

Contribution au prototypage virtuel de systèmes mécatroniques basé sur une architecture distribuée HLA. Expérimentation sous les environnements OpenModelica-OpenMASK

Hassen Hadj Amor

▶ **To cite this version:**

Hassen Hadj Amor. Contribution au prototypage virtuel de systèmes mécatroniques basé sur une architecture distribuée HLA. Expérimentation sous les environnements OpenModelica-OpenMASK. Sciences de l'ingénieur [physics]. Université du Sud Toulon Var, 2008. Français. NNT : . tel-00437932

HAL Id: tel-00437932
https://theses.hal.science/tel-00437932

Submitted on 1 Dec 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Présentée pour obtenir le titre de

Docteur de l'Université du Sud Toulon Var
Spécialité : Sciences et techniques industrielles

Par

Hassen Jawhar HADJ-AMOR

Maîtrise informatique de la faculté des sciences de Monastir
Master Recherche SIS ENSAM

Contribution au prototypage virtuel de systèmes mécatroniques basé sur une architecture distribuée HLA. Expérimentation sous les environnements OpenModelica-OpenMASK

Sous la direction de : M. Thierry SORIANO

Soutenue le 04 Décembre 2008 devant le jury composé de :

Mme Claire VALENTIN	Université de Lyon	Rapporteur
Mr Pierre SIRON	ISAE-Toulouse	Rapporteur
Mme Isabel DEMONGODIN	Université de Marseille	Examinateur
Mr Eric MOREAU	ISITV-Toulon	Examinateur
Mr Raouf FATHALLAH	ENISO-Sousse	Examinateur
Mr Thierry SORIANO	SUPMECA-Toulon	Directeur de thèse

Thèse préparée au Laboratoire LISMMA (EA 2336) Supmeca Toulon
dans l'équipe : Ingénierie Intégrée des Systèmes Industriels et Mécatroniques.

Remerciements

Les travaux présentés dans ce mémoire ont été réalisés au Laboratoire d'Ingénierie des Systèmes Mécaniques et des MAtériaux (LISMMA) à l'institut SUPérieur de MECAnique de Toulon (SUPMECA).

Je voudrais remercier toutes les personnes qui m'ont soutenu, de près et de loin, en particulier:
– *Monsieur* Thierry SORIANO, Maître de conférences HDR au LISMMA pour m'avoir accueilli dans son laboratoire et soutenu tout au long de la thèse, qui a consacré son temps pour encadrer ces travaux. Merci pour ses conseils scientifiques, l'encouragement et la confiance qu'il m'a accordés.
– *Madame* Claire VALENTIN, Professeur à l'Université Claude Bernard Lyon 1 au laboratoire LAGEP pour l'honneur qu'elle m'a fait en acceptant d'être rapporteuse de ce mémoire. Je la remercie pour les remarques très pertinentes apportées à cette thèse.
– *Monsieur* Pierre SIRON, Professeur de l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE/DMIA), pour avoir accepté d'être rapporteur de cette thèse et pour les conseils très intéressants qu'il m'a adressés.
– *Madame* Isabel DEMONGODIN, Professeur de l'Université Paul Cézanne au laboratoire LSIS, pour avoir accepté de me faire l'honneur de faire partie du jury.
– *Monsieur* Eric MOREAU, Professeur de l'ISITV au laboratoire LSEET, pour avoir bien voulu participer à ce jury.
– *Monsieur* Raouf FATHALLAH, Maître de conférences HDR à L'Ecole Nationale d'Ingénieur de Sousse (ENISO), pour l'honneur qu'il m'a fait en acceptant de participer à ce jury.
– *Monsieur* Alain RIVIERE, *Monsieur* Jean-Yves CHOLEY et *Monsieur* Régis PLATEAUX du LISMMA à SUPMECA Paris pour les discussions scientifiques et les échanges qui nous ont permis de mieux appréhender le sujet de la thèse.
– *Monsieur* Skander TURKI docteur de l'Université du Sud Toulon-Var et *Monsieur* Baptiste ARNAULT doctorant au LISMMA à SUPMECA Toulon pour leur soutien moral et les discussions scientifiques qui m'ont permis de découvrir des sujets de recherches variés.
– *Madame* Sabine SELLIER, qui a relu attentivement ce manuscrit. Je la remercie pour le temps qu'elle a consacré à redonner un peu de rigueur à ma plume qui a quelques fois tendance à déraper.

Je tiens à remercier très sincèrement le personnel administratif et scientifique du LISMMA, en particulier : *Madame* Pascale AZOU-BRIARD, *Monsieur* Marc BRIARD, *Madame* Lyudmyla YUSHCHENKO, *Monsieur* Olivier TORREMOCHA, *Monsieur* Christian TOUSSAINT, *Madame* Ingrid DESCAREGA, *Madame* Sabine SEGAL, *Monsieur* Jean-Louis CAMPELLONI, *Madame* Valéry GALLENE pour leurs compétences, leur gentillesse et leur disponibilité.

Pour conclure, je remercie et je dédicace cette thèse à toute la famille, d'abord ma mère et mon père pour leur support irremplaçable et inconditionnel et pour leur soutien sans faille et permanent malgré la distance qui nous sépare. Ils ont été présents pour écarter les doutes, soigner les blessures et partager les joies. Cette thèse est un peu la leur également. Je remercie mes sœurs Fatma et Meriam pour leur gentillesse et leur affection, et aussi ma fiancée Monia de m'avoir tenu la main jusqu'aux dernières lignes de ce mémoire. Merci d'être là tous les jours.

Table des matières

1. Chapitre 1 : Problématique 9
2. Chapitre 2 : Modélisation et simulation des systèmes mécatroniques 14
2.1 Introduction 14
2.1 Les systèmes mécatroniques 14
2.3 Modélisation du comportement: les systèmes dynamiques hybrides 17
2.3.1 Les automates hybrides 19
2.3.2 Les réseaux de Petri 22
2.3.3 Grafcet hybride 26
2.3.4 Autres approches 27
2.3.5 Choix du formalisme 28
2.4 Environnements de simulation des systèmes dynamiques hybrides 29
2.4.1 Introduction et vue d'ensemble des outils et langages de modélisation pour la simulation des systèmes hybrides 29
2.4.2 Le language multi physique support : Modelica 33
2.4.3 OpenModelica 45
2.4.4 Mathmodelica 48
2.4.5 Dymola 50
2.4.6 Choix de l'outil 52
2.5 Environnements virtuels 53
2.5.1 OpenMask 54
2.5.2 VrJuggler 56
2.5.3 3dVia Virtools 59
2.5.4 Choix du simulateur 3D 61
2.6 Conclusion 61
3. Chapitre 3 : Simulation distribuée 62
3.1 Introduction 62
3.2 Simulation : définition, intérêts et dangers 63
3.2.1 Définitions 63
3.2.2 Intérêts de la simulation 64
3.2.3 Dangers de la simulation 64
3.3 Simulation à évènements discrets 66
3.3.1 Le temps 66
3.3.2 Temps réel et temps réel mis à l'échelle 66
3.3.3 Mécanismes d'exécution des évènements 68
3.4 Simulation distribuée à évènements discrets 71
3.5 Conclusion 73
4. Chapitre 4 Les architectures pour la simulation distribuée 74
4.1 Introduction 74
4.2 DIS 76
4.2.1 Architecture de DIS 76
4.2.2 PDU (Protocol Data Unit) 80
4.3 ALSP 85
4.4 HLA 86
4.4.1 Architecture générale de HLA 87
4.4.2 Les spécifications de HLA 88
4.4.3 L'infrastructure de simulation RTI 99
4.4.4 Composants d'un fédéré HLA 100

