas perturber le comportement du système.
- *Contrôle de l'exécution du modèle.* L'arrêt du système est envisagé si une exigence de niveau de criticité haute est violée (cf. chapitre IV section 2.1). Par contre le système n'est pas interrompu si l'exigence violée est de niveau de criticité bas ; le résultat de la violation d'exigence figure sur la matrice de traçabilité. Dans le cas de violation d'une exigence haute, il faut construire des portes de communication (ST composé des portes stop[i]) entre l'observateur et tous les objets actifs du système pour arrêter l'exécution du système en préemptant les différents objets.

Le comportement interne de l'observateur est décrit par un diagramme d'activités directement généré à partir du TRDD (cf section 2.2).

Définition 8 : Observateur

TObserver= (Nom_Obs, Attributs_Obs, Gates_Obs, AD_Obs)
où
Name_Obs = Nom (voir définition 5)
Attributs_Obs est un ensemble de valeurs de frontières temporelles (Temporal_Frontier voir définition 6) représenté par des entiers
Gates_Obs Є {OP, VA, ST}
avec OP = <Start, Capture> (voir définition 6)
VA Є Label_of_Violated_Action (voir définition 5)
ST Є {Stop[1], … , Stop[nb_TC]} où nb_TC représente le nombre de Tclasses actives
AD_Obs = voir section 2.3

2.1.2. Méta-modèle UML d'une Tclass observateur

La Fig.41 montre le méta-modèle de la classe stéréotype « *TObserver* ». Un observateur est une Tclass particulière (héritage de la classe *::metamod_Tnative ::TClass*).

Il est composé, en plus des éléments présentés dans le chapitre IV (cf. définition 5 et Fig.34 section 2.1 du chapitre VI) :

- D'un ensemble d'attributs (classe *Attributes_Obs*) qui correspondent aux valeurs des frontières temporelles du TRDD (classe *Date_of_FT*).
- D'au-moins trois portes de communication (classe stéréotypée « interface » *Gate_Obs*) composée de trois types de portes, qui sont :
  - Deux portes d'observation (classe *Observation_Gates*) qui se synchronisent avec les portes correspondantes aux points d'observations du TRDD.
  - Un ensemble de portes de contrôle d'exécution pour arrêter l'exécution du système (classe *Execution_Control*) dépendant du niveau de criticité de l'exigence temporelle formelle, comme le montrent l'association *generated_if_risk_is_high* et la formule OCL qui décrit cette association.

- Une porte de notification de violation d'exigence (classe *Violated_Gate*) qui correspond à l'étiquette définie dans le diagramme d'exigence (voir section 2.1.3).

- D'un diagramme d'activités (classe *TAD_Obs*) construit à partir du TRDD (le méta-modèle du diagramme d'activités d'un observateur est montré dans la section suivante).

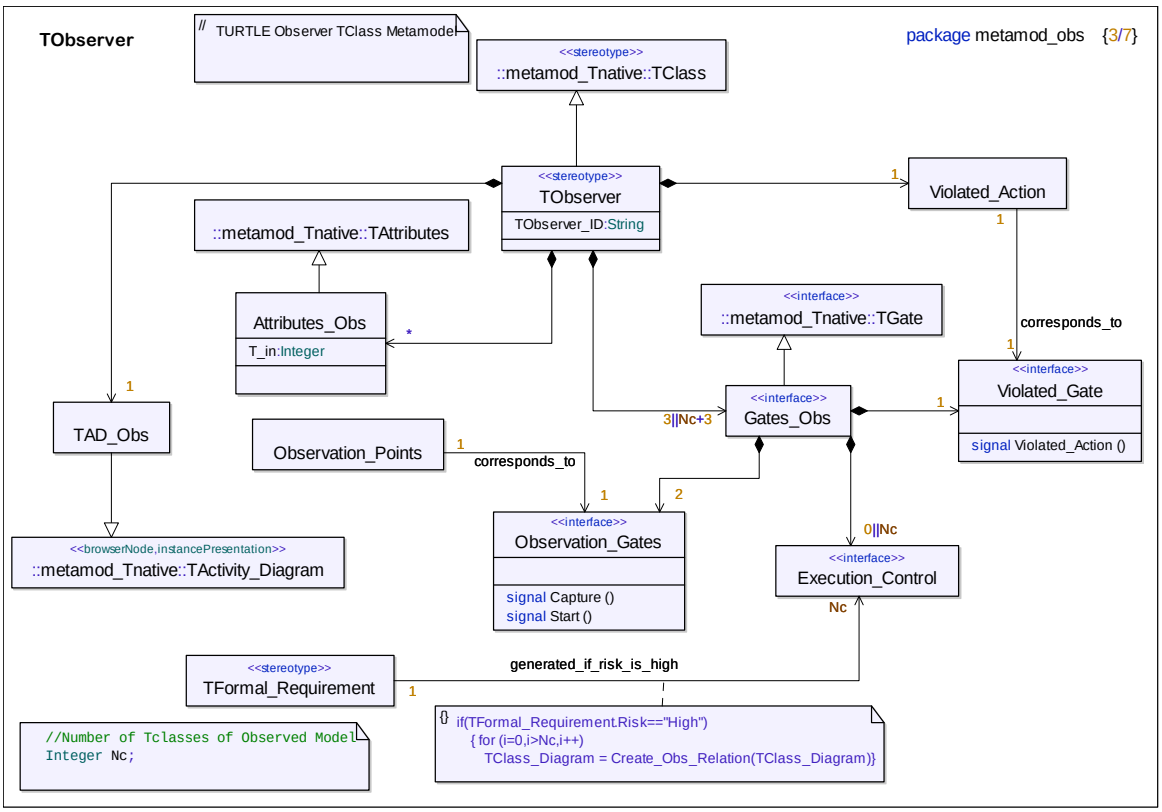

Fig.41. Méta-modèle d'une classe stéréotypé « TObserver »

2.2. Vue d'ensemble

La construction d'un observateur nécessite trois étapes (cf. algorithme 1). Dans un premier temps, il faut isoler les différentes étiquettes du TRDD (Partie_OK et Partie_KO) et les informations temporelles sur les frontières temporelles (`Tin`) (voir algorithme 2). Ensuite un embryon du comportement de l'observateur est construit à partir des informations collectées dans les deux tableaux (voir algorithme 3). Dans un premier temps, cet embryon (`Obs_label`) est composé uniquement d'actions de synchronisation « *Capture* » et d'offres limitées dans le temps.

Ensuite, à partir de `Obs_label`, le diagramme d'activités final de l'observateur est construit en respectant deux règles qui concernent :

- Le début et terminaison du diagramme d'activités d'un observateur (voir algorithme 4). L'observateur ne doit en aucun cas bloquer les objets observés, ni modifier leur comportement. .

- L'interruption de l'exécution des objets observés si une exigence haute est violée ; ceci concerne donc la modification d'une part du diagramme d'activités de l'observateur (voir algorithme 4) et d'autre part des comportements des objets observés (voir algorithme 5).

Algorithme 1 Construction du diagramme d'activités d'un observateur

```
String generation_DA_obs_TRDD(int[] TRDD, int[] T_in,int crit,int nb_TC, String VA)
   {
         int[] label;
         int[] T_out;
         String Obs_label = "";
         //Verification de la syntaxe du TRDD (cf. annexe A.1)
         if (verif_syntax_TRDD(TRDD,T_in)==1)
               write("erreur construction du TRDD : generation impossible");
         //Si TRDD correct
         else {
               // Algorithme 2 Traduire le TRDD en labels OK ou KO
               label = Build_transtab(TRDD,T_in);
               //Verifier semantique du TRDD (cf. annexe A.2)
               if (verif_semantic_TRDD(label)==1)
                     write("Attention erreur semantique TRDD");
               else write("semantique TRDD correcte");
               // Algorithme 3 Convertir le temps absolu en temps relatif
               T_out= Build_transtime(T_in);
               // Algorithme 4 Generation de l'embryon du DA de l'observateur
               Obs_label=Build_Obs_behavior_labels(label,T_out);
               // Algorithme 5 Generation du DA complet de l'observateur
               Obs_label=Build_complete_Obs_behavior(label,T_out,crit,nb_TC,VA);
               }
               return Obs_label;
   }
```

Glossaire: `TRDD` = tableau d'entiers représentant les éléments de la ligne de vie du TRDD
`T_in` = tableau d'entiers contenant les valeurs des dates temporelles du TRDD
`crit` = niveau de criticité de l'exigence (0 = bas et 1 = haut)
`nb_TC` = nombre de Tclasses du modèle observé
`VA` = étiquette de violation d'exigence définie dans le diagramme d'exigences représenté par une chaîne de caractères
`label` = tableau contenant les éléments de description des exigences temporelles (OK ou KO sous forme de tableau d'entiers
`T_out`= tableau contenant les informations temporelles converties en temps absolu
`Obs_label` = Diagramme d'activités de l'observateur représenté par une chaîne de caractères

La syntaxe et la sémantique du TRDD sont définies par des règles de grammaire et des méta-modèles (cf. chapitre IV section 2.). Les algorithmes de vérification de la syntaxe et de la sémantique (respectivement `verif_syntax_TRDD et verif_semantic_TRDD`) sont présentés en annexe A.

2.3. Traduction d'un TRDD vers le diagramme d'activités de l'observateur

2.3.1. Tables de traductions

Le Tab.9 illustre pour toutes les constructions des descriptions d'exigences, la sémantique associée par transposition en diagramme d'activités TURTLE [APV 04] (cf. chapitre II section 4.1.2).

Soit ***TRDD*** la dénomination pour une description d'exigence, ***T(TRDD)*** la traduction en diagramme d'activités correspondant à la traduction de ***TRDD*** et ***τ(TRDD)*** la traduction textuelle en fonction des labels définis dans le chapitre II (cf. Tab.2).

Label du TRDD	Symbole Graphique	Traduction en diagramme d'activités de l'observateur	Traduction textuelle
Begin	< ***TRDD***	Start ***T(TRDD)***	`Begin ;` `τ(TRDD) ;`
T_F (Tin)	***TRDD1*** >|< ***TRDD2*** Tin	Capture T out ***T(TRDD2)*** ***T(TRDD1)***	`TLO (`T_{out}`, Capture,` `τ(TRDD1), τ(TRDD2)) ;`
OK_Part	***TRDD*** **OK**	***T(TRDD)***	`τ(TRDD) ;`
KO_Part	***TRDD*** **KO**	***T(TRDD)*** Violated_Action	`τ(TRDD) ;` `Violated_Action ;`
End	***TRDD*** >	***T(TRDD)*** Capture	`τ(TRDD) ;` `Capture ;`

Tab.9. Table de traduction des descriptions d'exigences temporelles TURTLE en diagramme d'activités TURTLE

2.3.2. Méta-modèles UML du diagramme d'activités d'un observateur

La Fig.42 décrit le diagramme d'activités d'un observateur (classe *TAD_Obs*) tel qu'on le construit à partir du TRDD ; ceci est montré par l'association *is_buid_upon* entre les classes *TAD_Obs* et *TRDD*.

Le diagramme d'activités de l'observateur est composé de :

- D'une action de début (classe *::metamod_Tnative ::Start_AD*), comme dans un diagramme d'activités TURTLE en TIF, correspondant au symbole de début du TRDD (classe stéréotypée par « icon » *Begin*).
- D'au-moins une action de fin (classe *::metamod_Tnative ::Stop_AD*) comme dans un diagramme d'activités TURTLE classique.
- D'un ensemble de connecteurs (classe *::metamod_Tnative ::Connector*) de type choix (classe *::metamod_Tnative ::Choice*), jonction (classe *::metamod_Tnative ::Junction*) et boucle (classe *::metamod_Tnative ::Loop*).

- D'un ensemble d'actions de synchronisation (classe *::metamod_Tnative ::Synchronisation_Action*) qui correspondent soit aux portes d'observations (classe *Observation_Points*) qui seront synchronisées avec les portes des TObjets correspondant aux points d'observations (classes *::metamod_Tnative ::TGate*), soit à la porte de notification de violation d'exigence (classe *Violation_Gate*).
- De un à N-1 (pour N zones temporelles de satisfaction/violation d'exigence dans le TRDD cf. chapitre III) offres limitées dans le temps (classe *::metamod_Tnative ::Time_Limited_Offer*) qui correspondent aux frontières temporelles du TRDD et dont l'élément de synchronisation *Capture* est synchronisé avec la porte de communication correspondant au point d'observation du symbole *Capture_Action*.

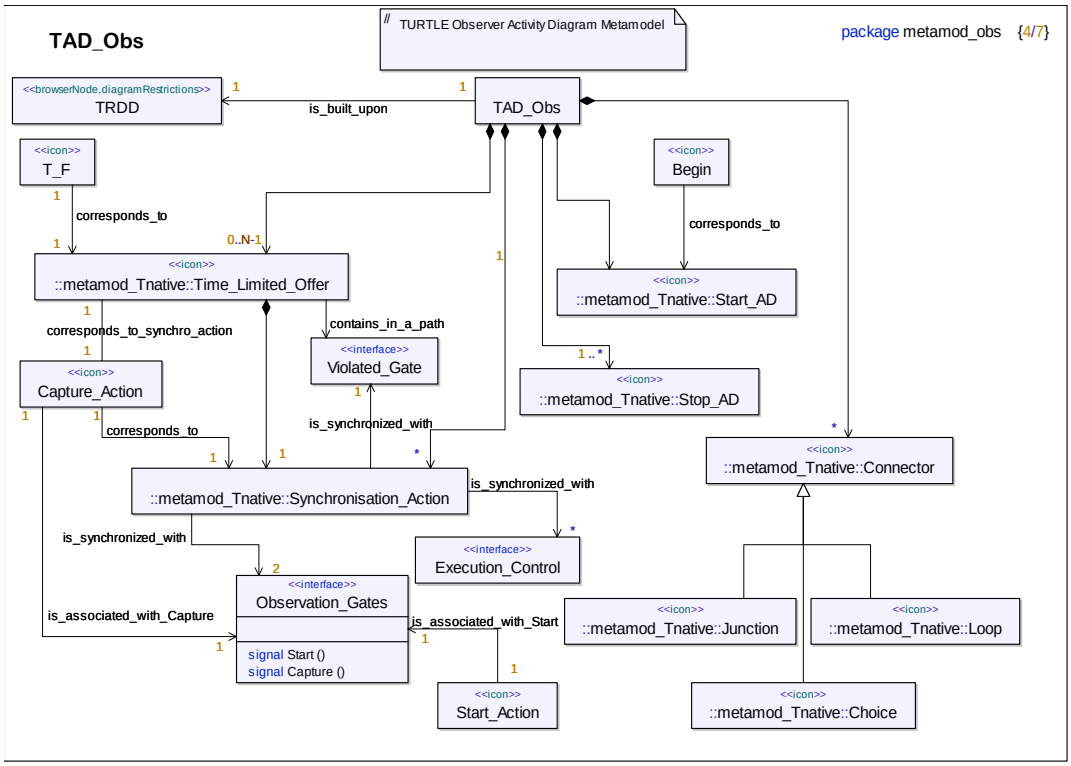

Fig.42. Méta-modèle du diagramme d'activités d'un observateur

La Fig.43 décrit les règles de construction du diagramme d'activités d'un observateur.

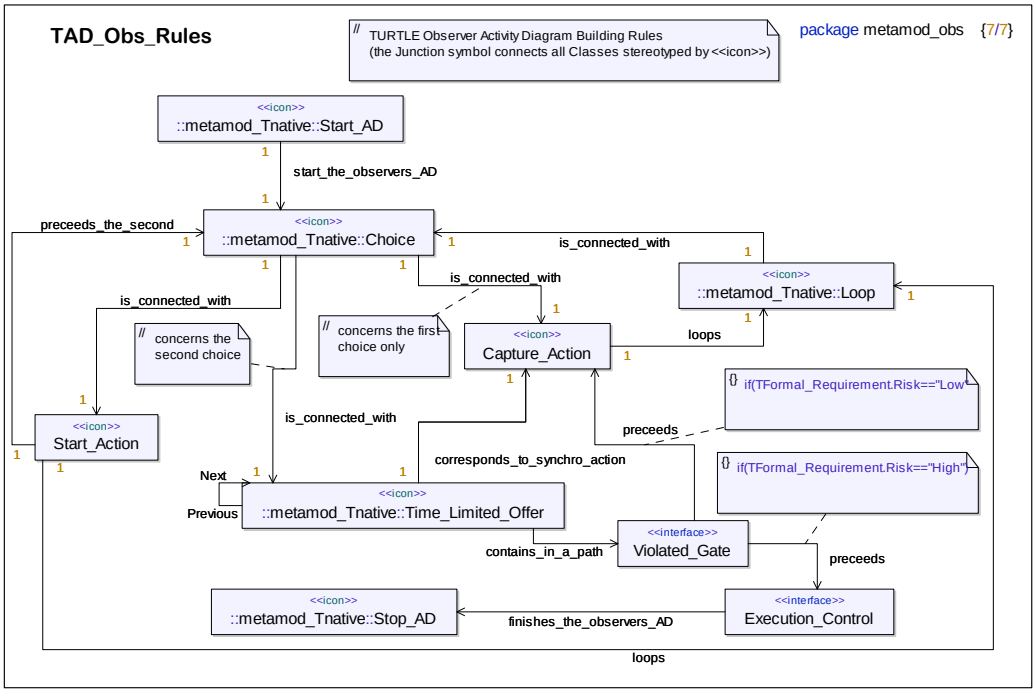

Fig.43. Méta-modèle des règles de construction du diagramme d'activités d'un observateur

2.3.3. Algorithmes de construction du comportement de l'observateur

L'algorithme 1 (cf. section 2.2) montre que la génération s'effectue en deux étapes :

- Construction du tableau correspondant aux informations des différents éléments du TRDD (`Label`) et conversion des dates des frontières temporelles (`T_out`). Pour pouvoir capturer les dates de frontières temporelles correctement, qui sont exprimées en temps absolu, il faut traduire ces dates en temps relatif par rapport à la date précédente.
- Construction du diagramme d'activités de l'observateur sous sa forme de base `Obs_Label`. Cette forme de base est directement traduite à partir les éléments des tableaux Label et Tout. Pour cela on assemble l'observateur « à l'envers » ; on commence par la fin du tableau.