4.4.5 Exécution d'une fédération 100
4.4.6 Caractéristiques de l'architecture HLA 102
4.5 Comparaison des architectures de simulation distribuée 103
4.5.1 Comparaison générale 103
4.5.2 Comparaison entre HLA et DIS 104
4.6 Outils et méthodologies HLA 106
4.6.1 CERTI 107
4.6.2 Travaux HLA 111
4.7 La gestion du temps sous HLA 117
4.7.1 Les évènements TSO et RO 118
4.7.2 Lookahead 119
4.7.3 LBTS (Lower Bound of the Time Stamps of messages) 120
4.7.4 Avancement du temps 121
4.8 Conclusion 123
5. Chapitre 5: Choix de distribution des modèles sous OpenModelica et implémentation des automates hybrides 124
5.1 Introduction 124
5.2 Choix de modèles et d'architecture sous OpenModelica 124
5.3 Implémentation des automates hybrides sous OpenModelica 126
5.3.1 Modélisation Orienté Objet 126
5.3.2 Modélisation d'un état 127
5.3.3 Modélisation de la transition 128
5.3.4 Description de la dynamique 130
5.3.5 Exemple 130
5.3.6 Modélisation du couplage 131
5.3.7 Implémentation de la méthode et description de la librairie HybridAutomataLib 134
5.4 Conclusion 142
6. Chapitre 6 : Intégration des services de HLA dans les simulateurs envisagés 143
6.1 Introduction 143
6.2 Conception de la fédération 144
6.2.1 Première approche 145
6.2.2 Deuxième approche 145
6.2.3 Choix de l'approche 146
6.3 Méthodes générales d'intégration 147
6.4 Intégration des services de HLA dans OpenModelica 148
6.4.1 Les Sockets 149
6.4.2 Communication entre OpenModelica et sa passerelle 151
6.4.3 Description de la dynamique 172
6.5 Intégration des services de HLA dans OpenMASK 173
6.6 Description de la dynamique globale des deux simulateurs 178
6.7 Conclusion 179
7. Chapitre 7: Gestion du temps et contraintes temporelles 180
7.1 Introduction 180
7.2 Synchronisation des simulateurs au début de la simulation 180
7.3 Stratégie et degré d'implication des fédérés 184
7.4 Module temps réel pour OpenModelica 185
7.5 Évolution du temps durant la simulation 190
7.6 Conclusion 191
8. Chapitre 8 : Application 192
Conclusion générale et perspectives 198

Bibliographie..... 201
Annexes..... 213

Table des figures

Figure 2.1 Exemple d'automate hybride modélisant une machine simple 20
Figure 2.2: Exemple de RdP hybride 24
Figure 2.3 Exemple d'héritage entre classes de Modelica 36
Figure 2.4 Classe Pin..... 39
Figure 2.5 Un composant avec un connecteur pour la mécanique..... 39
Figure 2.6 La fonction sample() de Modelica 42
Figure 2.7 La fonction noEvent() de Modelica 43
Figure 2.8 La fonction edge() retourne vrai quand x passe de faux à vrai. 44
Figure 2.9 Comportement de la fonction change()..... 44
Figure 2.10 Vue d'ensemble d'OpenModelica 47
Figure 2.11 Etapes de traduction du code Modelica vers un fichier exécutable 48
Figure 2.12 Architecture de l'environnement MathModelica..... 48
Figure 2.13 Architecture de Dymola..... 51
Figure 2.14 Les objets sous OpenMASK..... 55
Figure 2.15 arbre de simulation..... 55
Figure 2.16 Architecture de OpenMASK 56
Figure 2.17 Architecture de VRJuggler 58
Figure 3.1 Classification des simulations..... 68
Figure 3.2 Boucle de base pour une simulation temps réel dirigée par le temps 69
Figure 3.3 Problème de causalité 72
Figure 4.1 Standards de simulation distribuée 75
Figure 4.2 Structure de l'architecture DIS 77
Figure 4.3 Architecture ALSP..... 85
Figure 4.4 Fédérés et RTI d'une simulation HLA 87
Figure 4.5 Le cycle de vie d'une simulation HLA..... 98
Figure 4.6 Vue logique des composants d'un RTI..... 99
Figure 4.7 Composants d'un fédéré 100
Figure 4.8 Exécution d'une fédération..... 102
Figure 4.9 L'architecture du CERTI 108
Figure 4.10 Scénario de transfert des données..... 109
Figure 4.11 Application billard avec CERTI 111
Figure 4.12 Vue d'ensemble de l'outil GENESIS 113
Figure 4.13 Architecture oRisDis..... 116
Figure 4.14 Composants du gestionnaire oRisRTI 116
Figure 4.15 Lookahead et LBTS [Raulet]..... 121
Figure 4.16 Avancement du temps logique : time-step..... 122
Figure 4.17 Avancement du temps logique : basé évènement 123
Figure 5.1 Approche globale..... 125
Figure 5.2 Modèle d'un état 127
Figure 5.3 Modèle d'une transition 129
Figure 5.4 Modélisation orienté Objet 130
Figure 5.5 Exemple d'automate hybride simulé sous Modelica 131

Figure 5.6 Exemple de couplage 132
Figure 5.7 Exemple de couplage transformé 132
Figure 5.8 Exemple de connecteur entre deux états 133
Figure 5.9 Classe connector2 135
Figure 5.10 Résolution des équations de deuxième ordre sous Modelica 137
Figure 5.11 Une situation avec 3 transitions entrantes 138
Figure 5.12 La classe générique state 139
Figure 5.13 La classe générique Transition 140
Figure 5.14 Le modèle state et son interface graphique sous Dymola 141
Figure 5.15 Le modèle Transition et son interface graphique sous Dymola 142
Figure 6.1 Exemple d'une fonction externe. 149
Figure 6.2 Communication entre client et serveur en mode connecté 151
Figure 6.3 Vue de CommunicationLib dans l'explorateur de Dymola 152
Figure 6.4 Fonction enveloppante createSocket() 153
Figure 6.5 Fonction externe createSocket() 155
Figure 6.6 Fonction enveloppante sendMessage() 155
Figure 6.7 Fonction externe sendMessage() 156
Figure 6.8 Fonction enveloppante receiveMessage() 156
Figure 6.9 Fonction externe receiveMessage() 157
Figure 6.10 Fonction enveloppante recMsgBlock() 158
Figure 6.11 La fonction externe recMsgBlock() 160
Figure 6.12 Fonction enveloppante clean() 160
Figure 6.13 Fonction externe clean() 161
Figure 6.14 Modèle Socket 162
Figure 6.15 Echantillonnage et envoie de données 163
Figure 6.16 Boucle de blocage et de déblocage de OpenModelica 164
Figure 6.17 Serveur TCP / OpenModelica 165
Figure 6.18 Fédéré OpenModelica 167
Figure 6.19 La fonction join() 168
Figure 6.20 Appel de la fonction join() depuis la passerelle 169
Figure 6.21 Appel de la fonction getHandle() 169
Figure 6.22 Exemple d'implémentation de la fonction getHandle() 170
Figure 6.23 Appel de la fonction publishAndSubscribe() dans la passerelle 170
Figure 6.24 Boucle de simulation de la passerelle 171
Figure 6.25 Module HLAC 174
Figure 6.26 Fonction de callback générale receiveInteraction() 176
Figure 6.27 Diagramme de séquence des objets lors de la simulation 179
Figure 7.1 Synchronisation au niveau du module HLAC 181
Figure 7.2 Implémentation de la procédure d'activation/désactivation d'OpenModelica 183
Figure 7.3 Algorithme du modèle RealTime 187
Figure 7.4 La fonction externe initialisationTime() 188
Figure 7.5 La fonction externe getTime() 188
Figure 7.6 La fonction externe sleepMicro() 189
Figure 7.7 Modèle RealTime sous Dymola 190
Figure 8.1 Système de guidage 192
Figure 8.2 Automates hybrides de la partie commande et de la partie opérative 194
Figure 8.3 Résultats de la simulation sous OpenModelica 195
Figure 8.4 Différence entre le temps simulé et le temps d'horloge 196
Figure 8.5 Capture d'écran de l'animation du système 197

Glossaire

ABE (A Broadcast Emulator)

ACM (ALSP Common Module)

ACT (ALSP Control Terminal)

ALSP (Aggregate Level Simulation Protocol)
ALSP est un protocole de simulation de niveau agrégé, décidé par la DARPA en 1990, dont l'objectif est de permettre l'interconnexion plusieurs simulations constructives non temps réel déjà existantes.
Cf. page 80.

API (Application Program Interface)
Une API est une interface de programmation est un ensemble de fonctions, procédures ou classes mises à disposition des programmes informatiques par une bibliothèque logicielle, un système d'exploitation ou un service.

CGF (Computer Generated Forces)
CGF est un terme générique utilisé pour faire référence à des forces dans une simulation qui essayent de modéliser un comportement humain.

DARPA (Defense Advanced Research Projects Agency)
La DARPA est la fameuse agence créée le 10 Avril 1957 aux Etats-Units par le Président Eisenhower. Elle est particulièrement concernée par les recherches en informatique.

DIS (Distributed Interactive Simulation)
DIS (Distributed Interactive Simulation) est une architecture pour la simulation distribuée. Elle permet de relier des simulations afin de créer un monde virtuel.
Cf. page 71.

DMSO (Defense Modeling and Simulation Office)

DoD (Department of Defense)

EV (Environnement Virtuel)

FEDEP (Federation Development and Execution Process)
Outil méthodologique de conception générique d'une fédération HLA.
Cf. page 140

FOM (Federation Object Model)
Le FOM est un modèle objet de la fédération. Il doit décrire toutes les données échangées au cours de l'exécution d'une fédération, et doit indiquer les conditions de ces échanges. Cf. page 84.

HIL (Hardware In the Loop)

HLA (High Level Architecture)
HLA est une spécification d'architecture logicielle qui définit comment créer une simulation globale composée de simulations distribuées interagissant sans être recodées.
Cf. page 81.

OMG (Object Management Group)

OMT (Object Model Template)
L'OMT est un format de représentation commun pour les modèles objets de HLA. Il permet une identification de l'ensemble des objets représentant le «monde réel» (attributs, hiérarchie, interactions, …).
Cf. pages 85-86.

PDU (Protocol Data Unit)
Les PDUs sont des messages formatés, suivant une référence à l'ISO. Les PDUs définissent l'information " échangée " entres les sites de simulation.
Cf. page 75.

PVM (Parallel Virtual Machine)

RO (Receive order)
Cf. page 114

RTI (Run-Time Infrastructure)
Le RTI (Run-Time Infrastructure) constitue une implémentation informatique des spécifications d'interface HLA. Il s'agit d'un processus informatique assurant les communications entre les fédérés d'une même fédération, en offrant les services de HLA au travers d'une API.
Cf. page 83.

SOM (Simulation Object Model)
Le SOM est un modèle objet de la simulation. Il représente la spécification des capacités offertes aux fédérations par chaque simulation individuelle.
Cf. pages 84-85.

SIMNET (SIMulation NETworking)
C'est un environnement virtuel réparti développé pour DARPA par BBN (Bolt, Beranek and Newman), Perceptronics et Delta Graphics.

TSO (Time Stamp Order)
Cf. page 114

1. Chapitre 1 : Problématique

La mécatronique est l'intégration de différentes techniques de la mécanique, de l'automatisme, de l'électronique et de l'informatique. Cette synergie multidisciplinaire de technologies qui forme la mécatronique est adaptée par beaucoup d'entreprises pour concevoir des systèmes embarqués et plus généralement des produits plus performants. Les produits nécessitent l'utilisation de plusieurs sous systèmes de natures différentes, connectés ensemble, pour satisfaire des fonctionnalités spécifiques. Les systèmes mécatroniques complexes entrent dans une grande variété de produits industriels : des équipements industriels jusqu'aux équipements de la maison. Les premières applications ont concerné l'aéronautique, le spatial, le nucléaire et l'armement. Les produits mécatroniques naissent de l'analyse systémique d'un besoin, proviennent de la fusion et non de la simple juxtaposition des technologies et sont donc complexes. Cette complexité consiste à l'intégration de plusieurs disciplines. En conséquence, le comportement émergeant de ces systèmes à technologies différentes est difficile à modéliser.

Comme ces systèmes mécatroniques complexes sont intrinsèquement de nature pluridisciplinaire, l'approche à suivre pour la conduite du projet est de type «concurrent engineering»: un système mécatronique complexe doit être vu comme un système pluridisciplinaire dans lequel tous les aspects doivent être optimisés simultanément.

D'autre part, l'évolution rapide des marchés concurrents exige la diminution du temps de développement d'un produit en gardant la qualité et la performance du système. Il est donc nécessaire d'augmenter l'efficacité du processus de conception. Pour répondre à cette situation, en complément des outils d'analyse, la simulation, et spécialement le prototypage virtuel, est devenu l'une des clés technologiques utilisées pour augmenter cette efficacité.

La simulation d'applications en réalité virtuelle consiste en l'association de l'animation 3D temps réel des objets virtuels et de l'immersion de l'utilisateur. Si on applique ceci au domaine des systèmes mécatroniques, cela revient à dire que la simulation d'un système mécatronique consiste à l'animation en temps réel de la partie opérative du système relativement à l'ensemble des évènements de la partie commande et aux interactions de l'utilisateur. L'utilisation du prototypage virtuel peut être intégré dans 3 étapes du cycle de vie d'un système mécatronique. D'abord, il est possible d'utiliser le prototype virtuel comme un modèle géométrique pour aider les concepteurs dans la tâche de conception. Ensuite, le prototype virtuel peut être utilisé à la fin de la phase de conception pour vérifier, par différents scénarios, s'il y a des erreurs de conception. Finalement, il peut être utilisé dans la phase de conception lorsque deux solutions de conception sont envisageables, comme aide à la décision. En contre partie du réalisme apporté, les logiciels de prototypage virtuel doivent intégrer des contraintes fortes de temps réel s'il y a interaction.

Pour l'analyse du comportement, nous avons pu établir que ces systèmes entrent dans la catégorie des systèmes dynamiques hybrides où interagissent modèles à évènements discrets et modèles à équations différentielles à temps continu. En effet, le fonctionnement des composants des systèmes mécatroniques (moteur, vérin, …) est un fonctionnement continu. Néanmoins la commande de ces composants est en générale discrète. Il est donc nécessaire d'utiliser un formalisme de modélisation des systèmes dynamiques hybride pour décrire le comportement du prototype virtuel.

Le travail comporte en premier lieu un état de l'art sur les systèmes mécatroniques et la modélisation de leur comportement hybride. Nous présentons différents formalismes de description des systèmes dynamiques hybrides. Un choix de formalisme d'analyse est effectué aussi après l'étude de différents formalismes issus de la littérature des systèmes dynamiques hybrides. Par la suite, nous présentons les différents environnements virtuels et les outils de modélisation des systèmes complexes, en particulier du comportement des systèmes mécatroniques. Un choix d'un environnement virtuel est alors effectué après une étude comparative des

environnements de réalité virtuelle ainsi qu'un environnement de simulation des systèmes dynamiques hybrides.

Pour la simulation, l'analyse individuelle de chaque sous système appartenant à un système mécatronique complexe est normalement fait séparément à l'aide des outils de simulation spécifiques et qui ont atteint un grand degré de spécialisation et de performance. Par contre, l'analyse du système complet est plus difficile et il n'est pas facile de trouver des outils de simulation capables d'analyser des systèmes pluridisciplinaires dépendants de différents domaines physiques. Un environnement qui permet une simulation intégrée multidisciplinaire des systèmes mécatroniques complexes est nécessaire pour une évaluation fonctionnelle plus précise de la conception du produit et pour améliorer la qualité et l'efficacité de cette conception. C'est pour cela qu'une structure flexible et efficace utilisant un environnement intégré est nécessaire pour examiner le comportement global de ces produits. Dans cette structure, les outils de simulation distribuée dans plusieurs disciplines peuvent communiquer simultanément et fonctionner comme un seul simulateur dédié au produit.

HLA est une architecture standard IEEE de simulation distribuée. HLA (High Level Architecture) est imposée par le DMSO (Defense Modeling and Simulation Office). Elle offre la possibilité de connecter des simulateurs hétérogènes. Nous cherchons à adopter comme structure la norme IEEE 1516 HLA pour intégrer les outils de simulation de plusieurs disciplines pour la conception des systèmes mécatroniques. Cette stratégie devra permettre de réaliser une structure hétérogène fiable dans l'environnement de simulation intégrée. Ainsi, une vérification fonctionnelle et réaliste pourra être exécutée avant le prototypage physique.

Un de nos axes de travail est de pouvoir établir une communication entre différents simulateurs, en se basant sur l'architecture HLA, et en particulier l'intégration d'un simulateur 3D temps réel tel que OpenMASK et d'un simulateur des systèmes physiques tel que OpenModelica.