La première étape est une étape de lecture du TRDD. On regarde chaque frontière temporelle. L'étiquette précédant la frontière temporelle est stockée dans le tableau Label qui est indicé par chaque frontière temporelle. Il faut aussi convertir les dates des frontières temporelles (Cf algorithme 3) qui sont exprimées en temps absolu en dates relatives ; on construit donc le tableau `T_out` en respectant cette règle.

Algorithme 2 Construction des tableaux dérivés des TRDD

```
int[] Build_transtab(int[] TRDD_i, int[] T_in) {
          //Construction du tableau en fonction du nombre de frontieres temporelles
          int[] label= new int[T_in.length+1];
          int ind=0;
          for (int i =0 ; i<TRDD_i.length ; i++)
                //Test de frontiere temporelle ou caractere de fin
                {if ((TRDD_i[i] == 1)||(TRDD_i[i] == 4)) {
                            if (TRDD_i[i-1]==2) label[ind]=1; //label OK = 1
                                  else label[ind] = 0; //label KO = 0
                            ind++;
                      }
                }
          return label;
    }
```

Glossaire: `TRDD_i` = tableau d'entiers représentant les éléments de la ligne de vie du TRDD
`T_in` = tableau d'entiers contenant les valeurs des dates temporelles du TRDD
`ind` = nombre de frontières temporelles dans la description de l'exigence compteur pour la génération du tableau label
`label` = tableau contenant les éléments de description des exigences temporelles (OK ou KO sous forme de tableau d'entier)

Algorithme 3 Conversion temps absolu/relatif pour la génération des observateurs

```
int[] Build_transtime(int[] T_in) {
          int[] T_out=new int[T_in.length];
          int prec=0;
          for (int i =0 ; i<T_in.length ; i++)
                                    {
                                          T_out[i]=T_in[i]-prec;
                                          prec=T_in[i];
                                    }
                      return T_out;
    }
```

Glossaire: `T_in` = tableau d'entiers contenant les valeurs des dates temporelles du TRDD exprimées en date absolues
`T_out`= tableau contenant les informations temporelles converties en temps absolu pour la génération du diagramme d'activités de l'observateur

A partir des tableaux `Label` et `T_out`, un embryon de comportement de l'observateur est construit ; il est formé uniquement d'actions de synchronisation et d'offres limitées dans le temps. L'observateur est construit à l'envers ; nous partons donc de la fin des tableaux Label et T_{out} pour, au fur et à mesure, ajouter des offres limitées dans le temps jusqu'à atteindre la première frontière temporelle et avoir la forme de base de l'observateur `Obs_Label`. Cette étape est décrite par l'algorithme 4.

Algorithme 4 Génération de l'observateur sous sa forme de base

```
String Build_Obs_behavior_labels(int[] label, int[] T_out) {
          //On commence par la fin donc dernier caractere
          String Obs_label = "Capture;> ";
          if (label[label.length-1]==0) Obs_label = ("<Violated_Action; "+ Obs_label);
                else Obs_label=" <"+ Obs_label;
                System.out.println(Obs_label);
          for (int i =label.length-2 ; i>=0 ; i--) {
                if (label[i]==0) Obs_label = "Violated_Action;> " + ", " + Obs_label;
                else Obs_label= "> " + ", " + Obs_label;
                //Construction d une offre limite dans le temps TLO
                Obs_label= "TLO("+T_out[i]+ ", Capture, " + "<"+Obs_label+ ")";
                //Affichage traduction format intermediaire
                System.out.println(Obs_label);
                            }
    return Obs_label;
    }
```

Glossaire: `label` = tableau contenant les éléments de description des exigences temporelles (OK ou KO sous forme de tableau d'entier)
`T_out`= tableau contenant les informations temporelles converties en temps absolu pour la génération du diagramme d'activités de l'observateur
`Obs_Label` = Diagramme d'activités de l'observateur représenté par une chaîne de caractères

2.3.4. Exemples

Dans cette partie, nous proposons différents exemples pour montrer ce qui peut être exprimé dans un TRDD et comment l'embryon du comportement de l'observateur est généré. Ces exemples sont illustrés par la Fig.44. Ils sont déduits de la Fig.7 du chapitre IV, soit :

- En a) les exigences contenant une frontière temporelle, qui doivent se terminer/commencer à T unités de temps. Ces exigences correspondent respectivement aux propriétés de *délais maximum/minimum*.
- En b) les exigences contenant deux frontières temporelles, qui doivent être comprises/exclues dans/de l'intervalle [T_1 ; T_2].

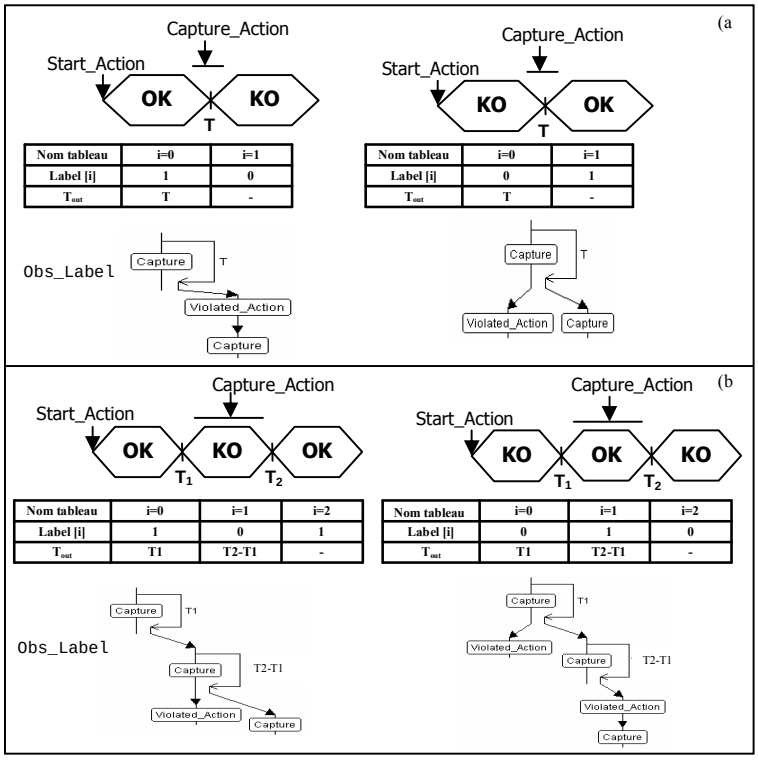

Fig.44. Application des algorithmes 2, 3 et 4 aux exemples de la Fig.39 (chapitre IV page 68)

2.3.5. Algorithmes de construction du comportement complet de l'observateur

Pour construire correctement le diagramme d'activités d'un observateur, il faut respecter deux règles :

- Un observateur doit rester passif durant la phase d'observation [JAR 88]. Il ne doit donc pas bloquer les objets observés. Pour respecter cette règle, il faut envisager un mécanisme pour ne pas bloquer l'observateur si les actions observées apparaissent dans le désordre (par exemple si « *Capture* » arrive avant « *Start* »). Cette règle est décrite par le mécanisme M1 d'antiblocage (M1, cf. algorithme 5).
- L'autre règle concerne l'analyse intrusive de l'observateur si l'exigence à satisfaire est violée. L'observateur peut alors stopper l'exécution de la vérification du système si le niveau de criticité de l'exigence formelle est haut. Ceci induit donc l'envoi de messages à tous les objets du système pour stopper leurs exécutions. Cette règle est respectée grâce au mécanisme d'interruption de l'exécution du modèle (M2, cf. algorithme 5). Dans le cas de la violation d'une exigence de niveau bas, l'observateur notifiera la violation de l'exigence sur le graphe d'accessibilité qui sera ensuite répertorié dans la matrice de traçabilité.

Tout comme dans les algorithmes 2,3 et 4, l'observateur est construit à l'envers en débutant par l'opérateur de fin.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Algorithme 5 Début et terminaison d'un observateur

```
   String Build_complete_Obs_behavior(int[] label,int[] T_out,int crit, int nb_TC, String
Violated_Action) {
             //On commence par la fin donc dernier caractere
             String Obs_label = "Capture; ";
             //Chemin si l'exigence est violee
             String end_interrupt="";
      //M2. Interruption de l'execution du modele en fonction du niveau de criticite
             if (crit==1) {
                    end_interrupt="End";
                    //Construire pour chaque TClasse un dispositif de preemption
                    for (int i=0;i<nb_TC;i++)
                           end_interrupt= " Stop_"+i+"; "+ end_interrupt;
             }
             //sinon rebouclage
             else end_interrupt="Loop(Begin)";
             if (label[label.length-1]==0) Obs_label = ("<" + Violated_Action+"; "+
Obs_label+ end_interrupt+";> ");
                    else Obs_label=" <"+ Obs_label + "Loop(Begin);> ";
             for (int i =label.length-2 ; i>=0 ; i--) {
                    if (label[i]==0) Obs_label = Violated_Action+"; "+end_interrupt+";> " +
", " + Obs_label;
                    else Obs_label= "Loop(Begin);> " + ", " + Obs_label;
                    Obs_label= "TLO("+T_out[i]+ ", Capture, <" + Obs_label+ ")";
                                  }
      //M1. Insertion des mecanismes antiblocage
             Obs_label= "Begin; Choice_1 (<Capture; Loop(Begin);> + <Start; Choice_2
(<Start; Loop(Choice_2);>"+ " + <" + Obs_label + ";> ));";
      return Obs_label;
      }
```

Glossaire: `label` = tableau contenant les éléments de description des exigences temporelles (OK ou KO sous forme de tableau d'entier)
`T_out`= tableau contenant les informations temporelles converties en temps absolu pour la génération du diagramme d'activités de l'observateur
`crit` = niveau de criticité de l'exigence (0 = bas et 1 = haut)
`nb_TC` = nombre de Tclasses du modèle observé
`Violated_Action` = étiquette de violation d'exigence définie dans le diagramme d'exigences représenté par une chaîne de caractères
`end_interrupt` = étiquette pour le processus d'interruption (ou le rebouclage) qui dépend du niveau de criticité de l'exigence représenté par une chaîne de caractères
`Obs_label` = Diagramme d'activités de l'observateur représenté par une chaîne de caractères.

2.4. Intégration de l'observateur dans le modèle

2.4.1. Définition des points d'observations

Le Tab.10 présente les deux pictogrammes retenus pour représenter les actions à capturer pour vérifier les exigences. Ces interactions sont ensuite reportées sur le diagramme de classes en représentant les associations entre objet(s) observé(s) et l'observateur.

Label du TRDD	Symbole Graphique	Traduction textuelle
Start_Action	« **Start_Action**»	assoc=<T_obs, synchro, Start=Start_Action, T_obj>
Capture_Action	« **Capture_Action**»	assoc=<T_obs, synchro, Capture=Capture_Action, T_obj>

Tab.10. Table de traduction des descriptions d'exigences temporelles TURTLE en relation dans le diagramme de Classes TURTLE

NB: R représente la(les) relation(s) entre les Tclasses, T_obs la TClass de l'observateur et T_obj la(les) Tclass(es) observé(s). On a assoc=<T, type, formule OCL, T'> [APV 04] où T et T' désignent des Tclasses dont le type correspond à l'opérateur de composition (synchro, séquence, préemption, invocation) attaché à l'association entre les TClasses et où la formule OCL désigne l'opération de synchronisation entre les portes des Tclasses.

2.4.2. Construction du modèle observable : principe

La Fig.45 montre que TTool traduit tous les diagrammes TURTLE (sauf les diagrammes de cas d'utilisation) en TIF (*TURTLE Intermediate Format*). Ce format intermédiaire correspond aux diagrammes de conceptions (CD ou AD) exprimés en TURTLE natif [APV 06] et sert de point de départ pour la transformation de la spécification TURTLE en langage formel RT-LOTOS.

Le modèle TIF est donc directement traduit des diagrammes de conception[11] (CD et ADs). A partir des diagrammes d'exigences (RD et TRDDs) des observateurs peuvent être générés pour guider et interpréter la vérification des exigences temporelles décrites par le diagramme de description d'exigences temporelles (TRDD). Les observateurs sont construits dans le format TIF du système spécifié en TURTLE. Le format TIF ainsi obtenu est traduit en spécification RT-LOTOS pour engendrer ensuite le graphe d'accessibilité.

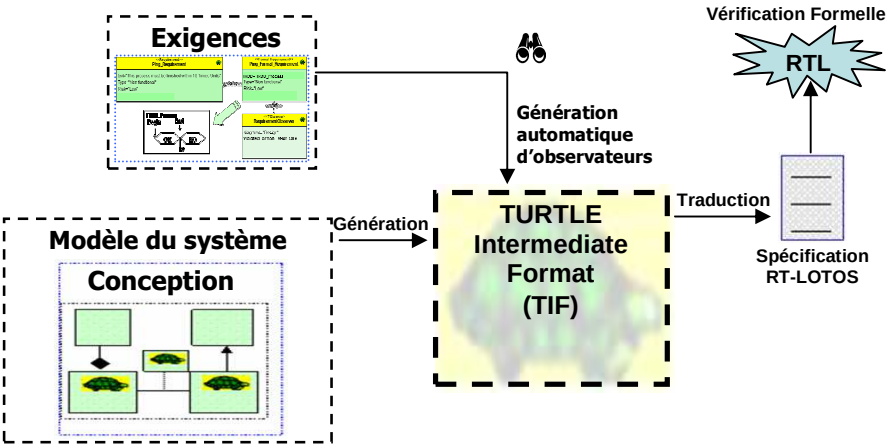

Fig.45. Génération des observateurs

[11] Ce modèle peut aussi être traduit à partir des diagrammes d'analyses (IOD et SDs) [APV 06] ; cet aspect sera uniquement traité dans les perspectives.

2.4.3. Méta-modèle UML du modèle couplé avec l'observateur

La Fig.46 montre le méta-modèle du package de conception TIF où il est possible de greffer un observateur permettant de se synchroniser avec le système observé. Le package TIF est composé d'observateurs et d'un package de conception qui est à observer (Classe *Observed_Design_Package*). Ce package est composé d'un diagramme de Tclasses composé lui-même d'un ensemble de Tclasses à observer (Classe *Observed_Tclass*) et de Tclasses qui ne sont pas observées (Classe *Non_Observed_Tclass*).

Les Tclasses observées contiennent des portes observées (Classe *Observed_Gate* qui correspondent une à une aux points d'observations du TRDD (Classe *Observation_Point*) comme le montre l'association étiquetée par *corresponds_to*. Les portes observées sont ensuite synchronisées avec les portes de l'observateur (Classe *Gates_Obs* composée par la classe *Observation_Gates*). La synchronisation est montrée par l'association étiquetée par *is_synchronized_with*.

Les portes de contrôle d'exécution de l'observateur (Classe *Execution_Control*) interrompent toutes les classes du système en TIF (classes observées, non observées et les observateurs) si l'exigence est violée (le méta-modèle de la Fig.46 le montre par l'association étiquetée par *interrupts* et le commentaire attaché sur la classe *Execution_Control*).

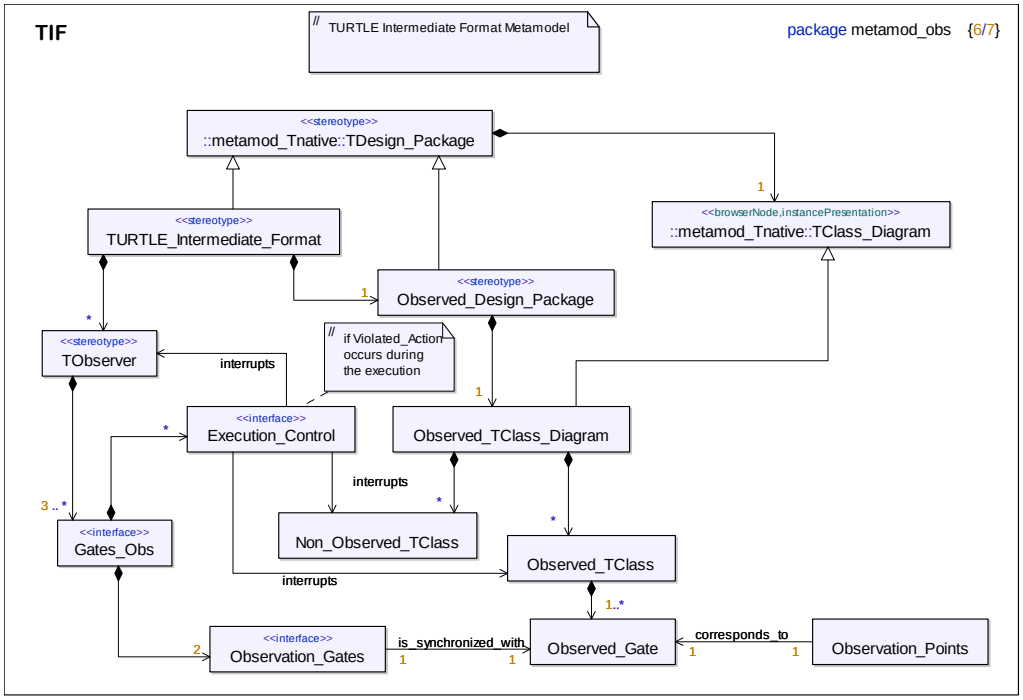

Fig.46. Méta-modèle du package TIF incluant des observateurs

2.4.4. Algorithme d'insertion d'un observateur dans le système à observer

Comme le montre l'algorithme 6, deux étapes sont nécessaires dans la génération du nouveau modèle TURTLE comportant les observateurs :

- Connecter l'observateur aux autres classes/objets du système par des relations de synchronisations (cf. algorithme 7).
- Pouvoir interrompre l'exécution du système si une exigence haute est violée en modifiant les diagrammes d'activité des objets du système, en ajoutant un mécanisme de préemption (cf. algorithme 8). Si une exigence basse est violée alors l'observateur n'interrompt pas l'exécution du système mais notifie sur le graphe d'accessibilité la violation d'exigence qui sera ensuite reportée sur la matrice de traçabilité.