Quand les modèles de comportement et CAO s'avèrent fin prêts, intervient l'intégration des services de gestion du temps, point fort de l'architecture HLA. La synchronisation des deux simulateurs afin de fonctionner comme un seul simulateur est une tâche indispensable. Nous nous basons sur les services HLA de gestion du temps pour établir cette synchronisation. Le prototype virtuel décrit par la simulation distribuée doit intégrer des contraintes fortes temps réel en contre partie du réalisme apporté. Un de nos axes de recherche est de pouvoir établir une méthodologie de gestion du temps pour la synchronisation des simulateurs ainsi que des mécanismes pour aboutir à une simulation en temps réel pour apporter du réalisme au prototype virtuel. Avec ce réalisme apporté, les concepteurs peuvent évaluer leur prototype en temps réel à travers des interactions. En effet, un utilisateur ne peut interagir avec un prototype virtuel sauf ci l'animation 3D de ce dernier s'effectue en temps réel.

Afin de valider les apports pour la conception mécatronique de cette simulation distribuée, nous nous basons, dans un premier temps, sur une plateforme expérimentale constituée d'un simulateur 3D interactif OpenMASK pour le prototypage virtuel, en communication avec le simulateur OpenModelica qui permet quand à lui de modéliser et simuler le comportement de la partie opérative et la partie commande d'un système mécatronique en s'appuyant sur une approche Orientée Objet. C'est sur cette base que nous avons effectué l'expérimentation.

Le mémoire que nous présentons ici s'organise de la manière suivante :

- Au début nous dressons un état de l'art de la modélisation et la simulation des systèmes mécatroniques. Nous explorons les différents formalismes de modélisation de tels systèmes que nous considérons comme des systèmes dynamiques hybrides. Nous présentons les principaux outils et langages de simulation de ces systèmes.
- Ensuite nous explorons en détail ces langages de modélisation, en particulier le langage Modelica.

- Ensuite nous dressons un état de l’art de la simulation en particulier la simulation distribuée à évènements discrets. Nous présentons les architectures de simulation les plus connues. Notre choix s’est porté sur l’architecture HLA que nous présentons en détail.
- Nous effectuons par la suite un choix de distribution des modèles de comportement sous OpenModelica. Ensuite, nous proposons une méthode générale d’implémentation du formalisme des automates hybrides avec le langage Modelica.
- Ensuite nous présentons nos méthodes pour rendre les simulateurs OpenModelica et OpenMASK compatibles HLA. Nous nous focalisons sur la synchronisation entre les différents simulateur et nous proposons une stratégie pour gérer le temps afin d’obtenir une simulation temps réel.
- Enfin nous exposerons l’implémentation de ces différentes méthodes ainsi qu’une application sur un système mécatronique.

2. Chapitre 2 : Modélisation et simulation des systèmes mécatroniques

2.1 Introduction

Dans notre approche d'un système mécatronique nous avons gardé l'hypothèse qu'il est formé d'une partie commande et d'une partie opérative munie d'une géométrie. Dans un premier lieu, nous présentons différentes définitions d'un système mécatronique et nous présentons notre approche de modélisation basée sur la distinction entre la partie opérative et la partie commande.

Le comportement des systèmes mécatroniques est celui décrit par les systèmes dynamiques hybrides. Pour cela, nous présentons les formalismes les plus pertinents de modélisation des systèmes dynamiques hybrides. Nous faisons un choix parmi ces formalismes. Par la suite, nous présentons les outils et langage de simulation des sytèmes dynamiques hybrides. Un choix d'un langage de modélisation et d'un outil de simulation est effectué.

Pour l'animation 3D du prototype virtuel d'un système mécatronique, nous nous sommes intéressés à un ensemble d'environnements virtuels. Nous décrivons ces environnements et nous effectuons un choix.

2.1 Les systèmes mécatroniques

La mécatronique se définit comme la combinaison synergique et systémique de la mécanique, de l'électronique et de l'informatique. L'intérêt de ce domaine d'ingénierie multidisciplinaire est de concevoir des systèmes complexes et de permettre leur contrôle. Le terme *mechatronics* a été introduit par un ingénieur de la compagnie japonaise « Yaskawa » en 1969. Le terme est apparu officiellement en France dans le Larousse 2005.

Plusieurs définitions sont données pour définir les systèmes mécatroniques dont :

« La mécatronique est l'intégration synergique de l'ingénierie mécanique avec l'électronique et le contrôle intelligent de calculateurs dans la conception et la fabrication de produits et processus industriels. » **[Harashima, 1996]**

Isermann **[Isermann, 2000]** résume les définitions données à la mécatronique. Il estime que toutes les définitions sont d'accord pour dire que la mécatronique est un domaine interdisciplinaire dans lequel les disciplines suivantes agissent ensemble :

- Systèmes mécaniques (éléments mécaniques, machines, mécanique de précision).
- Systèmes électroniques (micro-électronique, électronique de puissance, capteurs et actionneurs).
- Technologie de l'information (théorie des systèmes, automatisation, génie logiciel, intelligence artificielle).

Nous définissons un système mécatronique de la façon suivante :

Un système mécatronique est un système complexe pluridisciplinaire à dominante mécanique et électronique avec contraintes temps réel.

- Complexe : un système mécatronique est composé d'un grand nombre d'entités en interaction locale et simultanée où il y a des boucles de rétroaction (feedback) : l'état d'une entité a une influence sur son état futur via l'état d'autres entités. De plus, un système mécatronique est un système ouvert et soumis à un extérieur, il y a des flux d'énergie et d'information sur la frontière.
- Pluridisciplinaire : Plusieurs domaines technologiques sont mis en œuvre pour les parties commande et opérative. Un système mécatronique est composé d'éléments de différents domaines : des composants mécaniques, électroniques et des technologies de l'information.

- Avec contraintes temps réel : un système mécatronique est le plus souvent immergé dans son environnement et doit permettre une automatisation d'un ensemble de tâches. Le respect des contraintes temporelles dans l'exécution des tâches est aussi important que le résultat de ces tâches pour permettre aux clients de ces derniers de les exploiter correctement.

L'appellation « mécatronique » est un classement selon la technologie de réalisation. Du point de vue fonctionnel, un système mécatronique est un système pouvant être vu comme une collaboration de deux grandes parties :

- Une partie *Opérative* : Elle est constituée de parties mécaniques et électromécaniques.
- Une partie *Commande* : Elle est constituée à partir des technologies électroniques, informatiques et automatiques. Elle peut être un système de contrôle en boucle ouverte ou fermée.

La distinction des parties commande et opérative n'est pas la disjonction du système mécatronique en deux sous systèmes interagissant. Ainsi, le fait de distinguer une partie opérative d'une partie commande n'implique pas de les disjoindre et de les penser séparément mais de les comprendre et de comprendre les liens qui existent entre elles. L'approche de conception d'un système mécatronique, aussi appelée conception mécatronique, veut que l'on entreprenne une approche de conception simultanée de ces deux parties. Cette approche est exceptionnelle. En effet, elle ne découpe pas, ne trie pas pour représenter mais elle admet d'entrer dans la globalité pour la voir et la décoder. C'est une approche qui articule ce qui est séparé et relie ce qui est disjoint. Dans l'article « Pour une réforme de la pensée », Edgar Morin[1] décrit globalement la modélisation systémique complexe en disant : « La pensée complexe est une pensée qui cherche à la fois à distinguer - mais sans disjoindre - et à relier. D'autre part, il faut traiter l'incertitude. Le dogme d'un déterminisme universel s'est effondré. L'univers n'est pas soumis à la souveraineté absolue de l'ordre, il est le jeu et

[1] Philosophe et anthropo-sociologue, directeur de recherche émérite au CNRS

l'enjeu d'une dialogique (relation à la fois antagoniste, concurrente et complémentaire) entre l'ordre, le désordre et l'organisation ». Ainsi, la distinction des parties commande et opérative d'un système mécatronique doit permettre d'analyser chaque partie en dépend de l'autre et d'étudier les liens et échanges qui existent entre elles. Il s'agit d'appréhender les comportements de chaque partie, les aléas aussi et de ne pas leur opposer une résistance ou les disconvenir, mais de les intégrer pour voir meilleur et davantage. Pascal évoquait dans ses « pensées » en 1657 que pour connaître l'un il faut connaître l'autre, et que : " toutes choses étant causées et causantes, aidées et aidantes, médiates et immédiates, et toutes s'entretenant par un lien naturel et insensible qui lie les plus éloignées et les plus différentes, je tiens impossible de connaître les parties sans connaître le tout, non plus que de connaître le tout sans connaître particulièrement les parties". La conception mécatronique entreprend cette vision systémique de la conception. Plus près de l'ingénieur, il existe maintenant une norme d'ingénierie système IEEE 15288 qui concrétise cette vision systémique. **[IEEE15288]**.