Algorithme 6 Construction du nouveau modèle

```
M_final Create_Mod_Obs (M_init)
    {
    // Création des relations observateur objets observés
          CDfinal = Create_Obs_Relations (CDinit);                  //Algorithme 7
    //Modification des objets observés si besoin d'interruption
          if (criticality==high)
                Mfinal = Create_Obj_Interrupt (Minit);              //Algorithme 8
}
```

Glossaire: `M_final` = Modèle TURTLE comportant l'observateur
Où `M_final` = `<CDfinal, <ADfinal , ADobs>>`
`CDfinal` = Diagramme de classe final
`ADfinal` = Diagrammes d'activités des Tclasses autres que l'observateur
`ADobs`= Diagramme d'activités de l'observateur
`M_init` = Modèle TURTLE sans l'observateur
Où `M_init` = `<CDinit, ADinit >`
`CDinit` = Diagramme de classe initial
`ADinit` = Diagramme d'activités initial des Tclasses

Dans cette partie on connecte l'observateur aux objets observés. Ceci est fait à partir des points d'observations définis dans le TRDD qui correspondent à des synchronisations entrantes avec l'observateur qui les effectue de manière atomique.

Algorithme 7 Construction du nouveau modèle

```
CDfinal Create_Obs_Relations (CDinit)
    {
    // Ajout des relations de synchronisation avec les points d'observation
          CDfinal = CDinit;
          assoc=<T_obs, synchro, Start= Start_Action, T_obj[0]> ;
    // T_obj et T_obj' peuvent être différents ou non
          assoc=<T_obs, synchro, Capture= Capture_Action, T_obj[1]> ;
}
```

Glossaire: `CDfinal` = Diagramme de classe final
`CDinit` = Diagramme de classe initial

Pour respecter l'interruption de l'exécution des objets observés si une exigence haute est violée il faut modifier d'une part le diagramme d'activités de l'observateur (voir algorithme 5) et d'autre part les comportements des objets observés (voir algorithme 8). Il faut construire dans les

objets du système (objets observés ou non) les mécanismes pour stopper l'exécution du système. Il faut donc construire des dispositifs de préemption basés sur synchronisations entre les différents objets du système et l'observateur pour déclencher l'arrêt de l'exécution. De même les diagrammes d'activités des objets du système doivent être modifiés pour prendre en compte ce processus de préemption. Ceci est exprimé par l'algorithme 8.

Algorithme 8 Interruption de l'exécution des objets observés

```
M_final Create_Obj_Interrupt (M_init)
    {
            CDfinal=CDinit
            // Pour l'ensemble de objets/classes du système
            for (i=1, i<=nb_TC, i++)
            // Ajout d'un porte de communication pour interrompre la Tclasse
            {
                  CDfinal= CDinit; R[i]=<T_obs, synchro, Stop_[i]=Stop, T_C[i]>;
            // Ajout d'un mécanisme d'interruption dans le comportement de l'objet/classe
                  <ADfinal[i]> = Preemption (<ADinit[i]>, Stop);
            }
    }
```

Glossaire:
`nb_TC` = nombre de Tclasses dans le diagramme de Classes
`T_obs` = Tclass de l'observateur
`TC` = autres Tclasses
`R` = Relation entre la Tclass de l'observateur et les autres
`M_final` = Modèle TURTLE comportant l'observateur
Où `M_final = <CDfinal, <ADfinal ,ADobs>>`
`CDfinal` = Diagramme de classe final
`ADfinal` = Diagrammes d'activités des Tclasses autres que l'observateur
`ADobs`= Diagramme d'activités de l'observateur
`M_init` = Modèle TURTLE sans l'observateur
Où `M_init = <CDinit, ADinit >`
`CDinit` = Diagramme de classe initial
`ADinit` = Diagramme d'activités initial des Tclasses

3. Discussion sur l'approche de vérification guidée par les observateurs

La section 3.1 caractérise les différentes hypothèses prises sur le modèle TURTLE pour pouvoir formuler les algorithmes présentés dans la section 2. Ces hypothèses ne seraient pas complètes sans la caractérisation du « bon placement » des *points d'observations* dans les objets observés, point crucial dans la vérification guidée par observateurs présenté dans la section 3.2. Enfin, la section 3.3 aborde le prolongement immédiat des travaux présentés dans ce chapitre, à savoir l'implantation dans l'outil TTool [TTOOL].

3.1. Hypothèses

Les algorithmes présentés dans la section 2, fonctionnent correctement si un ensemble d'hypothèses sont satisfaites. Elles se formulent de la manière suivante :

- Seuls des objets exécutables (Tclasses actives stéréotypées par « *start* ») peuvent être connectés aux observateurs et interrompus par ces derniers.

- Le modèle contenant l'observateur est exprimé en langage TURTLE natif [APV 04]. Ce modèle est composé d'un diagramme de classes et de diagrammes d'activités. Ceci introduit deux ensembles d'hypothèses :
    - Les opérateurs de composition pris en compte dans le système sont les opérateurs de synchronisation, parallélisme et séquence. Le parallélisme correspond à deux Tclasses indépendantes sans relation de synchronisation et la séquence correspond à une synchronisation. L'utilisation de l'opérateur de préemption est interdite dans le modèle. Ce dernier peut bloquer un objet observé et amener au blocage non désiré du système par l'observateur. Ce dispositif pose aussi des problèmes pour l'interruption des objets observés.
    - Les opérateurs du diagramme d'activités à prendre en compte sont les opérateurs natifs du diagramme d'activités [APV 06] (synchronisation, choix, opérateur parallèle, séquence pour les opérateurs non temporels et délai, latence, offre limitée dans le temps pour les opérateurs temporels).
- Les *points d'observation* entre l'observateur et les objets observés doivent être associés par un opérateur de synchronisation (ceci est montré par le méta-modèle de la Fig.46)
- Les *points d'observation* « *Start* » et « *Capture* » ne sont pas pris en compte si ces derniers se chevauchent. Seules les occurrences immédiates Start-Capture sont mesurées par les observateurs ; les autres occurrences (par exemple Start-Start ou Capture-Capture) ne sont pas mesurées mais peuvent se produire durant l'exécution du système sans bloquer l'observateur (cf. section 2.3.5).
- Les *points d'observation* « *Start* » et « *Capture* » sont uniques et ne sont jamais dupliqués dans des processus observés en parallèle (problème de synchronisation entre objets observés et observateur ne prenant pas en compte les points d'observation en parallèle). Dans le cas de plusieurs processus en parallèle dupliquant les points d'observation, il faut construire un observateur pour chaque processus.
- Les *points d'observation* ne peuvent pas être inclus dans des dispositifs d'offres limitées dans le temps. Si le *point d'observation* considéré n'est pas dans les deux chemins de l'offre limitée dans le temps, cela peut contribuer au blocage de l'observateur et donc du système.
- Enfin il existe une limitation inhérente à la sémantique de RT-LOTOS associée à l'opérateur d'offre limitée dans le temps [COU 00]. Ce problème se rencontre si l'action de fin de capture d'exigence est effectuée à la date limite de satisfaction d'exigence (à la date T). A la date T, l'offre est possible mais devient périmée. Par conséquent le graphe d'accessibilité contient dans ce cas deux chemins. Le premier contient l'étiquette de violation de l'exigence et le second la satisfaction d'exigence. Il est donc nécessaire de formuler les dates de violation/satisfaction d'exigences dans les bornes supérieures/inférieures selon la construction du TRDD.

3.2. Placement des points d'observations

L'intérêt de construire un observateur dans le langage de spécification du système est que le coût de la preuve est réduit [JAR 88] [HAL 93] [ROG 06]. En effet, la structure de l'observateur est déjà formalisée et donc prouvée. Cependant la composition avec le système à observer peut conduire à un échec de diagnostic voire un blocage du système et nécessite d'être définie. Dans cette section,

nous caractérisons donc l'aspect composition de l'observateur avec le système observé en caractérisant le placement « correct » des *points d'observations*.

Comme mentionné dans la section 2 (cf. Fig.42 et Fig.43) de ce chapitre, un observateur peut être vu comme une succession d'offres limitées dans le temps qui se synchronisent avec le modèle observé de manière instantanée (atomique). Nous avons choisi d'abstraire le système (cf. partie « *objets du système* » de la Fig.47) par des latences qui représentent l'occurrence des événements caractérisant le début et la fin l'exigence (respectivement notés « *Action Start* » et « *Action_Capture* » sur la Fig.47). Les points d'observation (« *Start* » et « *Capture* ») se synchronisent de manière instantanée avec l'observateur (synchronisation atomique). Nous avons étudié le placement des points d'observation en séquence de deux manières différentes :

- Le point d'observation est placé avant l'action caractérisant l'exigence (équivaut aux portes *Start_1* et *Capture_1*).
- Le point d'observation est placé après l'action caractérisant l'exigence (équivaut aux portes *Start_2* et *Capture_2*).

Si les points d'observations sont placés avant les événements caractérisant l'exigence alors ceux-ci déclencheront l'observateur avant l'occurrence de l'événement dans l'objet observé. Ceci entraine donc une erreur de diagnostic de l'observateur. Dans le cas contraire, l'observateur se déclenche instantanément (par synchronisation atomique entre l'objet observé et l'observateur) après l'occurrence de l'événement : le diagnostic de l'observateur est, dans cette dernière configuration, exact.

Si on utilise un opérateur de parallélisme entre l'événement à observer (« *Action Start* » ou « *Action_Capture* ») et le point d'observation (« *Start* » ou « *Capture* »), on obtient les deux configurations à la fois. Ceci est expliqué par la sémantique du parallélisme en TURTLE. La synchronisation entre l'observateur et l'objet observé étant atomique, la synchronisation entre l'observateur et l'objet observé se produira avant l'occurrence des événements « *Action_Start* » et « *Action_Capture* » (cf. occurrence de Start_1 et Capture_1 sur la Fig.47). Ceci entrainera aussi un mauvais diagnostic de l'observateur.

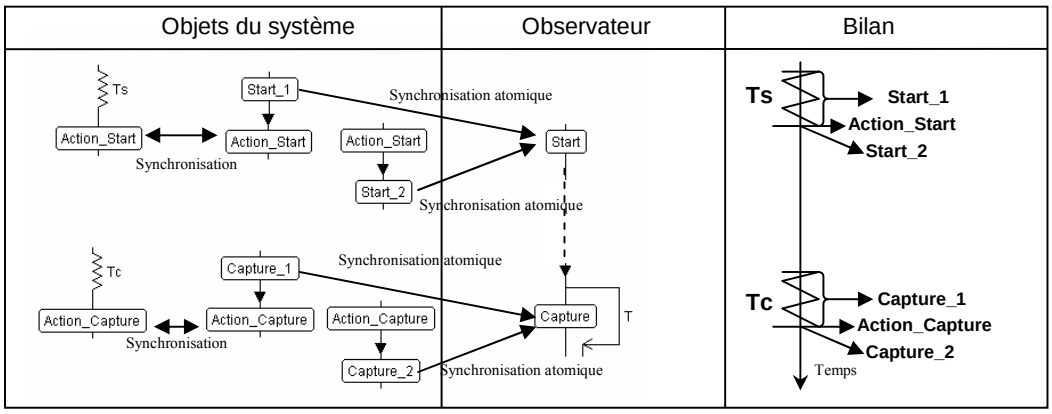

Fig.47. Caractérisation du placement des points d'observations

Pour conclure, il existe un seul branchement « correct » des points d'observations : ces derniers doivent être placés en séquence, ***immédiatement après*** ***les événements à observer***.

Le concept de « *committed location* » utilisé dans l'outil UPPAAL [UPPA] se prête bien pour définir la manière de placer les points d'observations. En UPPAAL, un état est dans une « *committed location* », si et seulement si on ne peut pas attendre cette action (non-bloquante). La prochaine action permet de sortir de la *commited location*. Cette dernière est vue comme une zone temporelle où les actions sont tirées instantanément, sans entrelacement avec les actions hors de la *committed location*, qui seront exécutées après. Une perspective serait donc une extension dans la sémantique de TURTLE pour définir tout comme UPPAAL la notion de « *committed location* » pour placer les points d'observations dans le modèle observé.

3.3. Vers une implantation complète de l'approche

Le prolongement immédiat de nos travaux concerne l'implantation complète du processus de vérification guidée par les observateurs à partir d'une spécification des exigences dans un diagramme d'exigence et TRDD. La Fig.48 présente ce qui a été implanté dans TTool [TTOOL] et ce qui reste à faire concernant le processus de génération automatique des observateurs. Les éléments déjà construits sont représentés par des lignes pleines, qui sont :

- Les algorithmes présentés dans le chapitre V (cf. Gene_Obs sur la Fig.48) prennent en entrées suivantes :
    - les informations du diagramme d'exigences soit le niveau de criticité, l'étiquette de violation d'exigences et le nombre de classes actives dans le système.
    - Les informations du TRDD soit un tableau d'entier représentant les valeurs de la ligne de vie de l'exigence soit O pour « KO » et 1 pour « OK » (`int[] TRDD`), les valeurs des frontières temporelles (`int[] T_in`) et les étiquettes correspondantes aux deux points d'observations (`String[2] P_O`) et aux objet observés (`String[2] T_Obj`) définis dans le TRDD.

  Ces algorithmes produisent en sortie deux chaînes de caractères représentant le comportement de l'observateur (`String obs_behavior`) et le modèle incluant l'observateur (`String observable_model`).
- Le diagramme d'exigences (TRequirement_Diagram) présenté par les méta-modèles du chapitre IV section 2.1.3), comme le montre les travaux présentés en [FON 06b].
- L'API (notée API_RD sur la Fig.48), entre les diagrammes d'exigences (TRequirement_Diagram présenté par les méta-modèles du chapitre IV section 2.1.3) et les algorithmes de génération des observateurs.

Enfin, les éléments restant à construire sont présentés dans la Fig.48 par des éléments en pointillé. Il reste donc à implanter :

- Le diagramme TRDD (noté TRDD sur la Fig.48) présenté par les méta-modèles du chapitre III section 2.2.3.
- Une API (notée API_TRDD sur la Fig.48), entre le TRDD et Gene_Obs, pour produire à partir du TRDD les tableaux `TRRD`, `T_in`, `P_O` et `T_obj`.
- Une API (notée API_TIF sur la Fig.48), entre Gene_Obs et le modèle de conception incluant l'observateur, pour produire un nouveau modèle de conception à partir des descriptions sous

forme de chaîne de caractères du comportement de l'observateur et du modèle couplé avec ce dernier.

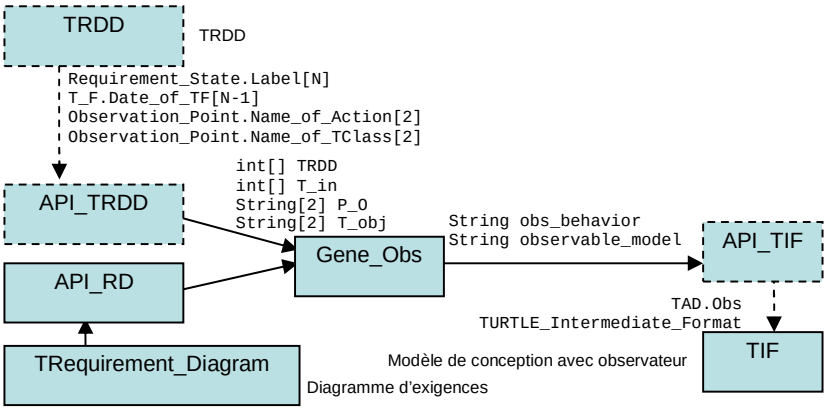

Fig.48. Vers l'implantation de nos travaux

4. Conclusion

Ce chapitre a traité le volet « vérification formelle des exigences non-fonctionnelles temporelles » de la méthodologie présentée dans le chapitre III instanciée au profil UML TURTLE. Notre choix s'est porté sur la construction d'observateurs pour guider la vérification des exigences non-fonctionnelles temporelles. En effet, cette technique est plus rapide à mettre en œuvre et nécessite moins de contrainte d'outillage que les approches de type de *model-checking*. L' approche retenue ici nécessite simplement de spécifier l'observateur dans le langage de spécification du système (en l'occurrence en diagrammes TURTLE). Enfin, cette méthode nous parait plus appropriée pour vérifier des exigences incluant des « distances temporelles ». En effet, de nombreuses logiques ne permettent pas d'exprimer directement ce type d'exigences (généralement on construit des observateurs représentant ces distances temporelles en procédant comme [MAL 06]).

Les contributions présentées dans ce chapitre reposent sur :

- La définition et les méta-modèles des observateurs dédiés à la vérification d'exigences non-fonctionnelles temporelles dans le profil TURTLE.
- La présentation d'algorithmes de génération automatique des observateurs dans le modèle TURTLE (d'analyse ou de conception) à partir des informations du diagramme d'exigences et du TRDD (définis dans le chapitre IV).
- La définition d'hypothèses pour la génération d'observateur ainsi que le placement correct des points d'observations dans le système pour que l'observateur puisse délivrer un diagnostic correct. Notons que cette dernière définition peut s'appliquer plus généralement à la construction d'observateur dans un modèle en langage UML.

Les contributions présentées dans ce chapitre préfigurent l'implantation de l'ensemble des algorithmes de génération d'observateurs dans TTool (cf. section 3.3). Ainsi, une de nos ambitions est de retravailler ensuite le générateur de code exécutable Java qui existe déjà dans l'outillage TURTLE afin que ces prototypes d'observateurs servent de premier filtre dans notre démarche de vérification et ce, avant la génération d'observateurs à partir d'un code exécutable « prouvé ».

Chapitre VI. Application : le projet SAFECAST

Les missions de sécurité civiles et militaires nécessitent la coopération de divers corps d'intervention tels que les secouristes, les pompiers et les policiers. Chaque corps possède ses spécificités en termes de règles hiérarchiques et de commandement. Leurs interventions sur des théâtres d'opération communs impliquent une organisation dynamique des effectifs placés sur le terrain pour former des groupes, hétérogènes de par leur corps d'origine, mais structurés et commandés de façon cohérente en fonction des rôles de chacun.

Telle est la problématique du projet SAFECAST [SFC] dédié à l'étude, la conception et la validation de fonctionnalités et de mécanismes garants de communications sécurisées à l'intérieur de groupes. Ce projet a mis en évidence l'intérêt d'attaquer un même problème – celui de la modélisation et de la vérification de protocoles de communication de groupes sécurisée – en utilisant deux outils complémentaires : AVISPA [AVI 05] pour la vérification d'exigences de sécurité et TURTLE [APV 04] pour la vérification d'exigences temporelles. Parmi les résultats qui attestent de l'utilisation de TURTLE [L 2.5] [L 4.1] [L 3.4] [FON 07] dans ce projet, ce chapitre se focalise sur un point dur de l'architecture SAFECAST : le service et le protocole de génération et de distribution des clés.

Ce chapitre est construit de la manière suivante. La section 1 présente brièvement le projet SAFECAST et le médium PMR [PMR] qui conditionne les choix réalisés dans ce projet. Ensuite, un panorama des travaux de recherche sur la modélisation et la vérification des protocoles de sécurité aide à positionner notre contribution à SAFECAST par rapport aux travaux du domaine. La section 2 propose des canevas d'exigences et d'architecture incluant les différents services du protocole SAFECAST. La section 3 introduit le modèle TURTLE pour la vérification formelle d'une exigence temporelle concernant la durée du service de génération et de distribution des clés. La section 4 dresse un bilan des apports de TURTLE au projet SAFECAST. Enfin la section 5 conclut ce chapitre.

1. Le projet SAFECAST

L'entreprise EADS [EADS], porteur de ce projet, propose des équipements et services dans le domaine de la sécurité civile. Ses prestations et produits ciblent divers organismes tels que la police ou les pompiers. Grâce au protocole TETRAPOL [TETRA] de la PMR [PMR] (Professional Mobile Radio), EADS garantit des communications sécurisées à ses clients.

Cependant, TETRAPOL est sensible à la perte de communication avec le centre qui gère la sécurité. Cette lacune s'est faite faiblement sentir jusqu'à des évènements récents où l'infrastructure PMR s'est révélée défaillante. Ainsi, les communications n'étaient plus relayées entre les différents groupes d'intervention. Leurs actions ne pouvaient plus être synchronisées et la sécurité du système se trouvait alors dégradée.

Le projet SAFECAST [SFC] a été lancé avec l'ambition de combler cette lacune. SAFECAST porte sur le développement d'une architecture globale de sécurité pour des communications de groupes en prenant en compte l'incidence des pertes de communication par le protocole utilisant le médium PMR (en l'occurrence le protocole SAFECAST).

1.1. Réseau sans fil : le médium PMR

La PMR est constituée d'un ensemble de moyens de communications mobiles sans fil à l'usage de professionnels. Les premiers équipements mobiles sans fil sont apparus dans les années 1930, avec des équipements mobiles pour les voitures, puis des équipements portables de type talkie-walkie.

Pour se faire une idée de l'utilisation du médium PMR, il est utile de rappeler les différences de finalité entre un réseau public cellulaire (GSM) et un réseau PMR :

- Un réseau public cellulaire offre des communications de type téléphonique entre deux utilisateurs mobiles quelconques situés sous la couverture du réseau.
- Un réseau de type PMR permet de piloter à partir de postes fixes, des opérations menées sur le terrain par des groupes d'utilisateurs mobiles (éteindre un feu, surveiller un événement, optimiser le ramassage de paquets par une flotte de véhicules, distribuer des courses à des taxis, etc…). PMR offre également des services de communication directe entre terminaux mobiles (talkie-walkie ou shelter[12]).

Au niveau de la transmission, les débits offerts par le PMR sont de deux types. La classe « bas débit » garantit un débit de 6 kb/s avec une portée de 3 km. La classe « moyen débit » supporte 100 kb/s avec une portée de 100 km. Le PMR offre un service en partie fiable (pas de duplication, ni de déséquencement). Le projet SAFECAST a traité les pertes de messages inhérentes au fait qu'un utilisateur de talkie-walkie puisse être hors de porté durant l'émission ou la réception d'un message.

La caractéristique principale d'une application utilisant le réseau PMR est donc de permettre à des groupes d'utilisateurs mobiles équipés de terminaux sans fil de communiquer entre eux de façon sécurisée pour des communications de voix, de données ou multimédia. Les caractéristiques importantes pour ces utilisateurs sont (les caractéristiques en italiques correspondent aux exigences vérifiés en TURTLE) :

⇒ la qualité de la voix,
⇒ le *délai d'établissement d'une communication,*
⇒ le *délai d'entrée tardive dans une communication établie,*
⇒ le *délai de réunion de deux communications établies en une seule,*
⇒ le débit en transmission de données,
⇒ le nombre de groupes pouvant établir des communications,
⇒ la taille maximale des groupes.

[12] Sac à dos utilisé pour la transmission, ayant une portée plus grande que les talkies-walkies de l'ordre de 100 km contre 3 km pour ces derniers.

1.2. Revue de travaux sur la modélisation et la vérification de protocoles de sécurité

Les techniques ou langages de modélisation de protocoles de sécurité associés à un outil de vérification peuvent être classés en quatre groupes :

(i) Les modèles basés sur des théories mathématiques qui manipulent les propriétés algébriques de façon automatique [COR 05].

(ii) Les modèles issus de la communauté de la logique qui vérifient des propriétés au moyen de preuves de théorèmes [BLA 01], selon des procédures complexes et peu automatisables.

(iii) Les modèles à base de règles de réécriture [RUS 04] qui sous-tendent de nombreux environnements dédiés aux protocoles de sécurité. CASRUL [RUS 04] et AVISPA [AVI 05] en sont des exemples. La principale faiblesse de ces outils est de ne pas prendre en compte le temps.

(iv) Les systèmes de transitions, dont le Tab.11 présente de manière synthétique les principaux environnements caractéristiques : FDR2, MURφ, UPPAAL, LYSA, SPIN et TURTLE. Un comparatif plus complet a été proposé en [FON 06a].

	Systèmes de transition pour la vérification de modèles					
Outil	FDR2	Murφ	UPPAAL	LYSA	SPIN	TTOOL-RTL
Référence	[FDR 05]	[DIL 92]	[JUR 02]	[BUC 04]	[MAG 02]	[APV 04]
Formalisme	Algèbre de processus	Algèbre de processus	Automates Temporels	Algèbre de processus	Automates étendus	Algèbre de processus
Langage Formel	CSP	Murφ	Automates	Lysa	Promela	RT-LOTOS
Compatibilité UML	Non	Non	Non	ForLysa	Profil UMLSec	Profil UML/TURTLE
Vérification Orientée	Contrôle	Contrôle	Contrôle/ Temps	Contrôle	Contrôle	Contrôle/ Temps
Type de Résultat	Trace	Trace	Diagnostic avec Trace	Trace	MSC	Automate Quotient

Tab.11. Outils de vérification de protocoles de sécurité basés sur les systèmes de transitions

L'état de l'art publié en [FON 06a] a mis en évidence que les travaux sur la modélisation et vérification formelle des protocoles de sécurité ignorent majoritairement les aspects temporels. Tel n'est pas le cas du profil UML TURTLE présenté dans le chapitre II (section 4) et utilisé dans le cadre du projet SAFECAST.

1.3. Pourquoi utiliser TURTLE dans le projet SAFECAST ?

A l'origine, le profil UML temps réel TURTLE n'a pas été développé dans le but de modéliser et vérifier des protocoles de sécurité. Son applicabilité en ce domaine a été testée avec succès sur le protocole NSPK (Needham Schroeder Public Key protocol) [FON 06a]. Au-delà de cet exemple canonique, l'usage de TURTLE dans le projet SAFECAST se justifie par les points suivants :

- En TURTLE, la vérification formelle n'est pas une activité isolée. Elle s'intègre dans un processus qui démarre avec la spécification formelle des exigences temporelles (relations entre exigences, description formelle), une analyse (cas d'utilisation, scénarios) et embraye sur la conception (architecture, comportements).

- Une des forces du langage TURTLE réside dans le traitement des aspects temporels. Outre les retards de durées fixes et les temporisateurs que l'on retrouve dans les outils UML 2.0, TURTLE permet de travailler avec des intervalles temporels. De plus l'analyse d'accessibilité prend en compte le temps et gère ces intervalles temporels.
- Tant le langage de modélisation que les outils de vérification sont bien adaptés à traiter la partie « contrôle » (échanges de messages et respect des contraintes temporelles) d'une machine de protocole. Les traitements qui s'expriment dans un paradigme états/transitions se modélisent bien en TURTLE. Par contre, les algorithmes –par exemple des fonctions de cryptage/décryptage– qui manipulent des données complexes ne peuvent pas être détaillés. Le plus souvent le modèle TURTLE représentera l'algorithme par sa durée de traitement estimée et une abstraction des résultats qu'il produit (par exemple, succès ou échec du cryptage d'une donnée).

Par ailleurs, l'expérience montre que plus l'on cherche à analyser de matière pointue et exhaustive une des « facettes » d'un système –par exemple pour apprécier le niveau de sécurité offert par ce système ou encore pour vérifier qu'il respecte un certain nombre d'exigences temporelles– plus il faut avoir recours à un langage de modélisation spécialisé. Le pouvoir d'expression de ce langage est en général très grand dans la facette qu'il permet d'aborder mais s'avère plus limité dans les autres facettes du système. On peut dire sans caricaturer que les langages formels dotés de techniques d'analyse puissantes sont souvent « trés spécialisés ». Ainsi, il est rare qu'un même outil traite à un même niveau la partie « contrôle » d'un protocole (échanges de messages et respect des contraintes temporelles) et la partie « données » (format des messages et sémantique des données véhiculées).

Ceci nous amène à dire que l'étude d'un système complexe tire bénéfice de « modélisations croisées » impliquant plusieurs langages de modélisation et outils de vérification. C'est l'approche retenue dans la cadre du projet SAFECAST qui « croise » une modélisation orientée « données » et « sécurité » à l'aide de l'outil AVISPA [AVI 05] avec une modélisation orientée « contrôle » et « contraintes temporelles » mise en œuvre par l'outil TURTLE [APV 04].

Nos contributions dans le projet SAFECAST concernent plus particulièrement la vérification des aspects « contrôle » et « contraintes temporelles » (notés en italiques dans la section 1.1 dans le dernier paragraphe) d'une application PMR de communication de groupes sécurisés. Ce projet fut aussi une occasion d'instancier la méthodologie proposée dans le chapitre III sur une étude de cas industrielle.

2. Exigences, architecture et services considérés

L'instanciation de la méthodologie du chapitre III sur le projet SAFECAST, nous a permis de définir une approche générale de modélisation et d'étude des exigences temporelle liées à un protocole de sécurité. Cette approche est présentée dans cette section au travers de la définition d'exigences temporelles inhérentes aux protocoles de sécurité et d'un canevas d'architecture. Dans cette section, nous détaillons ces deux aspects : les exigences (majoritairement temporelles) inhérentes à la conception de protocoles de sécurité en général et un canevas d'architecture avec les différents services sur lesquels repose le modèle TURTLE du protocole SAFECAST.

2.1. Exigences temporelles d'un protocole de sécurité

La construction d'un modèle dédié à la vérification du protocole SAFECAST, nous a amené à définir des exigences généralisables à l'ensemble des protocoles de sécurité. Ce type de protocole pose des exigences liées à la fiabilité. Pour pouvoir résister aux attaques, le protocole doit être « robuste ». Au niveau protocolaire il doit être fiable et en mode connecté et donc garantir les exigences *E 2 et E 3* (cf. chapitre II section 4.1).

Il est communément admis dans les protocoles de sécurité que les données doivent être échangées dans un délai borné sur le réseau pour pouvoir garantir les propriétés d'intégrité et de confidentialité. Selon la *théorie des graphes* de cryptographie [DOD 05], toute clé de cryptographie est « cassable » à partir d'une période donnée (appelée *temps de cyclage*). Il découle de cette affirmation trois classes d'exigences non-fonctionnelles temporelles types :

E 5. Un protocole de sécurité doit avoir un délai borné pour la génération et la distribution de clés ; ceci correspond à la propriété de promptness définie dans le chapitre II section 1.

E 6. Un protocole de sécurité doit avoir un délai borné pour le transfert de données ; ceci correspond à la propriété de promptness définie dans le chapitre II section 1.

E 7. Un protocole de sécurité doit avoir un temps de session borné dans un intervalle temporel ; ceci correspond à la propriété d'interval delay définie dans le chapitre II section 1.

2.2. Architecture générique

A notre connaissance, il n'existe pas de patron d'architecture communément admis pour modéliser des systèmes de sécurité [AMI 05]. Dans la lignée du modèle de référence présenté dans le chapitre II section 5, nous avons opté pour une architecture dite *en couches de protocoles*. Les niveaux les plus bas sont proches des fonctionnalités de communication du médium PMR ; les plus élevés se rapportent aux fonctionnalités de gestion de l'information.

Quatre niveaux développés sous la forme de couches indépendantes composent l'architecture de modélisation du protocole de communication de groupe sécurisés (cf. Fig.49). Les niveaux les plus bas se chargent des opérations de communication sécurisées ; ils représentent le modèle du *medium* (couches *Medium* et *SO*). Les deux niveaux les plus élevés se chargent des opérations de gestion des éléments mis en œuvre (clés et groupes) ; ce sont les *entités de protocoles* dont on vérifie le comportement (couches *GCKM* et *GMM*). Au niveau applicatif, se trouvent les utilisateurs connectés au système (couche *User*).

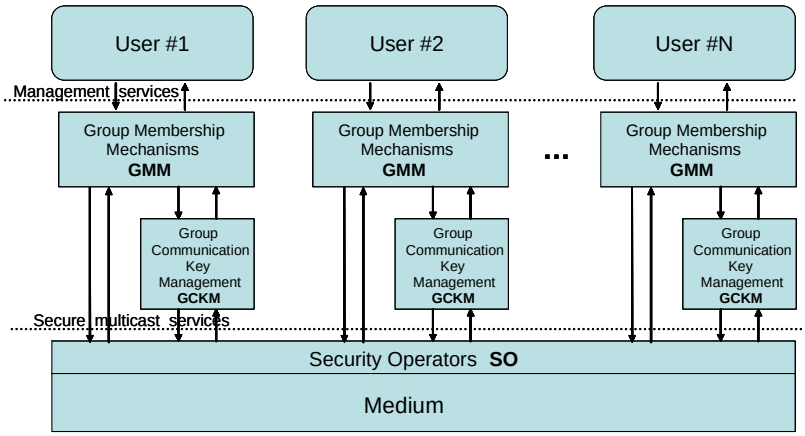

Fig.49. Architecture de modélisation de communications de groupes sécurisées

2.2.1. Services de diffusion sécurisée

La couche de communication la plus basse (*Medium* sur la Fig.49) offre les services élémentaires de diffusion multipoint. Trois types de services ont été définis dans ce médium : un service de communication point-à-point, un service multipoint (de 1 vers N) et un service de diffusion (1 vers tous).

La deuxième couche, *Security Operators* (SO), offre les actions élémentaires de sécurité qui garantissent l'intégrité, la confidentialité et le non rejeu. On trouve des opérations de : hashing pour garantir l'intégrité des données ; chiffrement qui sert à la confidentialité ; empêchement de rejeu avec l'indexation des numéros de séquence des messages reçus.

2.2.2. Services de gestion des groupes

La couche *Group Communication Key Management* (GCKM) gère les opérations de sécurisation des groupes liées aux clés de session. Le service de renouvellement de clés inclut deux sous-services : la génération et la distribution des clés. Les clés de session générées sont destinées à chaque niveau de hiérarchie d'un groupe de communication. La distribution des clés peut se faire par deux modes différents : par un chef de groupe vers d'autres chefs de groupes différents, ou par un chef vers l'ensemble de son groupe. Le renouvellement des clés doit se faire en maintenant l'intégrité du système vis-à-vis des changements de rôles (un membre peut être à un moment donné être un superviseur et perdre ses droits durant une session) et des vis-à-vis des entrée/sorties des utilisateurs.