2.3 Modélisation du comportement: les systèmes dynamiques hybrides

Bien que les commandes d'un système mécatronique soient souvent des commandes discrètes, le comportement global est quant à lui constitué de sauts entre des états continus. Ce type de comportement est celui des systèmes dynamiques hybrides. Il existe un grand nombre de formalismes qui permettent la modélisation des systèmes dynamiques hybrides. En effet, l'intérêt pour ces systèmes a connu un regain ces dernières années à la fois dans la communauté automaticienne et dans la communauté informaticienne. Parmi ces formalismes on peut citer les automates hybrides, les Bond Graph à commutations, les réseaux de Petri hybrides, les Grafcet hybrides...etc. Dans l'étude de l'art suivante nous faisons l'analyse des formalismes qui nous semblent les plus proches de notre application.

Les systèmes hybrides se retrouvent dans beaucoup de disciplines, comme par exemple l'automobile, l'avionique, la robotique et la génétique, pour en citer quelques-unes. L'étude des SDH s'intéresse principalement aux classes de problèmes qui n'ont pu être traités avec les approches traditionnelles basées sur une modélisation

homogène. Il n'existe pas pour l'instant de théorie globale pour l'étude de ces systèmes, mais plutôt des approches basées sur l'extension de méthodes classiques issues des systèmes continus ou discrets, en vue de couvrir une gamme plus étendue d'applications.

Une première définition :

Un système dynamique hybride est un système contenant des variables d'état continues/discrètes et des variables d'état booléennes en interaction.

Une définition formelle d'un SDH

Soit x(t) □ X □ □^{n}, q(t) □ Q □ □^{m}, u(t) □ U □ □^{c}, v(t) □ Ω_c □ □^{d}, et t □ □$^+$ où n, m, c, d sont donnés.

L'ensemble X représente l'ensemble des états continus et l'ensemble Q représente l'ensemble des états discrets. L'ensemble U représente l'ensemble des commandes continues et l'ensemble Ω_c représente l'ensemble des commandes discrètes. La variable t représente le temps.

Soit $S = X \times Q$ et soit P un sous ensemble fermé de □^{n} × Q. On définit l'état hybride s(t) □ S du système à l'instant t par la donnée du couple [$x(t),q(t)$] et on appelle SDH un système dynamique décrit par les équations suivantes pout t □ [t_0,t_f] :

$$[x(t^+), q(t^+)] = G(x(t), q(t), v(t)) \qquad \square\ s(t) \square S \square \square P \tag{1}$$

$$[x(t^+), q(t^+)] = G(x(t), q(t), v(t)) \quad \begin{cases} \text{si } s(t) \square \partial P \\ \text{si } v(t) \square \Omega_c \text{ intervient} \end{cases} \tag{2}$$

$x(t_0)=x_0$ (3)

$q(t_0)=q_0$ (4)

La fonction F représente la dynamique continue du système hybride et G représente la dynamique hybride du système correspondant aux phéenomènes hybrides décrits

précédemment. L'ensemble *P* est celui des zones de déclenchement des phénomènes hybrides et *∂P* est la frontière de *P*. La commande discrète *v(t)* détermine les instants où la fonction *G* intervient (instants de commutation/saut). Elle d´etermine également le nouveau modèle ou l'amplitude du saut del'état suivant l'action du phénomène hybride. Les fonctions *F* et *G* sont supposées suffisamment régulière pour que le système défini par les équations (1-4) admette une solution unique **[Cébron, 2000]**.

2.3.1 Les automates hybrides

Pour pouvoir modéliser une plus grande variété de dynamiques continues, le formalisme des automates temporisés a été étendu pour obtenir les automates hybrides.

Définition d'un automate hybride

Un automate hybride **[Henzinger, 1995]** est défini par la donnée de (Q, X, Σ, A, Inv, F, q_0, x_0) où :

- Q, est un ensemble fini de sommets, appelés situations, et q_0 la situation initiale ;
- X, est l'espace d'état continu de l'automate, $X \subset \mathbb{R}^n$; x_0 est la valeur initiale de l'état continu ;
- Σ, est un ensemble fini d'évènements ;
- A, est un ensemble de transitions définies par un quintuple (q,Guard,σ,Jump,q') et représentées par un arc entre les situations, où :
    - $q \in Q, q' \in Q$,
    - Guard, est un sous ensemble de l'espace d'état dans lequel doit se trouver l'état continu pour que la transition puisse être franchie,
    - Jump, représente la transformation de l'état continu lors du changement de situation ; elle est généralement exprimée sous la forme d'une fonction de la valeur de l'état, avant commutation, dont le résultat est affecté comme valeur initiale de l'état continu dans la

nouvelle situation ;

- $\sigma \in \Sigma$, est l'événement associé à la transition, cette association n'implique pas de donner un sens en terme d'entrée ou de sortie de l'événement ;

- Inv, est une application qui associe à chaque situation un sous-ensemble de l'espace d'état, appelé *invariant* de la situation, dans lequel l'état continu doit rester, lorsque la situation est q, l'état continu doit vérifier

- F, définit pour chaque situation, l'évolution de l'état continu lorsque la situation est active ; cette évolution de l'état continu est le plus souvent exprimée par une équation différentielle ; elle peut dans certains cas être définie sous une forme moins explicite ou impérative, telle que l'inclusion différentielle qui définit, pour chaque variable, un intervalle dans lequel sa dérivée peut évoluer. On la nomme *flow* **[Henzinger, 1995]**.

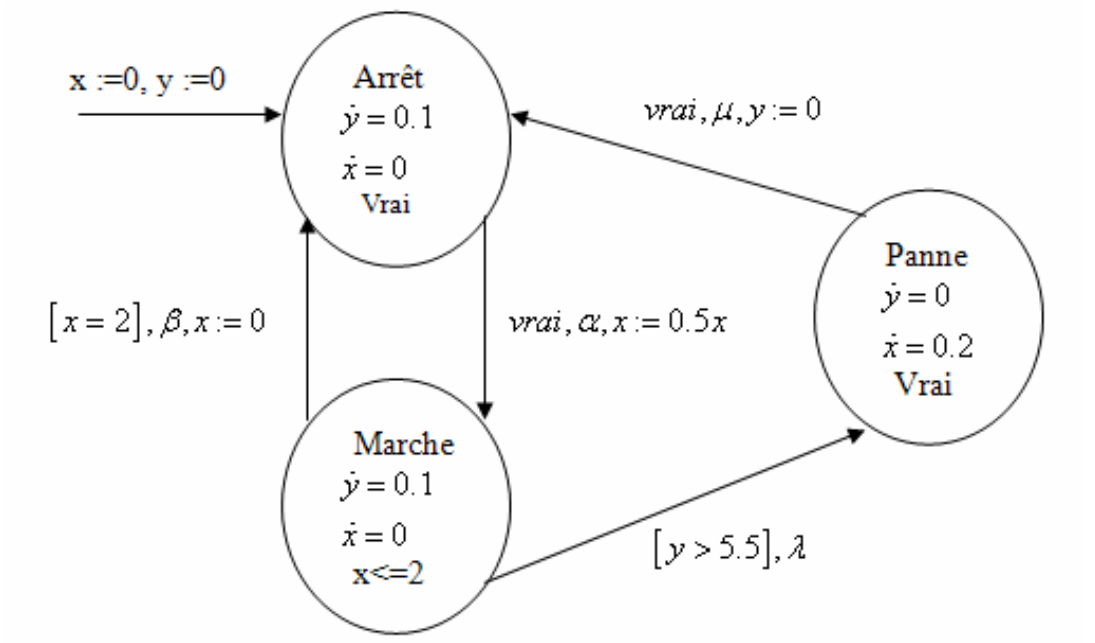


Figure 2.1 Exemple d'automate hybride modélisant une machine simple

Evolution d'un automate hybride

La sémantique des automates hybrides est définie en considérant qu'à chaque instant, l'état d'un automate hybride est donné par la paire (q, x) correspondant à l'association d'une situation et d'une valeur du vecteur d'état, et que cet état peut évoluer :

- Soit par une transition instantanée qui change la situation et la valeur de l'état continu (fonction de saut),
- Soit par la progression du temps dans la situation courante, ce qui entraîne un changement de l'état continu conformément à l'activité, F, de la situation.