La couche supérieure, *Group Membership Mechanisms* (GMM), se charge des éléments de gestion de groupes en contrôlant leur structure, leur évolution et leur dynamique. Cette couche intègre les fonctionnalités intra-groupe de la couche GCKM et l'intégration faite sous la forme d'un service par fonctionnalité. Le fonctionnement de ces services met en relation les mécanismes de gestion des clés de session de la couche GCKM et l'utilisation des rôles qui donnent des droits aux membres.

3. Etude du délai d'établissement d'une communication chiffrée

Cette section se focalise sur la vérification d'une exigence temporelle concernant le service de génération et de distribution de clés du protocole de gestion de clés de communication. Une description complète de la conception et de la vérification du protocole SAFECAST a été proposée en [L 4.2] et [FON 07]. Dans ce chapitre, une exigence suffit à illustrer la mise en œuvre de la méthodologie TURTLE.

La structuration de cette section respecte ainsi la méthodologie présentée dans le chapitre III. La section 3.1 présente la phase de traitement d'exigences, particularisée à l'exigence non-fonctionnelle temporelle de délai d'établissement d'une communication chiffrée. La section 3.2 décrit la phase de conception et traite du protocole de gestions de clé (en particulier le service décrit par la fonctionnalité *Key_Renewal*), et du mécanisme d'abstraction du PMR : le *Routage Hiérarchique Virtuel* (noté par la suite RHV). Enfin, la section 3.3 présente la phase de vérification –guidée par un observateur– de l'exigence de délai d'établissement d'une communication chiffrée.

3.1. Traitement des exigences

L'exigence temporelle considérée dans cette section concerne la confidentialité des échanges. Il est dit dans [L. 1.3] section 3.1.1 que :
« Le mécanisme de chiffrement doit posséder les propriétés suivantes compte tenu du contexte PMR:

- Etablissement de connexion (EC) : le délai d'établissement d'une communication chiffrée doit être inférieur à 350 ms. »

3.1.1. Recueil des exigences

Le délai d'établissement de connexion correspond au temps nécessaire pour générer et distribuer une nouvelle clé à l'intérieur de chaque groupe. Le tableau ci-dessous montre une partie de la base de données correspondant à cette exigence. Nous avons choisi de montrer l'exigence EC (de numéro 1.0) qui correspond à la confidentialité des échanges (exigence trop abstraite pour être fonctionnelle ou non-fonctionnelle). Cette exigence est raffinée en deux sous-exigences : l'exigence de délai d'établissement de communication chiffrée qui est de type non fonctionnelle temporelle et le mécanisme de renouvellement de clé lui-même qui est une exigence fonctionnelle garantissant la confidentialité des échanges. Ces deux exigences sont respectivement numérotées par 1.1 et 1.2. Elles sont de niveaux techniques car nécessaires pour la gestion de groupe correspondant aux besoins utilisateur.

Requirement ID		Requirement Name	Requirement Text	Level of Goal	Kind
1	0	EC	…	Technical	-
1	1	EC_Duration	…	Technical	Non-Functional
1	2	Key_Renewal	…	Technical	Functional

Tab.12. Base de données des exigences liées à l'établissement de connexion et au renouvellement de clés

L'exigence non-fonctionnelle temporelle apparait dans le Tab.13. Cette exigence est directement liée à la fonctionnalité de renouvellement de clés service fournie par la couche *Group Communication Key Management* (GCKM). Il s'agit du service le plus utilisé dans la gestion de clés. Il est très critique pour le fonctionnement de l'ensemble du protocole SAFECAST, ce qui justifie son niveau de criticité « haut ». Les acteurs concernés par cette exigence non-fonctionnelle temporelle

sont tous les utilisateurs de ce service, tous les participants devant posséder une clé pour pouvoir participer à la communication. Cette exigence temporelle de délai maximum (*promptness*) doit être inférieure à 350 ms. Elle est directement liée à la fonctionnalité de renouvellement de clés ; ce service est fourni par la couche *Group Communication Key Management* (GCKM) qui est l'exigence fonctionnelle *Key_Renewal*.

Requirement ID		Requirement Name	Risk Level	Concerned Actors	Kind of temporal Requirement	Use Case
1	1	EC_Duration	High	All_User_of_GCKM_Layer	Promptness	Key_Renewal

Tab.13. Base de données présentant l'exigence de délai de renouvellement de clés

3.1.2. Hypothèses de modélisation

Concernant les hypothèses intangibles, l'aspect « données » (cryptage/décryptage) a été abstrait, comme pour la majorité des outils basé sur les systèmes de transitions (cf. section 1.2). De plus, TURTLE est un outil dédié à la vérification d'exigences temporelles et non aux exigences de sécurité (contrairement à l'outil AVISPA utilisé par d'autres équipes du projet SAFECAST). Les mécanismes de cryptographie (cryptage/décryptage de récupération de certificat, hashing) ont des contraintes temporelles (cf. section 2.1). Ces mécanismes dépendent des caractéristiques liées à la sécurité (taille des clés, algorithme de cryptologie utilisé, …) d'une part et du système (par exemple la vitesse de calcul du microprocesseur) d'autre part. Nous avons opté pour l'abstraction suivante :

HI 2. Tout processus de type cryptographique (cryptage/décryptage de récupération de certificat, hashing) est abstrait par un délai et une latence fixés par les caractéristiques de sécurité du protocole (algorithmes de cryptographie, type et tailles de clés) et du système (vitesse CPU,..).

Concernant la définition des hypothèses appelées à être levées et les *aléas* correspondants (cf. chapitre III section 4.1.2), nous avons considéré les pertes dans le modèle en application de la règle *HL 4.1* (cf. chapitre III section 4.1.2). Le réseau PMR est un réseau sans fil déployé à grande échelle (de 3km à 100 km), pouvant entrainer la perte de connexion de certains utilisateurs.

3.1.3. Spécification d'une exigence temporelle

La Fig.50 est un diagramme d'exigences conforme au pattern présenté dans le chapitre III section 3.1.3, qui représente l'exigence *EC_Duration* que doit satisfaire la fonctionnalité de renouvellement de clés (*Key_Renewal)*. L'exigence informelle *EC_Duration* est dérivée en exigence formelle pour obtenir l'exigence *Formal_ECD* (comme le montre la flèche stéréotypée par « derive »). L'exigence *Formal_ECD* peut alors être vérifiée par un observateur directement généré à partir de cette dernière (flèche stéréotypée par « verify ») ; cet observateur est appelé *RequirementObserver*. *Formal_ECD* s'applique ainsi à la fonctionnalité de renouvellement de clés *KeyRenewal* (comme le montre la flèche en pointillé stéréotypée par « satisfy » sur la Fig.50).

L'exigence *Formal_ECD* est ensuite spécifiée formellement par un TRDD permettant d'exprimer l'exigence temporelle de *promptness* (cf. chapitre IV section 2.3). On introduit donc deux points d'observations *Begin_KR* et *End_KR* qui représentent respectivement le début de l'établissement d'une connexion (un renouvellement de clé demandé par le superviseur) et la fin de

l'établissement de la connexion. Ce dernier point d'observation sera donc synchronisé avec l'ensemble des classes utilisant le service de renouvellement de clés, conformément à ce qui a été mentionné dans le Tab.13.

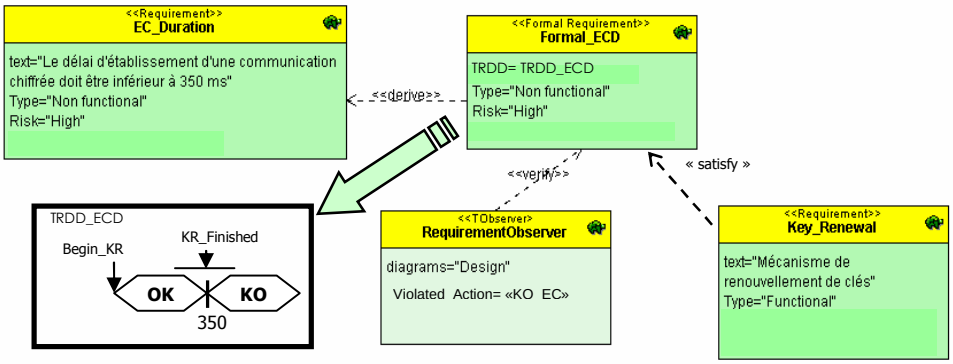

Fig.50. Diagramme d'exigence *EC_Duration*

3.2. Modèle de conception pour la fonctionnalité Génération et Distribution des clés (Key_Renewal)

Le modèle présenté sur la Fig.51, repose sur l'architecture de la section 2.2 et sur le pattern à trois niveaux proposé par la règle *C.1* (cf. chapitre III section 4.3). Par souci de clarté, seules apparaissent les classes qui interviennent dans la fonctionnalité de renouvellement de clé. Les couches Médium et PMR sont dessinées dans une syntaxe ad-hoc (différente de celle de TURTLE) pour des raisons de place et de clarté.

La demande de renouvellement de clé se fait par l'intermédiaire du superviseur (le chef de compagnie qui correspond à la classe *Chief_KeyRenCKR*). Cette demande peut se faire de manière automatique (toute les 24 heures) ou à partir d'une fonctionnalité incluant un renouvellement de clé, par exemple pour une fusion ou une descente en grade. Cette classe est ensuite connectée aux deux couches sous-jacentes *GMM* et *GCKM* (cf. section 2.1). La couche à vérifier dans cet exemple est celle qui est concernée par la génération et la distribution : la couche *GCKM*.

Pour tous les appels de fonctionnalités incluant le renouvellement de clés, la demande est effectuée par la couche *GMM* qui utilise le service de renouvellement de clés. Ces deux couches sont ensuite connectées aux couches *SO* et *PMR* représentant le service de diffusion sécurisée.

La vérification d'un modèle de médium à diffusion est un problème récurrent des techniques de modélisation basées sur une sémantique d'entrelacement. Le risque d'explosion combinatoire s'accroit avec un modèle temporel. C'est par une modélisation « astucieuse » que nous limitons ce risque en l'absence de rendez-vous multiple dans TURTLE. Ainsi, la structuration du modèle TURTLE du médium à diffusion *PMR* reproduit la hiérarchie (compagnie, domaine, utilisateur) du système SAFECAST. Appelée *RHV* [L 4.1] (*Routage Hiérarchique Virtuel*), cette technique explicitée en [L 4.1], permet – dans le cas d'une diffusion d'un message crypté concernant un sous-groupe d'utilisateurs suivie d'une diffusion d'un message concernant tous les utilisateurs – de voir le graphe d'accessibilité se construire en moins d'une minute alors qu'une heure ne suffisait pas à

construire le graphe en l'absence de RHV. Notons au passage que le RHV n'élimine pas du graphe d'accessibilité des chemins qui seraient importants pour les propriétés de sécurité à vérifier. La couche supérieure *Security Operators* (*SO*), offre les actions élémentaires de sécurité qui garantissent l'intégrité, la confidentialité et le non rejeu. Toutes ces opérations sont abstraites par des opérations temporelles, notre but étant de vérifier les exigences temporelles (cf. définition *HI.1* dans la section 2.1) et non pas les exigences liées à la sécurité et aux « données » (vérifiées dans le cadre du projet par l'outil AVISPA [AVI 05]).

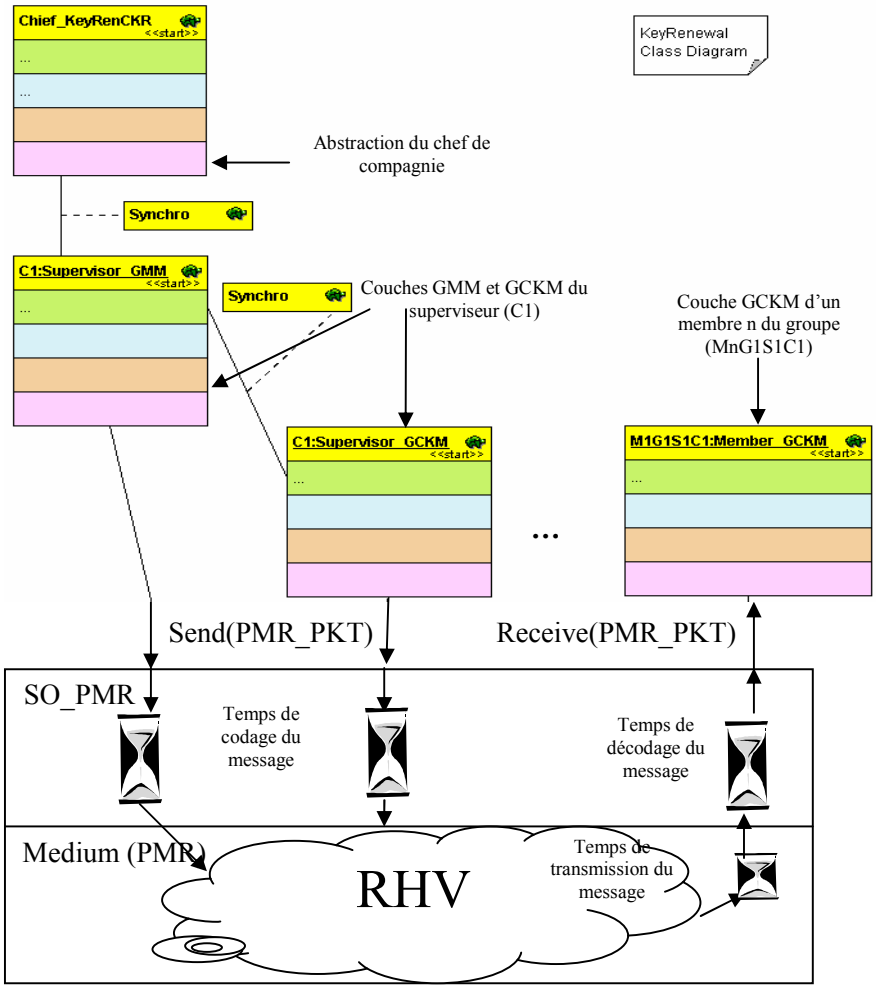

Fig.51. Diagramme de classes simplifié du protocole de gestion de clés

3.3. Construction des observateurs

Conformément à la définition des *points d'observations* (cf. section 3.1.3), le premier point d'observation *Begin_KR* est synchronisé avec la classe qui va générer et distribuer les clés, à savoir la classe représentant la couche GCKM du superviseur (soit la classe *Supervisor_GCKM*, cf. Fig.51). Le point d'observation *End_KR* est prélevé sur l'ensemble des utilisateurs au niveau de la couche GCKM (les classes *Member_GCKM*). Le prélèvement de ce dernier *point d'observation* pose des

problèmes liés à l'entrelacement entre les messages de synchronisation de l'observateur et les messages diffusés pendant le processus de distribution de la clé. Pour éviter ce type de problèmes, nous avons construit un observateur de type intrusif qui bloquera l'exécution du protocole à des moments appropriés (en début et fin de session). L'observateur collecte les informations concernant le temps de réception de la clé à la fin de la distribution ; le dispositif de capture du temps (une offre limitée dans le temps cf. chapitre V) est donc délocalisé dans la machine de protocole sans modification du comportement de celle-ci. Si l'exigence est violée, un attribut (que nous avons appelé *timeout*) est mis à vrai dans l'entité de protocole concernée : l'observateur collectera cette information à la fin de la distribution et enverra un message en cas de violation.

La Fig.52 correspond à des éléments de diagrammes d'activités de l'observateur et d'une entité quelconque du protocole de gestion de clés. Le dispositif de capture de temps (l'offre limitée dans le temps) est délocalisé dans les entités de protocoles, (cf. ellipse en pointillé sur la Fig.52).

Ce dispositif ne sert qu'à marquer l'attribut *timeout*. Si le message *Receive?PMR_TEK_Message* qui correspond à la nouvelle clé, est reçu après les 350 ms alors *timeout* sera mis à vrai ; sinon il reste à faux. Ce dispositif est construit à partir d'une offre limitée dans le temps. Il est donc nécessaire de synchroniser toutes les machines de protocole en début de session (dès l'occurrence du premier point d'observation *Begin_KR*) pour que le dispositif de capture du temps se synchronise au début du service de distribution de clés, permettant ainsi que l'observateur puisse délivrer un diagnostic correct.