Il est ainsi possible de franchir une transition si l'état continu est dans le sous-espace défini par son *Guard*, et est tel que sa transformation par le saut satisfait l'*Invariant* de la situation d'arrivée. Cependant, rien n'impose ce franchissement tant que l'*Invariant* de la situation de départ est satisfait.

On peut utiliser l'automate hybride pour modéliser le comportement d'un SDH complexe. Mais, si l'on veut utiliser ce formalisme pour réaliser un SDH de commande industrielle, alors il faut ajouter des commandes telles que les variables continues et les évènements typés Entrée/Sortie. **[Hien, 2001]**

Couplage des automates hybrides

Il est possible de coupler des automates hybrides en se basant sur le mécanisme de composition synchrone. Ce couplage a pour but de séparer la partie contrôle de la partie opérative. Une première définition est la suivante: « Si l'on considère deux automates et un événement qui appartient à l'ensemble des évènements de chacun d'eux, une transition étiquetée par cet événement ne peut être franchie dans l'un des automates que s'il existe une transition étiquetée, par ce même événement, franchissable dans l'autre automate. Les deux transitions sont alors franchies simultanément. » **[Zaytoon, 2001]**

Synthèse

Le formalisme des automates hybrides est issu d'un formalisme de description de systèmes à événements discrets. Les situations de l'automate hybride comportent les équations différentielles des états continus. Il est donc très simple d'extraire les équations d'états du système. D'un autre coté, ce formalisme permet de concevoir des commandes grâce au couplage entre les automates hybrides. Un autre avantage est la disponibilité d'outil de vérification formelle pour ce formalisme. On peut citer HyTech. **[Henzinger, 1997]**

2.3.2 Les réseaux de Petri

Un réseau de Petri **[Brams, 1983],** est un graphe formé de deux types de nœuds : les places et les transitions, reliées par des arcs orientés. L'état d'un RdP est représenté par son marquage qui est le vecteur courant du nombre de marques dans l'ensemble de ses places. Les réseaux de Petri (RdP) sont utilisés à la base pour modéliser les systèmes à événements discrets. Ils ont été ensuite étendus pour modéliser le comportement continu d'un système dynamique hybride. Nous nous limitons aux réseaux de Petri qui permettent de modéliser les systèmes dynamiques hybrides.

2.3.2.1 Les réseaux de Petri hybrides

Un réseau de Petri hybride (HPN) peut être vu comme composé d'un RdP classique et un RdP continu.

Définition 1

Un réseau de Petri hybride est le 6-uplet : **HPN= (P, T, Pré, Post, h, Mo)** tel que **[Zaytoon, 2001]** :

- P= (P1, P2,…, Pn) est un ensemble fini et non vide de places ;
- T= (T1, T2,…, Tn) est un ensemble fini et non vide de transitions ;
- Pré : PxT→ N est l'application d'incidence avant où N représente l'ensemble des entiers non négatifs ;

- Post : PxT→ N est l'application d'incidence arrière ;
- h : P U T → {D, C} appelée « fonction hybride » qui indique si un nœud est discret ou continue.
- Mo est le marquage initial ;
- Si $h(p_k) = D$ et si $h(t_i) = C$ alors $pre(p_k, t_i) = post(p_k, t_i)$

D'un point de vue graphique, les places discrètes sont représentées par des cercles et les transitions discrètes par des barres. Les places continues sont représentées par des cercles doubles et les transitions continues par des boites pleines. **[Alla, 2001]**

Définition 2

Le marquage d'un RdP hybride est la fonction M (Pj) qui assigne un nombre entier non négatif à chaque place discrète et un nombre réel à chaque place continue. Le marquage est une représentation instantanée de l'état du système. **[Febbraro, 2003]**

$$\dot{M}(p_i, \tau) = \Sigma_{j \in IN} Post(p_i, t_j) \square v_j(\tau) - \Sigma_{l \in OUT} Pre(p_i, t_l) \square v_l(\tau)$$

L'exemple de la figure 2.2 **[Andreu, 1996]** correspond à une chaîne où le remplissage de bouteilles (transition t1) est sous le contrôle d'une vanne deux états : {ouverte ; fermée}. Quand la quantité *x* est atteinte la vanne se ferme (p3 démarquée et donc p4 marquée). Ceci illustre le contrôle d'un processus continu par un événement discret. D'autre part, les RdP hybrides permettent de représenter la transformation en lots. Ainsi, dans l'exemple, 10 litres donnent 20 bouteilles de 50 cl car, lorsque la valeur seuil 10 de l'arc entrant de t4 est atteinte, 20 jetons discrets sont déposés dans p5.

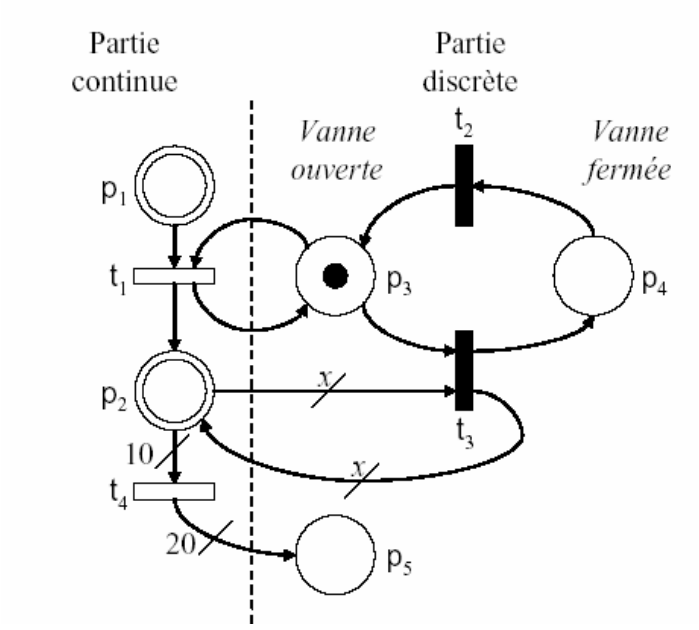


Figure 2.2: Exemple de RdP hybride

Les RdPs hybrides ont été utilisés pour modéliser un système d'approvisionnement en eau, un système urbain de réseaux (les routes et les ronds-points) **[Febbraro, 2003]** ou un réseau de communication basé sur le protocole TCP/IP **[Bitam, 2003]**.

2.3.2.2 Les réseaux de Petri mixtes

Les réseaux de Petri mixtes sont une extension des RdP interprétés dans lesquels des équations peuvent être associées aux places et des fonctions de saut aux transitions. Les réseaux de Petri mixtes encapsulent la puissance de structuration des réseaux de Petri et la puissance de description continue des équations algébro-différentielles **[Valentin, 1999]**. En effet, les réseaux de Petri mixtes (RdPM) consistent à donner une interprétation aux réseaux de Petri par un ensemble d'équations algébro-différentielles.

L'objectif des réseaux de Petri mixtes est de représenter graphiquement le parallélisme d'activités, leurs synchronisations et le partage de ressources tout en

dupliquant le moins possible les équations décrivant les évolutions dynamiques continues **[Valentin, 2005]**.

Un réseau de Petri initialement marqué est donc un quintuple **{R, PC, IH, Mo, xo}** où :

- **R** est un ensemble de réseaux de Petri interprétés qui peuvent se synchroniser par des évènements internes ; comme dans les statecharts chaque transition pourra être associée à un couple d'évènements, l'un déclenchant le franchissement et l'autre étant généré par le franchissement ;
- **PC** est la partie continue du modèle ; elle comprend l'ensemble des variables continues et l'ensemble des équations algébro-différentielles qui modélisent les différentes dynamiques continues du système ; les variables sont globales et leur valeur est accessible partout dans le modèle à tout instant ;
- **IH** est l'interface hybride entre la partie continue du modèle et les réseaux de Petri ;
- $\mathbf{M_o}$ est le marquage initial de R ;
- $\mathbf{X_o}$ est l'état initial des variables continues.

2.3.2.3 Les réseaux de Petri Prédicats -Transitions-Différentiels-Objets : (RdP-PTD)

Le formalisme RdP-PTD Objets est une extension du formalisme RdP prédicats-transitions différentiels (RDP-PTD) développée au LAAS à Toulouse **[Champagnat, 1998]**. Ce modèle se caractérise essentiellement de deux aspects : **[Perret, 2003]**

- D'une part, il combine un réseau prédicats-transitions et des systèmes d'équations différentielles algébriques. A chaque place du réseau, est associé un système d'équations. Celui-ci lie les variables continues propres à la place et les variables transportées par les jetons présents.