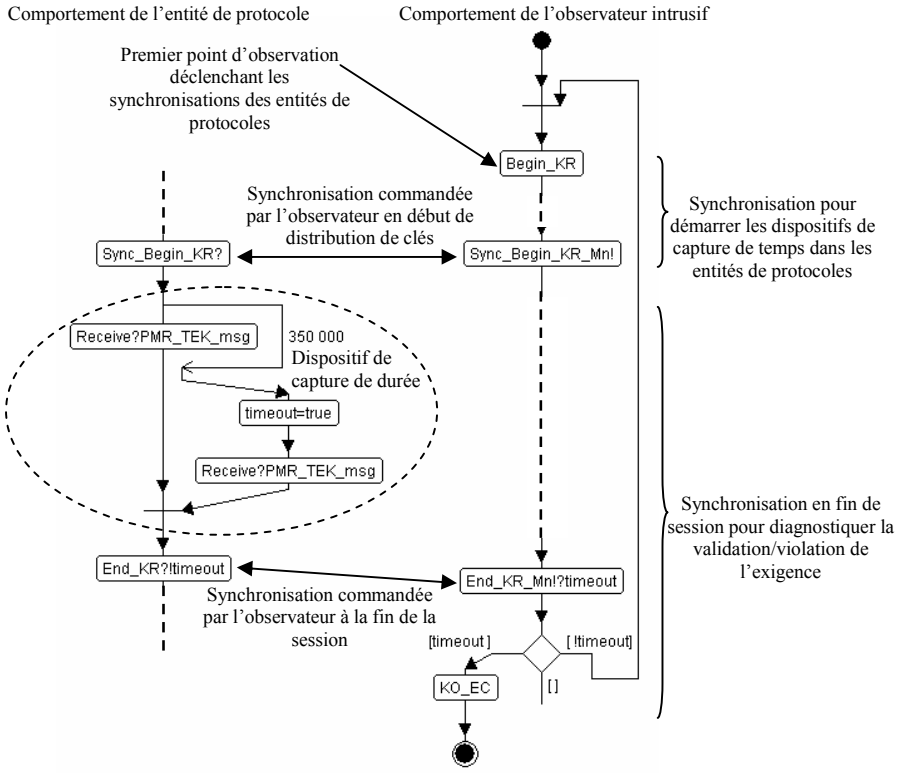

Fig.52. Diagramme d'activités d'une entité du protocole de gestion de clés et de l'observateur

L'observateur a un double emploi. Il synchronise, dans un premier temps, toutes les entités de protocoles de manière atomique et séquentielle (ce qui n'influe pas sur le comportement temporel du modèle) en début de renouvellement de clés. Puis, en fin de session, il échange des données par synchronisation avec toutes les entités de protocoles pour établir un diagnostic. Le dispositif de synchronisation en fin de session est indiqué dans la Fig.52 par la double flèche. Lorsque la session de renouvellement de clé s'achève, l'observateur va commander la synchronisation et demander la réception de *timeout* par l'action de synchronisation *End_KR_Mn !?timeout*. Cette expression est héritée de la sémantique LOTOS de passage de valeurs bidirectionnelles pendant une synchronisation. Le point d'exclamation correspond à une demande de synchronisation de l'observateur à l'entité de protocole et le point d'interrogation correspond à un passage de la valeur *timeout* de l'entité de protocole vers l'observateur. L'observateur va donc en fin de session interroger toutes les entités de protocoles individuellement et de manière séquentielle pour supprimer tout entrelacement des messages liés au diagnostic de l'observateur. Si l'exigence est violée, l'observateur stoppe les synchronisations avec les autres entités de protocoles. Ceci provoque l'arrêt de l'exécution du protocole car l'exigence temporelle du renouvellement de niveau de criticité « haut » a été violée.

L'exigence temporelle de la fonctionnalité de renouvellement de clé est satisfaite pour la configuration en moyen débit et violée pour la configuration en bas débit. Cette violation est expliquée principalement par le temps de transmission trop élevé dans la configuration en bas débit. Les messages contenant les clés de communication sont trop volumineux (de l'ordre du Kilo Octet) pour pouvoir être transmis en configuration bas-débit dans les délais demandés.

4. Contributions et résultats

L'instanciation de la méthodologie présentée dans ce mémoire a permis différentes contribution sur les livrables :

- [L 2.5] concerne la spécification globale du système en phase d'analyse (représenté par des diagrammes de cas d'utilisations, des diagrammes globaux d'interaction et par des diagrammes de séquences).
- [L 4.1] concerne la spécification de l'architecture (cf. section 2.2) du modèle TURTLE et la vérification qualitative des exigences temporelles.
- [L 3.4] concerne la vérification quantitative des exigences temporelles.

Notons qu'une contribution supplémentaire aurait pu être apportée au livrable [L 1.3] qui traite des exigences temporelles et de sécurité du protocole SAFECAST, concernant le recueil et la spécification des exigences respectivement par une base de données et des diagrammes d'exigences SysML. Cette contribution aurait permis de construire directement une matrice de traçabilité des exigences temporelles et de sécurité pour que ces dernières puissent être vérifiées de manière systématique.

Par ailleurs, le retour d'expérience de notre participation au projet SAFECAST nous a permis de participer sur les livrables :

- [L 4.3] qui concerne l'étude des outils de vérification des protocoles de sécurité, à partir de l'état de l'art de [FON 06a].
- [L 4.2] et [L 4.4] qui concernent la définition de points dur inhérents à la conception de protocoles de communication de groupes sécurisés.

Au niveau de la vérification des exigences temporelles, la conclusion que nous pouvons tirer (cf. [L4.2]) est que les mécanismes de sécurité proposés dans le projet SAFECAST sont trop exigeants vis-à-vis des contraintes du réseau à bas débit. Cette violation de contraintes est détectée par modélisation avant toute mise en œuvre, évitant ainsi de procéder à la réalisation d'un système dont on s'apercevrait lors de son utilisation, c'est à dire top tard, que les contraintes sont irréalisables. Dans le cas du bas débit, le concepteur devra choisir deux pistes : soit garder des mécanismes sécurisés mais avec des contraintes temporelles plus lâches, soit ne pas utiliser les mécanismes de sécurité s'il ne peut pas relâcher les contraintes. Dans le cas du moyen débit, toutes les exigences temporelles sont satisfaites, excepté le temps d'accès à un groupe multimédia.

Les résultats obtenus en [L 4.1], [L 4.3] et [FON 07] mettent en exergue la nécessité de compromis entre l'utilisation de fonctionnalités sûres mais coûteuses en temps par rapport au respect des performances. Ces résultats guident ainsi l'identification des compromis à réaliser et sur les choix à effectuer avant toute réalisation d'un protocole de communication de groupes sécurisés utilisant un médium sans fil [L 4.1].

5. Conclusion

Le projet SAFECAST [SFC] dédié à la communication de groupes sécurisés a fourni une étude de cas pour appliquer la méthodologique présentée dans ce mémoire. Ce projet a permis de définir des techniques générales pour la conception de protocoles de sécurité modélisés dans le profil UML temps réel TURTLE.

Nos contributions dans ce projet reposent d'une part sur l'application de la méthodologie présentée dans le chapitre III (pour les livrables [L 2.5], [L 4.1] et [L 3.4]) et de nos retours d'expériences de notre participation au projet (pour les livrables [L 4.3] et [L 4.2]). Enfin les résultats de la vérification des aspects temporels a permis de mettre en évidence la non-satisfaction de la majorité des exigences temporelles en configuration « bas débit » et la satisfaction (sauf une) de ces dernières dans une configuration moyen débit.

L'originalité de l'approche de vérification proposée dans le projet SAFECAST, a été de « croiser » les résultat d'une modélisation orientée « données » et « sécurité » à l'aide de l'outil de AVISPA [AVI 05] avec une modélisation orientée « contrôle » et « contraintes temporelles » mise en œuvre par l'outil TURTLE [APV 04]. Ceci a permis une collaboration entre les laboratoires LORIA et LAAS-CNRS, pour définir une méthodologie de conception de protocoles de sécurité basés sur la vérification couplée des exigences temporelles et de sécurité à l'aide des outils AVISPA et TURTLE.

Chapitre VII. Conclusions et perspectives

En dépit de ses treize diagrammes, le langage UML 2.1 normalisé par l'OMG n'offre aucune facilité particulière pour traiter convenablement la phase de traitement des exigences qui démarre le cycle de développement d'un système temps réel. Ce constat s'appliquait aussi aux profils UML, avant que ne soit normalisé le langage SysML qui amène avec lui l'idée de *diagrammes d'exigences*. Cependant ces deux langages manquent encore de support méthodologique. Ce constat est encore plus criant pour les profils UML/SysML adossés à des *méthodes formelles* qui impliquent une démarche rigoureuse sur la modélisation du système et la spécification des exigences pour fournir des résultats de vérification « exploitables ».

Dans cette optique, l'objectif premier de cette thèse a été de développer le volet « méthodologie » du profil UML temps réel TURTLE (*Timed UML and RT-LOTOS Environment*) [APV 04] orienté « vérification formelle d'exigences non-fonctionnelles temporelles ». La définition de cette méthodologie a permis de proposer une étape supplémentaire dans le processus méthodologique TURTLE : le *traitement des exigences*. L'insertion de cette nouvelle étape est le point de départ de la génération automatique d'observateurs dédiés à la vérification des exigences temporelles. Cette technique est en effet moins couteuse en termes d'outillage que l'approche de vérification des exigences basée sur le *model-checking*. De plus, un observateur, de par le fait qu'il est exprimé en UML, est réutilisable dans les phases de déploiement et codage. Enfin, la faisabilité de ces travaux a été évaluée, avec succès, dans le cadre d'un projet industriel (SAFECAST [SFC]).

1. Bilan des contributions

Les contributions de cette thèse ont été développées au travers des chapitre III à VI de ce mémoire. Ces contributions ont consisté à :

- **Définir une « méthodologie » de vérification d'exigences temporelles d'un modèle en contexte UML/SysML**. Notre première volonté dans ce mémoire est de fournir une assistance en termes de méthodologie à un utilisateur de profil UML/SysML temps réel préoccupé par la vérification formelle d'exigences temporelles. Nous nous sommes dégagés dans un premier temps de profils particuliers. Pour décrire de manière rigoureuse la méthodologie nous avons étendu les *diagrammes d'activité* UML pour définir un langage de description de processus méthodologique intégrant la notion de *production/consommation* de biens livrables (en l'occurrence des diagrammes UML et SysML). Des patterns de diagrammes UML et SysML (pour les phases de traitement des exigences, d'analyse de conception) sont ensuite proposés ainsi qu'un ensemble de règles de définition d'exigences, de construction de diagrammes d'analyse et de conception pour la conception de systèmes temps réel et distribués en considérant les exigences temporelles de *qualité de service* (en termes de *jitter* et *délai de bout en bout*). Enfin, la méthodologie, présentée dans le chapitre III, a pour finalité d'être instanciée dans le profil UML TURTLE. La Fig.53 présente un récapitulatif de nos contributions sur le

profil TURTLE en identifiant nos apports par rapport à l'existant avant les travaux présentés dans ce mémoire. Cette figure permet de se faire une idée rapide d'une partie des contributions présentées dans ce mémoire.

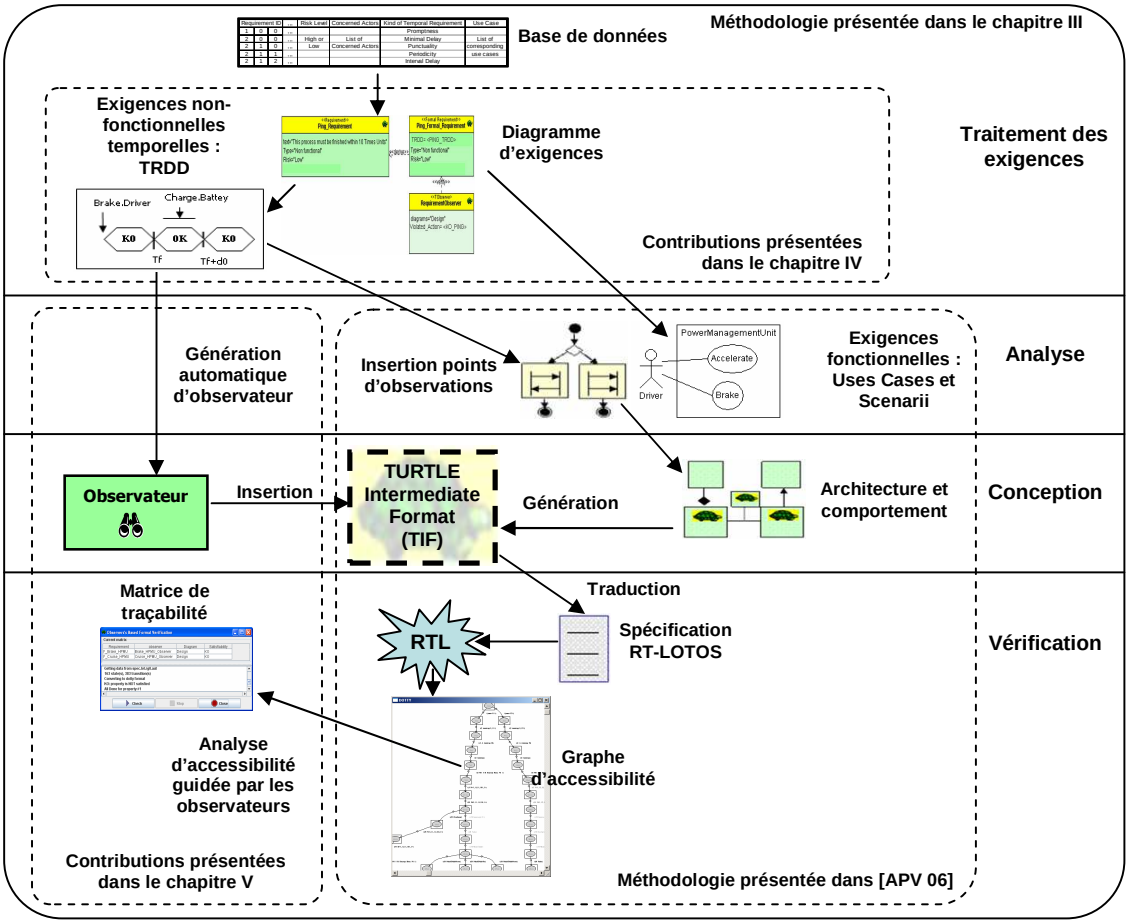

Fig.53. Récapitulatif de la méthodologie et présentation des contributions

- **Introduire une phase supplémentaire dans le processus méthodologique TURTLE : le *traitement des exigences***. La standardisation de SysML nous a amené à intégrer les *diagrammes d'exigences* dans le profil TURTLE. Ces diagrammes sont étendus en introduisant le stéréotype « *Formal_Requirement* » désignant une exigence formelle dérivée d'une exigence informelle. Les exigences non-fonctionnelles temporelles qui concernent la distance temporelle entre deux occurrences, sont décrites non pas, comme dans de nombreux travaux, par des formules de logiques ni des scénarios, mais par un langage visuel de type « chronogrammes » : TRDD (*Timing Requirement Description Diagram*). Ce langage a vocation à être utilisé hors du périmètre de TURTLE (cf. les définitions formelles et les méta-modèles UML/SysML). Cette contribution est présentée dans le chapitre IV.

- **Tracer les exigences non fonctionnelles temporelles par vérification guidée par observateurs**. L'ambition de ces travaux est de fournir un processus de traçabilité des exigences

temporelles reposant sur la vérification formelle des exigences, définies par les diagrammes d'exigences et TRDD, guidée par observateurs. Cette approche de vérification permet de pouvoir vérifier des exigences temporelles incluant des « distances temporelles ». Se souciant également d'avoir des travaux réutilisables, le processus de vérification est présenté sous forme de méta-modèles et d'algorithmes. Ces derniers ont été validés expérimentalement ; ils sont en cours d'implantation dans l'outil TTool (cf. chapitre V section 3.3). Cette contribution fait l'objet du chapitre V.

- **Appliquer ces travaux dans un projet industriel.** Notre participation dans le projet SAFECAST [SFC] sur la conception d'un protocole de communication de groupes sécurisée, nous a donné l'opportunité d'instancier nos travaux dans un modèle complexe. Ceci à permis de contribuer sur la production de biens livrables ([L 2.5] [L 4.1] [L4.2] [L 4.3] [L 3.4] [L 4.4]). Outre l'aspect validation temporelle du processus, l'originalité de l'approche de vérification proposée dans le projet SAFECAST, est de « croiser » les résultats d'une modélisation orientée « données » et « sécurité » à l'aide de l'outil de AVISPA [AVI 05] avec une modélisation orientée « contrôle » et « contraintes temporelles » mise en œuvre par l'outil TURTLE. Cette contribution est exposée dans le chapitre VI.

2. Perspectives

Nous ne saurions clôturer ce mémoire sans aborder les prolongements souhaitables à nos travaux.

2.1. Enrichir les descriptions d'exigences non-fonctionnelles temporelles

Cette perspective se décline sous deux aspects :

- Intégrer des relations de compositions temporelles entre exigences basées sur l'algèbre des intervalles d'Allen [ALL 83]. Les travaux de [ALL 83] définissent une algèbre des intervalles, décrivant les interactions possibles entre deux processus. [ALL 83] a identifié 13 schémas caractéristiques du positionnement respectif de deux intervalles temporels. Par le jeu de symétrie il ne reste que 7 schémas fondamentaux. Nous souhaiterions nous en inspirer pour définir des relations d'interaction non pas entre deux processus mais entre deux exigences dans le diagramme d'exigence en introduisant des relations de dépendance supplémentaires. Notons qu'une étude sur l'observation des relations basée sur l'algèbre des intervalles d'Allen entre processus a déjà été effectuée en [FON 06b].

- Coupler les approches de vérification guidée par observateurs et *model-checking*. L'implantation future d'une interface TTool [TTOOL] vers TINA [BER 04] sur la base des travaux de [SAD 07], permettra aussi d'enrichir notre « palette » d'expression des exigences non-fonctionnelles en termes de sûreté et de vivacité. Les travaux présentés dans ce mémoire concernent uniquement le sous-ensemble des exigences liés à la *vivacité bornée* [AH 93]. L'outil TINA offre une plateforme de validation dont un *model-checker* supportant la logique se-LTL [CHA 04] qui permet de définir des exigences en termes de *sureté* et de *vivacité*. Nous nous inspirerons des travaux du projet [TCAD] qui cherche en autres à convertir des diagrammes d'exigences en formules de logique se-LTL.