- D’autre part, il intègre les concepts objets qui permettent de décrire les aspects *statiques* et *structurels* du système. Le jeton n’est plus seulement une structure regroupant des variables continues mais un objet en capsulant à la fois des attributs (dont les variables continues) et un ensemble de méthodes qui traitent ces données (dont des équations qui peuvent être ajoutées au modèle déjà présent dans la place).

Un même principe de base est partagé par les RdP mixtes et les RdP-PTD. En effet, l’évolution des variables réelles est définie par un ensemble d’équations différentielles et algébriques. Les équations actives sont définies à partir du marquage du réseau de Petri. Quant au franchissement des transitions, dépend de l’apparition de seuils sur les variables réelles ou leurs dérivées.

Synthèse

Le formalisme des réseaux de Petri hybrides est issu d’un formalisme de description de système à événements discrets. En effet, la partie discrète est très bien structurée de manière graphique. Par contre, Il est compliqué de déduire les équations d’états à partir du graphe. En effet, le système d’équations algébro-différentielles associé à chaque marquage du modèle est un assemblage d’équations associées à des places du réseau de Petri donc sa structure n’apparaît pas directement.

2.3.3 Grafcet hybride

Un GRAFCET est défini par un ensemble constitué :

- D’éléments graphiques de base : les étapes, les transitions, les liaisons orientées. Une étape est soit active, soit inactive. L’ensemble des étapes actives définit la situation de la partie commande.
- D’une interprétation associant des expressions logiques : les actions associées aux étapes, les réceptivités associées aux transitions.
- De règles d’évolution définissant formellement le comportement dynamique de la partie commande.

Les possibilités offertes pour exprimer le parallélisme et la synchronisation ainsi que le concept de réceptivité en font, en effet, un outil bien adapté pour décrire simplement la commande des systèmes logiques fortement parallèles et/ou synchronisés. **[Zaytoon, 2001]**

Le Grafcet permet de visualiser de façon particulièrement claire toutes les évolutions du système. Un Grafcet hybride est une extension de Grafcet dans laquelle les équations peuvent être associées à des étapes, et les actions instantanées peuvent être associées à des transitions. **[Gueguen, 2001]**

2.3.4 Autres approches

- Les Statecharts hybrides

L'utilisation des statecharts pour décrire le comportement dynamique des systèmes complexes augmente d'une année à une autre. Les statecharts ont été définis comme une solution pour améliorer la structuration des spécifications à base d'automates afin de faciliter leur description, leur lecture et leur modification. Un statecharts hybride est un statecharts auquel est associé un ensemble de variables d'états continues et pour lequel un système d'équations algébro-différentielles portant sur un sous ensemble de ces variables peut être associé à chaque état.

- Bond graphs à commutation

Le bond graph est un outil classique de modélisation de systèmes physiques connus. C'est une approche graphique indépendante du domaine d'application et qui permet de décrire le modèle en se basant sur des considérations énergétiques. L'intérêt de cette approche est l'interaction constante entre la physique du système, le modèle bond graph et ses propriétés structurelles.Les bonds graphs sont de plus en plus utilisés pour la modélisation de systèmes complexes ou multi domaines **[Zaytoon, 2001]**. Parmi les applications remarquables, on peut noter l'utilisation de modèles complets de chaînes de motorisation de véhicules automobiles.

- Formulation Hamiltonienne

C’est une approche générique et systémique issue de la théorie des graphes pour la modélisation des systèmes dynamiques physiques. Elle est basée sur le concept d’énergie et les graphes d’interconnexion des ports. Cette méthode permet d’obtenir un modèle structuré de toutes les configurations du système dynamique hybride sous une seule représentation paramétrée par l’état discret des commutateurs. **[Valentin, 2006] [Valentin, 2007]**

2.3.5 Choix du formalisme

A travers l'étude des différents formalismes de description des systèmes dynamiques hybrides, nous avons choisi d’utiliser le formalisme des automates hybrides qui convient le mieux à notre application. En effet, parmi les formalismes qui combinent un sous-modèle pour les dynamiques continues et un sous-modèle pour les dynamiques discrètes avec des interfaces entre les deux sous-modèles et qui prend en compte l’aspect hybride, le modèle formel le plus élémentaire est celui des automates hybrides. En effet, à chaque instant un seul état discret est actif et un ensemble d’équations différentielles lui sont associées. Le formalisme des automates hybrides permet facilement d’extraire les équations d’état du système. Par contre dans les modèles basés sur les RdP, le système d’équations algébro-différentiellesassocié à chaque marquage du modèle est un assemblage d’équations associées à des places du RdP et donc sa partie continue n’est pas bien structurée. La partie discrète est très bien structurée dans les RdP et les Graphcets à cause de l’aspect graphique alors que dans les automates hybrides le parallélisme et la synchronisation ne sont pas visibles directement. Malgré cet inconvénient, le formalisme des automates hybrides permet de modéliser le couplage entre les parties commande et opérative modélisées par des automates hybrides distincts.

2.4 Environnements de simulation des systèmes dynamiques hybrides

2.4.1 Introduction et vue d'ensemble des outils et langages de modélisation pour la simulation des systèmes hybrides

La simulation des systèmes dynamiques hybrides combine la simulation à temps continu et la simulation à événements discrets. Malheureusement, il n'est pas suffisant de juxtaposer deux simulateurs de ces deux types pour simuler un système dynamique hybride. En effet, d'autres moyens sont essentiels pour traiter les phénomènes hybrides ainsi que le comportement continu et discret.

Il existe des techniques de simulation des systèmes dynamiques hybrides. Les principales techniques **[van der Schaft and Schumacher, 2000]** sont les suivantes.

- Méthode de lissage (*smoothing method*) : Cette méthode consiste à transformer le modèle hybride en modèle lissé en effectuant des approximations sur les aspects hybrides du modèle.
- Méthode de détection des évènements : Cette méthode est la plus pertinente pour simuler des systèmes hybrides. Tout d'abord, les conditions initiales sont établies. Ensuite, le comportement continu est simulé jusqu'à la détection d'un évènement. Quand un évènement se produit, le temps de l'évènement et les variables d'états correspondants sont calculés. Ensuite, un nouveau mode continu est déterminé et le cycle est répété.
- Méthode à pas de temps : Dans cette approche les évènements ne sont pas détectés. Un pas de discrétisation est choisi et le système hybride est approximé par le système discrétisé.

La méthode de détection des évènements est le standard de la simulation des systèmes dynamiques hybrides de facto. Mais plusieurs variantes de cette méthode existent. Andersson dans **[Andersson, 1994]** a étudié l'application de cette technique à la simulation orientée objet des systèmes dynamiques hybrides. L'application de la méthode de détection des évènements dans le simulateur 20-sim est présentée dans **[Broenink, 1996]**. Dans **[Mosterman, 2002]**, on trouve une discussion sur

l'implémentation de la méthode de détection des évènements pour la simulation des systèmes dynamiques physiques.

Il existe de nombreux outils et langages de modélisation pour la simulation et l'analyse des systèmes hybrides. Un aperçu exhaustif des outils et des langages est fourni dans **[Carloni, 2004]**. Nous présentons dans ce qui suit une description concise des outils les plus utilisés et connus.

20-sim: 20-Sim est un programme de modélisation et de simulation qui fonctionne sous Microsoft Windows. Avec 20-sim, il est possible de simuler le comportement de systèmes dynamiques, tel que des systèmes électriques, mécaniques et hydrauliques ou n'importe quelle combinaison de ces systèmes. Il supporte une grande varieté de systèmes dynamiques : les systèmes linéaires, non-linéaires, continus, à événements discrets et hybrides. 20-sim est le premier outil commercial qui a supporté la modélisation avec les bonds graphs **[Site web 20-sim]**. La première version de 20-sim avec une librairie des bonds graphs est parue en 1995. Un effort continu pour améliorer la modélisation avec les bonds graph a permit à 20-sim d'être un standard de modélisation avec les bonds graphs. 20-sim supporte aussi la modélisation graphique.