2.2. Extension de la méthodologie dans les phases de déploiement et de codage

Une de nos principales ambitions est de poursuivre la méthodologie proposée dans le chapitre III dans les phases de déploiement et de codage. Il faut noter qu'un observateur, exprimé en UML, est réutilisable dans les phases de déploiement et de codage. Il faudra envisager, dans un premier temps, d'entendre les diagrammes de déploiement TURTLE pour y intégrer des prototypes d'observateurs (des *sondes*) qui se grefferont dans le modèle de déploiement. Dans un second temps, nous pensons retravailler le générateur de code exécutable Java qui existe déjà dans l'outillage TURTLE afin que ces prototypes générés à partir d'un code exécutable « prouvé » soient soumis à un dispositif de simulation. La Fig.54 présente quelques réflexions autour du dispositif d'observation par sondes dans le modèle de déploiement. Nous distinguerons trois types de sondes :

- Des sondes correspondant à la mesure de la *Qualité de Service* (par exemple en termes de gigue et de délai de bout en bout). Ces dernières seront connectées avec la couche utilisant le service à valider.
- Des sondes pour les machines de protocole. Ces dernières mesureront les temps de traitement des différentes opérations (segmentation, réassemblent des données, …) de la couche à valider permettant d'évaluer l'efficacité des mécanismes pour la *Qualité de Service*. Ces sondes seront connectées avec la couche de protocole à valider.
- Des sondes pour mesurer l'état du réseau. Il faut utiliser, pour cela, les concepts liés à la *métrologie des réseaux* [OWE 05] pour évaluer l'état du réseau et donc analyser les performances du service.

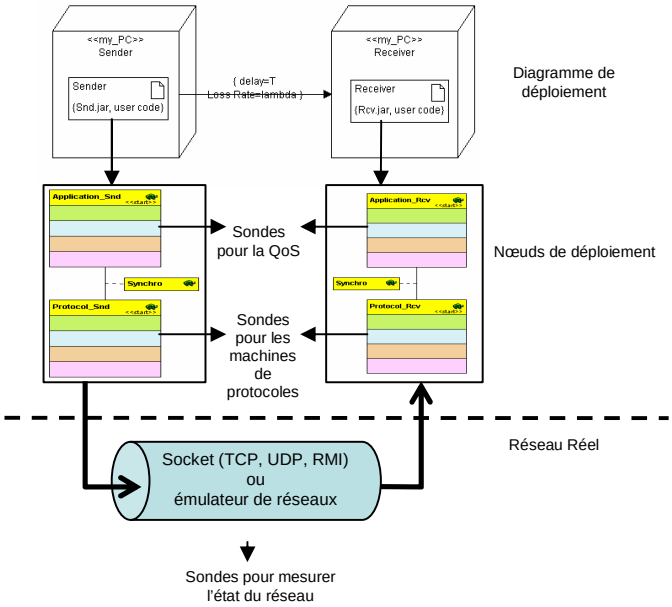

Fig.54. Caractérisation du dispositif d'observation pour le déploiement

L'aspect validation des exigences temporelles peut encore être étendu en ajoutant une facette supplémentaire dans la validation par vérification et simulation : l'utilisation d'outils d'émulation. Un couplage avec l'outil d'émulation W-NINE [DAI 07] semble possible, ce dernier proposant des API par socket pouvant être directement connectées au modèle TURTLE généré en java. L'*émulation*

active [GIN 05] permettrait des mesures beaucoup plus fines que la simulation, l'environnement pouvant être entièrement contrôlé dans un émulateur.

Enfin la construction de TRDD, peut permettre de générer automatiquement des jeux de test pour valider les exigences non-fonctionnelles temporelles dans la phase de mise en œuvre du système à développer. Cette perspective peut se placer dans le cadre des travaux de [ADJ 07] qui propose une approche de génération de séquences de test pour les réseaux de Petri temporels à chronomètres (RdPTC) [BER 04] complémentaire aux travaux de [SAD 07] qui présentent la génération de RdPTC à partir de modèles TURTLE.

2.3. Perspectives liées au projet SAFECAST

La participation au projet SAFECAST a permis d'ouvrir notre champ d'investigation à d'autres perspectives : définir un profil SECTURTLE. L'utilisation conjointe d'outils de vérification formelle des exigences non-fonctionnelles temporelles et de sécurité (respectivement les outils TTool [APV 04] et AVISPA [AVI 05]) à partir d'un modèle commun, nous motive à définir une extension orienté sécurité du profil TURTLE. Cette extension pourrait être traduite dans des spécifications formelles HPSL (*High level Protocol Specification Language*) qui seraient vérifiées par le biais de l'outil AVISPA [AVI 05]. L'état de l'art proposé dans le chapitre VI nous conforte dans ce choix : il n'existe pas à notre connaissance de profils UML de conception de protocoles de sécurité permettant de vérifier à la fois des exigences non-fonctionnelles relatives à la sécurité (aspect « donnée ») et aux contraintes temporelles (aspect « contrôle »). Ceci pourrait être fait en incluant tout comme le profil UMLSec [JUR 02] des stéréotypes intégrant des mécanismes de cryptographie. Ceci en incluant un processus automatique de vérification des exigences non-fonctionnelles liées à des propriétés de sécurité (par exemple la confidentialité, l'intégrité, la non-répudiation) qui sont proposés par l'outil AVISPA démarrant de la spécification de ces dernières dans un *diagramme d'exigences*.

Références

[ABR 96] J.R. Abrial, “The B Book, assigning programs to meanings” *Cambridge* UP, 1996.

[ACCOR] Page Web ACCORD UML, http://www-list.cea.fr/labos/fr/LLSP/accord_uml/AccordUML_presentation.htm.

[ADJ 07] N. Adjir, P. de Saqui-Sannes, M.K. Rahmouni, « Génération des séquences de test temporisées à partir des réseaux de Petri temporels à chronomètres », Conférence Internationale sur les NOuvelles TEchnologies de la REpartition (NOTERE 2007), pp.313-327, Marrakech, Maroc, Jun 2007.

[AGAT] Page Web AGATHA, http://www.iop.org/EJ/abstract/0954-3899/31/10/076

[AGIL] Page Web Manifeste AGILE http://agilemanifesto.org/

[ALB 07] A. Albinet, J.L. Boulanger, H. Dubois, M.A. Peraldi-Frati, Y. Sorel, Q-D. Van, “Model-based methodology for requirements traceability in embedded systems,” 3rd ECMDA workshop on traceability, Haifa, Israel, June 2007.

[ALL 83] J.F. Allen “Maintaining knowledge about temporal intervals”, Communication of the ACM 26(11) pp.832-843, Nov 1983.

[ALU 89] R. Alur, T.A. Henzinger, “A Really Temporal Logic,” Proceedings of the 13th Symposium on the Foundations of Computer Science, pp. 164-169, 1989.

[ALU 93] R. Alur, T. Henzinger, “Real-time logics: Complexity and expressiveness,” Information and Computation, (104), pp. 35-77, 1993.

[ALU 94] R. Alur, D. Dill, “A theory of Timed Automata,” Theoretical Computer Science, 126, pp. 183-235, 1994.

[AMI 05] Y. Amir, C. Nita-Rotaru, J. Stanton, G. Tsudik, “Secure Spread : An integrated Architecture for Secure Group Communication,” IEEE Transactions on Dependable and Secure Computing , September 2005.

[AML 99] N. Amla, E.A. Emerson, K.S. Namjoshi. “Efficient Decompositional Model Checking for Regular Timing Diagrams”, In Conference on Correct Hardware Design and Verification Methods (CHARME 1999). Springer-Verlag, pp. 67-81, Sep 1999.

[AND 06] C. André, A. Cuccuru, R. de Simone, T. Gautier, F. Mallet, J. Talpin. “Modeling with logical time in UML for real-time embedded system design,” in: MARTES, satellite workshop of Models'2006, Genova, Oct 2006.

[APV 04] L. Apvrille, J.-P. Courtiat, C. Lohr, P. de Saqui-Sannes , “TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit”, IEEE Transactions on Software Engineering, 30(7), pp. 473-487, July 2004.

[APV 06] L. Apvrille, P. de Saqui-Sannes, R. Pacalet, A. Apvrille, « Un environnement de conception de systèmes distribués basé sur UML », Annales des Télécommunications, Vol. 61, n 11/12, pp. 1347-1368, Nov 2006.

[AVI 05] Page web AVISPA. http://www.avispa-project.org

[BAB 89] J.P. Babau, J.L. Sourouille, “Expressiong Real Time Constraints in a Refective Object Model,” Control Engineering Practice, Vol 6, pp. 421-430, 1989.

[BAB 01] J.P. Babau, A. Alkhodre, “A development method for PROtotyping embedded SystEms by using UML and SDL (PROSEUS),” In workshop SIVOEES 2001 - ECOOP, Budapest, Hungary, 2001.

[BEH 99] P. Behm, P. Benoit, A. Faivre, J.M. Meynadier, “Météor: A Successful Application of B in a Large Project,” Lecture notes in computer science, ISSN 0302-9743, pp.712-724, 1999.

[BER 91] B. Berthomieu, M. Diaz, “Modeling and Verification of Time Dependant Systems Using Time Petri Nets,” IEEE Transaction on Software Engineering, 17(3), pp.259-273, 1991.

[BER 04] B Berthomieu, P.O. Ribet, F. Vernadat, “The TINA Tool: Construction Abstarct State Space for Petri Nets and Time Petri Nets,” International Journal of Production Research, Vol. 42, No. 14, pp.2741-2756, 2004.

[BLA 01] B. Blanchet, “An efficient cryptographic protocol verifier based on prolog rules,” In Proc 14th Computer Security Foundations Workshop (CSFW’01), pp. 82–96, Cape Breton, Canada, 2001.

[BOC 90] G. Bochmann, “Protocol specification for OSI,” Computer Networks and ISDN Systems, v.18 n.3, pp.167-184, April 1990.

[BOD 06] J.P. Bodeveix, P. Dissaux, M. Filali, P. Gaufilet, F. Vernadat, “Behavioural descriptions in architecture description languages, Application to AADL,” 3rd European Congress on Embedded Real Time Software (ERTS'06), Toulouse, France, Jan 2006.

[BOL 89] T. Bolognesi, E. Brinksma,“Introduction to the ISO specification language LOTOS,” in the Formal description Technique LOTOS, van Eijket al., Eds. Amsterdam, The Netherlands: North-Holland, pp. 23–73, 1989.

[BOO 00] G. Booch, J. Rumbaugh, I. Jacobson, « Le guide de l'utilisateur UML (version française) », c/o Eyrolles, ISBN 2-212-09103-6, 2000.

[BRA 05] V. Braberman, N. Kicillof and A. Alfonso. “A Scenario-Matching Approach to the Description and Model-Checking of Real-Time Properties“, IEEE Transactions on Software Engineering, special issue on interaction and state based modelling, 31(12), pp. 1028-1041, December 2005.

[BRA 06] J Bradfield, C. Stirling. “Handbook of Modal Logic,” chapter Modal Mu Calculi. Elsevier, 2006.

[BUC 04] M. Buchholtz, C. Montangero, L. Perrone, S. Semprini. “For-LySa: UML for authentication analysis”. In C. Priami and P. Quaglia, editors, Proceedings of the second workshop on Global Computing, LNCS 3267, pp. 92–105, 2004.

[BUC 05] A. Bucchiarone, S. Gnesi, P. Pierini, “Quality analysis of NL requirements: an industrial case study,” 13th IEEE International Conference on Requirements Engineering, pp. 390- 394, Paris, France, August 2005.

[CADP] Page web CADP, www.inrialpes.fr/vasy/cadp/

[CHA 04] S. Chaki, E.M. Clarke, J. Ouaknine, N. Sharygina, N. Sinha, “State/event-based software model checking,” Proceedings of the 4th International Conference on Integrated Formal Methods (IFM '04), LNCS 2999, pp. 128-15=47, 2004.

[CHO 05] H. Chockel and K. Fisler, “Temporal Modalities for Concisely Capturing Timing Diagrams”, Correct hardware design and verification methods, 13th IFIP WG 10.5 advanced research working conference, CHARME 2005, Saarbrücken, Germany, October 2005.

[CLA 86] E.M. Clarke, E.A. Emerson, A.P. Sistla, “Automatic verification of finite state concurrent systems using temporal logic,” ACM Trans. on Programming Languages and Systems, 8(2), pp. 244–263, 1986.

[CLA 99] E.M.Clarke, O Grumberg, D. Peled, “Model Checking,” MIT Press, Cambridge, 1999.

[CLA 00] R. Clarck, A. Moreira, “Use of E-LOTOS in Adding Formality to UML,” Journal of Universal Computer Science, Vol. 6, No. 11, pp.1071-1087, 2000.

[COR 05] V. Cortier, S. Delaune and P. Lafourcade,” A Survey of Algebraic Properties used in Cryptographic Protocols,” Journal of Computer Security, IOS Press, 2005.

[COU 91] J.P Courtiat, M. Diaz, V.B Mazzola, P. de Saqui-Sannes, « Description formelle de protocole et de services OSI en Estelle et Estelle* - Expérience et Méthodologie. » CFIP’91 Ingénierie des protocoles ; Pau, France, Sep 1991.

[COU 92] J.P. Courtiat and D.E Saidouni, “A Case Study in Protocol Design.” In E Brinksma , T Bolognesi and C.A Vissers, editors, Third LotoSphere Workshop & Seminar, Pise, Italy, Sep 1992.

[COU 00] J.P. Courtiat, C.A.S. Santos, C. Lohr, B. Outtaj, “Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique”, Computer Communications, Vol. 23, No. 12, pp. 1104-1123, 2000.

[DAI 07] L. Dairaine, G. Jourjon, E. Lochin, S. Ardon, “IREEL: Remote Experimentation with Real Protocols and Applications over Emulated Network,” ACM SIGCSE inroads (ACM Special Interest Group on Computer Science Education), Vol 39, N°2, June 2007.

[DAM 01] W. Damm and D.Harel. “LSCs: Breathing Life into Message Sequence Charts”, Formal Methods in Systems Design, 19, 45-80, 2001.

[DHA 07] P. Dhaussy, J.C. Roger, F. Boniol, « Mise en œuvre d'unités de preuve pour la vérification formelle de modèles», Ingénierie Dirigée par les Modèles (IDM’07), Toulouse, France, mars 2007.

[DIA 94] M. Diaz, G. Juanole, J.P. Courtiat. “Observer: a concept for formal on-line validation of distributed systems,” IEEE Transaction Software Engineering, 20(12) :900–913, 1994.

[DIL 92] D. L. Dill, A. J. Drexler, A J. Hu, C. H. Yang, “Protocol Verification as a Hardware Design Aid,” IEEE International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society, pp. 522-525, 1992.

[DOD 05] Y. Dodis, A. Smith, “Entropic Security and the encryption of High Entropic Message,” Theory of Cryptographic Conférence (TTC), February 2005.

[DOL 03] L. Doldi. “Validation of Communications Systems with SDL,” Wiley, 2003.

[DOU 02] B. Douglas, “Real-Time UML Tutorial,” OMG Real-Time and Embedded Systems Distributed Object Computing Workshop, Arlington, VA, USA, July 2002.

[DOU 04] B. Douglass, “Doing Hard Time: Developping Real-Time Systems with UML, Objects, Frameworks, and Patterns,” Addison Wesley, ISBN 0-201-49837-5, 2004.

[DUB 04] H. Dubois, N. Torrecillas, “Performance Evaluation of Real-Time Embedded Systems with the Accord/UML methodology”, 20th Annual UK Performance Engineering Workshop, University of Bradford, UK, 2004.

[EADS] Page web EADS http://www.eads.com/

[EAST] Page Web Projet EAST-EEA www.east-eea.net/

[EME 90] E. A. Emerson, “Temporal and modal logic.” In Jan van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B, chapter 16, pp. 997-1072. MIT Press, 1990.

[EXP 03] E. Exposito, M. Gineste, R. Peyrichou, P. Senac, M. Diaz, “XQOS: XML-based QoS specification language,” MMM’03 The 9th International Conference on Multi-Media Modeling, Taiwan, January 2003.

[EXP 04] E.Exposito, « Spécification et mise en œuvre d'un protocole de transport orienté Qualité de Service pour les applications multimédias », Doctorat de l’Institut National Polytechnique, Toulouse, France, décembre 2003.

[FDR 05] Page Web FDR2. http://www.verified.de/fdr.html

[FRA 01] M. Fränzle and K. Lüth, “Visual Temporal Logic as Rapid Prototyping Tool”, Computer Languages, Vol. 27, pp. 93-113, 2001.

[FON 04] B. Fontan, « Application d’UML 2.0 et du profil TURTLE à la modélisation et validation du protocole DCCP », Rapport DEA Système informatique, ENSICA, Toulouse, France, juin 2004.

[FON 06a] B.Fontan, S. Mota, T. Villemur, P. de Saqui-Sannes, J.P. Courtiat. “UML-based modeling and formal verification of authentication protocols,” Int. Symposium on Secure Software Engineering, Washington DC, USA, March 2006.

[FON 06b] B. Fontan, L. Apvrille, P. de Saqui-Sannes, J.-P. Courtiat, “Real-Time and Embedded System Verification Based on Formal Requirement”, IEEE Symposium on Industrial Embedded Systems (IES'06), Antibes, France, October 2006.

[FON 07] B. Fontan, S. Mota, P. de Saqui-Sannes, T. Villemur, “Temporal Verification in Secure Group Communication System Design,” International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2007), Valencia, Spain, October 2007.

[GER 02] S. Gérard, F. Terrier, Y. Tanguy, "Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML" in OOIS'02-MDSD, Montpellier, France, Springer, LNCS 2426, 2002.

[GER 04] S. Gérard, C. Mraidha, F. Terrier, B. Baudry, “A UML-based concept for high concurrency: the Real-Time Object,” The 7th IEEE International Symposium on Object-Oriented Real-Time distributed Computing (ISORC), Vienna, Austria, 2004.

[GIN 05] M. Gineste, H. Thalmensy, L. Dairaine, P. Senac, M. Diaz, “Active Emulation of a DVB-RCS Satellite Link in an End-to-end QoS-oriented Heterogeneous Network,” 23rd AIAA International Communications Satellite Systems Conference , Rome, Italy, September 2005.

[GOT 02] R. Gotzhein, F. Khendek, « Conception avec Micro-Protocoles », Colloque Francophone sur l’Ingenierie des Protocoles CFIP’02, Hermes Science, Montréal, Canada, 2002.

[GRA 05] S. Graff, I. Ober, I. Ober, “Validating Timed UML Models by Simulation and Verification”, Int. Journal On Software Tools for Technology Transfer, 2005.

[HAL 93] N. Halbwachs, F. Lagnier, P. Raymond, “Synchronous Observers and the Verification of Reactive Systems”, AMAST’93 (3rd Int. Conference on Algebraic Methodology and Software Technology), June 1993.

[HAT 87] DJ. Hatley, J. Derek, I.A. Pirbhai, “Strategies for real-time system specification,” Dorset House Pub, New York, 1987.

[HAS 06] J. Hassine, J. Rilling, R. Dssouli. “Timed Use Case Maps”, In System Analysis and Modeling: Language Profiles, 5th International Workshop, SAM 2006, Kaiserslautern, Germany, pp. 99-114, June 2006.

[HOL 03] G. Holzmann. “The SPIN model checker SPIN,” Addison-Wesley, 2003.

[HUL 04] E.Hull, K. Jackson, J. Dick, “Requirement Engineering – Second Edition,” Springer, ISBN 1-85233-879-2, 2004.

[HUT 04] M. Huth, M. Ryan, “Logic in Computer Science modelling and reasoning about systems (2nd edition),” Cambridge University Press, ISBN 0 521 54310X, 2004.

[ITU 96] ITU-TS “Recommendation Z.120: Message Sequence Chart (MSC),” ITU-TS, Geneva, Switzerland, 1996.

[IS 8807] ISO, “IS 8807: LOTOS: A Formal Description Technique,” 1989.

[IS 9074] ISO, “IS 9074: Estelle: A Formal Description Technique Based on an Extended State Transition Model,” 1989.

[JAH 86] F. Jahanian, A.K. Mok "Safety Ananysis of Timing properties in Real-Time Systems", IEEE Transactions on software engineering, Vol SE-12, N° 9, pp. 890-904, September 1986.

[JAN 99] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, P. Stappen, “Model Checking for Managers”, Spin’99, pp.92-107, 1999.

[JAR 88] C. Jard, J.-F. Monin, R. Groz, “Development of Veda, a Prototyping Tool for Distributed Algorithms”, IEEE Transactions on Software Engineering, Vol.14, No. 3, pp339-352, March 1988.

[JUR 02] J. Jürjens. “UMLsec: Extending UML for secure systems development.” In UML 2002- The Unified Modeling language, LNCS 2460, pp. 412-425, 2002.

[JUS 94] M. Just, “Methods of Multi-party Cryptographic Key Establishment,” Master thesis, School of Computer Science, Carleton University, Ontario, Canada, 1994.

[KEN 00] B. Kent, “Extreme Programming Explained, Embrace Change,” Addison-Wesley, ISBN 201-61641-6, 2000.

[L 1.3] EADS, ENST Paris, LAAS-CNRS, LORIA, UTC Compiègne. Projet RNRT SAFECAST. « L3.3. Définition d’un protocole de gestion de clés. » juin 2006.

[L 2.5] EADS, ENST Paris, LAAS-CNRS, LORIA, UTC Compiègne. Projet RNRT SAFECAST. « L 2.5 : Spécification du système global, intégration des services de sécurité au protocole de gestion de clés, », novembre 2005.

[L 3.4] EADS, ENST Paris, LAAS-CNRS, LORIA, UTC Compiègne. Projet RNRT SAFECAST. « L3.4: Evaluation des performances pour la validation du facteur d'échelle », janvier 2007.

[L 4.1] EADS, ENST Paris, LAAS-CNRS, LORIA, UTC Compiègne. Projet RNRT SAFECAST. « L 4.1 : Validation de l''architecture », septembre 2006.

[L 4.2] EADS, ENST Paris, LAAS-CNRS, LORIA, UTC Compiègne. Projet RNRT SAFECAST. « L4.2: rapport sur l'identification des points durs », novembre 2006.

[L 4.3] EADS, ENST Paris, LAAS-CNRS, LORIA, UTC Compiègne. Projet RNRT SAFECAST. « L 4.3 : identification et mise en œuvre des outils de simulation et vérification », octobre 2006.

[L 4.4] EADS, ENST Paris, LAAS-CNRS, LORIA, UTC Compiègne. Projet RNRT SAFECAST. « L4.4 : Bilan des contributions originales du projet SAFECAST et perspectives de recherche ouvertes », juin 2007.

[LAM 06] A. Van Lamsweerde, "Goal-Oriented Requirements Engineering", System Objectives to UML Models to Software Specifications, Wiley, 2006.

[LEP 00] S. Leppänen, M. Luukkainen, "Compositional Verification of a Third Generation Mobile Communication Protocol," ICDCS Workshop on Distributed System Validation and Verification, 2000: E118-E125, 2000.

[LOH 02] C. Lohr, P. de Saqui-Sannes, L. Apvrille, P. Sénac, J.-P. Courtiat, "UML and RT-LOTOS. An integration for real-time system validation," Journal Européen des Systèmes Automatisés (APII-JESA), Vol.36, N°36, pp.1029-1042, 2002.

[MAG 02] P. Maggi, R. Sisto, "Using SPIN to Verify Security Properties of Cryptographic Protocols," In Model Checking of Software: Proceedings of the 9th International SPIN Workshop, LNCS 2318, Grenoble, France, Apr 2002.

[MAL 06] F. Mallet , M. Peraldi-Frati , C. André, "From UML to Petri Nets for non functional Property Verification," in: First IEEE Symposium on Industrial Embedded Systems (IES'06), Antibes, France, Oct 2006.

[MeMV] Page web MeMVaTEx, www.memvatex.org/

[MIL 89] R. Milner, "Communication and Concurrency," Prentice Hall, 1989.

[MUR 89] T Murata. "Petri nets: Properties, analysis and applications," Proceedings of the IEEE,77(4), pp. 541–580, April 1989.

[MUS 07] G. Mussbacher, D. Amyot, M. Weiss, "Formalizing Patterns with the User Requirements Notation," Design Pattern Formalization Techniques, T. Taibi (Ed.) IGI Global, pp. 304-325, March 2007.

[OBE 99] I. Ober, "Using goal observers to extends the geode simulator," Technical report, VERILOG, 1999.

[OBJ] Page web Outil Objectiver, http://www.objectiver.com/

[OMEG] Page Web OMEGA www-omega.imag.fr/profile.php

[OMG] Page Web OMG www.omg.org/

[OMG 05] OMG, SPT Profile, "UML Profile for Schedulability, Performance, and Time, v1.1." January 2005. www.omg.org/docs/formal/05-01-02.pdf

[OMG 07] OMG, MARTE profile, "UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE)," July 2007. http://www.omg.org/docs/ptc/07-08-04.pdf

[OWE 07] P. Owerzarski, N. Larrieu, « Techniques et outils de métrologie pour l'internet et son trafic », Techniques de l'ingénieur, pp. R 1090-1-R 1090-22, juin 2007.

[PEH 90] B. Pehrson, “Protocol verification for OSI,” Computer Networks and ISDN Systems, 18(3), pp.185-201, April 1990.

[PHA 04] T. H Phan, S. Gerard, S., F. Terrier, “Real-time system modeling with ACCORD/UML methodology: illustration through an automobile case study.” In Languages For System Specification: Selected Contributions on Uml, Systemc, System Verilog, Mixed-Signal Systems, and Property Specification From Fdl'03, C. Grimm, Ed. Kluwer Academic Publishers, pp. 51-70, Norwell, MA, 2004.

[PMR] Page web PMR http://www.eads.com/1024/fr/businet/defence/dcs/solutions/pmr/pmr.html

[POP 06] M. Popović, “Communication Protocol Engineering,” Computer network protocols, CRC Press, ISBN 0849398142, 2006.

[RATIO] Page Web RATIONAL www-306.ibm.com/software/rational/uml/

[RAS 03] A. Rashid, A. Moreira, J. Araújo, “Modularisation and composition of aspectual requirements,” Proceedings of the 2nd int. conference on Aspect-oriented software development, pp.11-20, Boston, USA, March 2003.

[ROQ 04] P. Roques, F. Vallée, « UML en action, 2e édition : De l'analyse des besoins à la conception en Java », éditions Eyrolles, 2004.

[ROG 06] J.C. Roger, « Exploitation de contextes et d’observateurs pour la validation formelle de modèles. », Mémoire de doctorat, Ecole Nationale Supérieure des Télécommunications, Brest, décembre 2006.

[RUM 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, “Object-Oriented Modeling and Design,” Prentice Hall, 1991.

[RTL] Page Web Real-Time Lotos Laboratory (RTL), http://www.laas.fr/ RT-LOTOS

[RUS 04] M. Rusinowitch, “A Decidable Analysis of Security Protocols,” 18th IFIP World Computer Congress on Theoretical Computer Science - TCS'2004, 2004.

[SAD 07] T. Sadani, « Vers l'utilisation des réseaux de Petri temporels étendus pour la vérification de systèmes temps-réel décrits en RT-LOTOS », Mémoire de doctorat, Institut National Polytechnique, Toulouse, mai 2007.

[SCO 06] W.A Scott. “Introduction to the Diagrams of UML 2.0.”, www.agilemodeling.com/essays/umlDiagrams.htm.

[SDL 96] ITU-T, “Recommendation Z.100, Specification and Design Language (SDL)”, 1996.

[SEL 01] B. Selic, “A UML Profile for Modeling Complex Real-Time Architecture,” www.omg.org/news/meetings/workshops/presentations/realtime2001/6-3_Selic.presentation.pdf

[SFC] Page web Projet SAFECAST http://rnrt-safecast.org/

[STE 93] B. Stefani, “Computational Aspects of QoS in an object-based, distributed systems architecture,” 3rd Workshop on Responsive Computer systems, Lincoln, NH, USA, September 1993.

[STE 02] M. Steiner, “Secure Group Key Agreement”, Ph.D. thesis, Saarland University, March 2002.

[STI 96] C. Stirling. “Modal and temporal logics for processes,” proc. VIII Banf. Higher order workshop conference on Logics for concurrency: structure versus automata, pp. 149–237, Secaucus, New York, USA, 1996.

[SysML] Object Management Group, Norme SysML, Version 1.0, http://www.SysML.org/docs/specs/SysML-v1-Draft-06-03-01.pdf

[TAR 83] H. Tardieu, A. Rochfeld, R. Colletti, « La méthode Merise - Tome 1 Principes et outils », Editions d'organisation, ISBN 2-7081-1106-X., 1983.

[TAU] Page web Telelogic Tau G 3.0 http://www.telelogic.fr/products/tau/g2/index.cfm

[TCAD] Projet TopCAseD, http://www.topcased.org/

[TETRA] Page web TETRAPOL http://www.eads.net/1024/fr/eads/world_of_eads/products/pmr/pmr.html

[TFRC] Draft TFRC www.ietf.org/internet-drafts/draft-ietf-dccp-tfrc-faster-restart-03.txt

[TOU 97] J. Toussaint, F. Simonot-Lion, « Vérification formelle de propriétés temporelles d'une application distribuée temps réel, » In Real-Time Systems - RTS'97, Teknea, pp. 53-66, 1997.

[TOU 05] J.C. Tournier, V. Olive, J.P. Babau. "A Qinna Evaluation, a Component-Based QoS Architecture for Handheld Systems." In ACM Symposium on Applied Computing, SAC 05, Santa Fe, USA, March 2005.

[TTOOL] Page web TURTLE Toolkit (TTool) http://labsoc.comelec.enst.fr/turtle/

[UPPA] Page web UPPAAL, http://www.uppaal.com/

[UML] Object Management Group, Norme UML, Version 2.1.1, "Unified Modeling Language Specification", Version 2.1.1, http://www.omg.org/docs/formal/07-02-03.pdf.

[VAR 86] M.Y. Vardi, P. Wolper, "An automata–theoretic approach to automatic program verification," Proc. IEEE Symp. on Logic in Computer Science, pp. 332–344, Boston, USA, June 1986.

[WAH 94] T. Wahl, K. Rotherme, "Representing Time in Multimedia Systems." International Conference on Multimedia Computing and Systems (ICMCS'94), pp.538-543, Boston, USA, 1994.

[WOL 91] P. Wolper, 'Introduction à la calculabilité," InterEditions, Paris, France,1991.

[WUA 05] B. Wua, J. Wua, E.B. Fernandeza, M. Ilyasa, S. Magliveras, "Secure and efficient key management in mobile ad hoc networks", Journal of Network and Computer Applications, Elsevier, 2005.

[ZIM 80] H. Zimmerman, "OSI Reference Model; The ISO Model of Architecture for Open Systems Interconnection." IEEE Transactions on Communications, 28(4), pp. 425-732, Apr 1980.

Annexes

A. Vérification de la syntaxe et de la sémantique d'un TRDD

A.1. Algorithme de vérification de la syntaxe d'un TRDD

Algorithme A1 Vérification de la syntaxe d'un TRDD

```
int verif_syntax_TRDD(int[] TRDD_i, int[] T_in) {
            int erno=0;
            //Test caractere begin
            if (!(TRDD_i[0]== 0)){
            System.out.println("erreur TRDD doit commencer par begin");
            erno=1;}
            //si dernier caractere different de end
            if (!(TRDD_i[TRDD_i.length-1]== 4)){
            System.out.println("erreur TRDD doit finir par end");
            erno=2;}
            else{
                  //Parcourir TRDD
                  for (int i =1 ; i<TRDD_i.length ; i++){
                  //Bornage des valeurs
                  if((TRDD_i[i]<0)||(TRDD_i[i]>4)){
                  System.out.println("erreur symbole" + TRDD_i[i] + "n'existe pas
                  ");erno=3;
                  }
                  //regle debut suivi de OK ou KO
                  if(TRDD_i[i-1]==0) {
                        if(!((TRDD_i[i]==2)||(TRDD_i[i]==3))){
                              System.out.println("erreur symbole OK ou KO apres begin
                              ");erno=4;
                        }
                  }
                  //regle FT suivi de OK ou KO
                  if(TRDD_i[i-1]==1){
                        if(!((TRDD_i[i]==2)||(TRDD_i[i]==3))){
                              System.out.println("erreur symbole OK ou KO apres FT
                              ");erno=4;
                              }
                  }
                  //regle OK ou KO suivi de FT
                  if((TRDD_i[i-1]==3)||(TRDD_i[i-1]==2)){
                                    if(!((TRDD_i[i]==1)||(TRDD_i[i]==4))){
                                          System.out.println("erreur symbole FT apres
                                          OK ou KO ");erno=4;
                                    }
                  }
                  //regle de placement de end
                  if (TRDD_i[i-1]==4) {
                              System.out.println("erreur symbole end a la fin ");erno=2;
                              }
                        }
                  }
            //test consistance des valeurs temporelles
            for (int i =1 ; i<T_in.length ; i++){
```

```
                if (T_in[i] <= T_in[i-1]) {
                        System.out.println("erreur dans les valeurs temporelles valeur
                        "+ T_in[i]);erno=5;
                        }
                }
                return erno;
        }
```

Glossaire: `TRDD_i` = tableau d'entiers représentant les éléments de la ligne de vie du TRDD
`T_in` = tableau d'entiers contenant les valeurs des dates temporelles du TRDD
`erno` = code d'erreur produit en sortie de l'algorithme

A.2. Algorithme de vérification de la sémantique d'un TRDD

Algorithme A2 Vérification de la sémantique d'un TRDD

```
int verif_semantic_TRDD(int[] label_i) {
            int erno=0;
            for (int i =1 ; i<label_i.length ; i++){
                  if (label_i[i-1]==1)
                  {if(label_i[i]==1) {
                              System.out.println("erreur de semantique deux OK a la
                              suite");erno=6;
                        }
                  }
                  else if(label_i[i]==0) {
                        System.out.println("erreur de semantique deux KO a la
                        suite");erno=7;
                  }
            }
            return erno;
        }
```

Glossaire: `Label_i` = tableau généré à partir de l'algorithme 2 (cf. chapitre V section 2.3.3)
`T_in` = tableau d'entiers contenant les valeurs des dates temporelles du TRDD
`erno` = code d'erreur produit en sortie de l'algorithme

ABSTRACT:

The Unified Modeling Language (UML) standardized by the Object Management group (OMG) offers thirteen diagrams, but no facility to handle the requirement management phase which usually starts the development cycle of real-time systems. The SysML standard, including the concept of requirement diagram, has opened new avenues. Nevertheless both UML and SysML lack methodological support.

This dissertation aims therefore to develop a methodology for real-time UML profiles as soon as the latter cover the upper phases (requirement capture – analysis – design) in the development cycle of real-time and distributed systems, and gives prime importance to formal verification of temporal requirements. The proposed methodology is instantiated on the TURTLE (Timed UML and RT-LOTOS Environment) profile. SysML requirement diagrams are extended with a chronogram-based visual language (TRDD = Timing Requirement Description Diagram). They are used to described non functional and temporal requirements. The latter are the starting point for automatic generation of observers whose role is to verify the requirements in question. The contributions presented throughout the dissertation are formalized and defined by meta-models. The results are not restricted to TURTLE. The proposed approach has been applied to secure group communication protocols (SAFECAST RNRT project).

KEYWORDS:

Methodology, Real-Time UML, SysML, Temporal Requirements, Formal Verification, Observers, Protocols, Secure Group Communication.