AnyLogic [Borshchev,2000] Anylogic est un environnement de prototypage virtuel, développé par XJ Technologies Co. Ltd[2] , basé sur UML-RT, Java et les équations algébro-différentielles. C'est un outil de simulation de type discret, continu et hybride. Les résultats de la simulation doivent être évalués en prenant en compte que les sémantiques de l'outil ne sont pas formelles.

Charon [Alur, 2000] La boite à outils Charon est basé sur le langage Charon. Elle est implémentée en Java. La boite à outil inclut un éditeur avec un parseur et vérificateur de la syntaxe, un explorateur de modèles et un simulateur visuel. **[Site Web Charon]**.

[2]www.xjtek.com

HyVisual et Ptolemy [Lee, 2005] est un outil de modélisation visuel et de simulation pour les systèmes dynamiques continus et les systèmes hybrides. HyVisual est une partie du projet Ptolemy et il est basé sur Ptolemy II. Ce dernier est une boite à outils développée en Java pour la modélisation et la conception des systèmes embarqués, temps réel et concurrents. HyVisual fournit des moyens pour la modélisation et la simulation des systèmes dynamiques continus et des systèmes hybrides. Les modèles sont conçus en utilisant les diagrammes de block. Les sémantiques du langage sont présentées dans **[Lee, 2005]**.

Modelica [Fritzon, 2002] est un langage orienté objet pour la modélisation des systèmes physiques dans le but de simulation. C'est un langage multi domaine et non causal. Le nombre de librairies utilisant Modelica est en augmentation véloce. Plusieurs outils commercials ont utilisé Modelica ainsi que des outils open source. Ce langage ainsi qu'une partie de ces outils sont présentés en détail dans la suite.

Scicos [Stephen, 2006] Scicos est une boite à outils, dévéloppé à l'INRIA[3], faisant office de modeleur et de simulateur graphique. Scicos permet en utilisant les diagrammes de block de modéliser et simuler le comportement des systèmes dynamiques hybrides. De nouvelles extensions permettent de générer des composants pour la modélisation des circuits électriques et hydrauliques basés sur Modelica. Scicos est inclus dans le logiciel de calcul scientifique Scilab[4]. Scilab peut être considéré comme une version gratuite de Matlab et Scicos de Simulink.

De plus, nous présentons dans le tableau suivant une comparaison des caractéristiques principales des outils présentés précédemment. Nous précisons ensuite notre choix du langage et de l'outil à adopter.

[3] Institut National de Recherche en Informatique et en Automatique
[4] http://www.scilab.org/

Outils	Sémantiques formelles	Formalisme	Maturité
20-sim	oui	Bond Graphes	industriel
AnyLogic	non	UML-RT et DAE[5]	industriel
Charon	oui	Charon	académique
HyVisual	oui	HyVisual	académique
Scicos	non	Diagrammes de block	industriel
Dymola	partiel[6]	Modelica	industriel
OpenModelica	partiel	Modelica[7]	académique

Outils de Modélisation et de simulation des systèmes hybrides

Dans notre approche, nous avons choisi d'implémenter les automates hybrides. Si la programmation dite **procédurale** est constituée de procédures et fonctions sans liens particuliers agissant sur des données dissociées pouvant mener rapidement à des difficultés en cas de modification de la structure des données, **la programmation objet**, pour sa part, tourne autour d'une unique entité : l'objet, offrant de nouvelles perspectives. Un langage orienté objet et qui supporte la modélisation des systèmes dynamiques hybrides est utile pour implémenter les automates hybrides d'une manière modulaire. Ainsi, nous pouvons développer des composants réutilisables pour l'implémentation des automates hybrides. Le langage Modelica répond parfaitement à ce critère de choix.

Nous avons constaté aussi que la plupart des outils industriels ne sont pas basés sur des formalismes avec des sémantiques formelles au contraire des outils académiques. L'augmentation du nombre d'outils implémentant des sémantiques formelles montre l'intérêt croissant pour ces outils. Plusieurs recherches sont

[5] Equations algébro-différentielles
[6] Une partie du comportement est définie par des sémantiques formelles
[7] Nous traitons ici les spécifications formelles du langage Modelica

effectuées pour spécifier formellement des parties du langage Modelica, mais jusqu'à aujourd'hui il n'existe pas une spécification formelle complète de l'ensemble du langage **[Broman, 2007]**.

L'existence d'outils à source ouverte (Open Source) implémentant le langage Modelica nous a encouragé à choisir ce langage. Ainsi, nous pouvons proposer une approche pour l'implémentation des automates hybrides avec Modelica accessible à tous. Dans ce qui suit, nous présentons le langage Modelica, ainsi que la plupart des outils implémentant les spécifications Modelica **[Modelica Specifications, 2007]**.

2.4.2 Le language multi physique support : Modelica

Modelica est un langage orienté objet dont le but est de modéliser des systèmes physiques complexes comprenant des composants électriques, mécaniques, hydrauliques, thermiques. Le développement et la promotion de Modelica sont assurés par l'association à but non lucratif Modelica. Les quatre aspects les plus importants de Modelica sont **[Fritzon, 2003]** :

- Il est principalement basé sur des équations et non sur des instructions d'affectation. Ceci permet une modélisation qui favorise une meilleure réutilisation des classes puisque les équations ne spécifient pas de direction de flux de données.

- Il a la capacité de fournir une modélisation multi domaine. Les modèles de composants correspondant aux objets physiques de plusieurs domaines différents, électrique, mécanique, thermodynamique, hydraulique, biologique et des applications de contrôle peuvent être décrits et connectés.

- C'est un langage orienté objet avec un concept de classe générale qui englobe les classes. Ceci facilite la réutilisation des composants, l'évolution des modèles et le lie à des modèles d'analyse (UML/SysML) **[Turki, 2008]**.

- Il permet de créer des modèles de composants, avec des constructions pour connecter ces composants. Ainsi le langage est bien adapté comme langage de description architectural pour les systèmes physiques complexes, et dans une certaine mesure pour les systèmes logiciels.

Les programmes Modelica contiennent des classes, qui sont appelées aussi des modèles. A partir de la définition d'une classe, on peut créer un certain nombre d'objets qui sont des instances de cette classe. Une classe Modelica contient des éléments, des déclarations de variables et des équations, en particulier des équations différentielles. Les variables contiennent des données appartenant aux instances de la classe. Elles composent le stockage de données de l'instance. Les équations d'une classe spécifient le comportement des instances de cette classe.

2.4.2.1 Modelica : Langage orienté Objet

Sous Modelica, le paradigme orienté objet est vu comme un concept structurant utilisé pour traiter la complexité des systèmes. Un modèle Modelica est une description mathématique déclarative. Les propriétés des systèmes dynamiques sont exprimées par des équations d'une façon déclarative.

Le concept de programmation déclarative est inspiré des mathématiques. Ce paradigme de programmation traite le calcul comme l'évaluation des fonctions mathématiques, au contraire d'une programmation procédurale où le programme contient une série d'étapes ou d'instructions à réaliser montrant comment atteindre le but final. L'approche déclarative orienté objet de la modélisation des systèmes et de leurs comportements est d'un niveau d'abstraction plus haut que celui de l'approche orienté objet ordinaire puisque quelques détails d'exécution peuvent être omis. Par exemple, on n'a plus besoin de code pour transporter explicitement les données entre les objets à travers des assignations ou des opérations de passage de variables. Un tel

code est généré automatiquement par le compilateur Modelica en se basant sur les équations données.

Comme dans les langages orientés objet, les classes sont des modèles de base pour créer les objets. Les variables, les équations et les fonctions peuvent être héritées par d'autres classes.

2.4.2.2 Les classes

Comme tous les langages orienté objet, Modelica fournit la notion de classe et objet. Chaque objet dans Modelica a une classe qui définit son comportement. Une classe a trois sortes de membres :

- Les variables : Elles contiennent les valeurs de la résolution des équations d'une classe.
- Les équations : Elles déterminent le comportement d'une classe.
- Les classes : Elles peuvent êtres membres d'une autre classe.

Une classe Modelica est structurée en blocs : bloc *initial algorithm*, bloc *algorithm* et bloc *equation*. Les membres d'une classe peuvent avoir deux niveaux de visibilité : publique (*public*) ou protégé (*protected*). Par défaut, un membre est public si rien n'est indiqué avant. Un membre public est accessible de partout et sans aucune restriction. L'autre niveau de visibilité est spécifié par le mot « *protected* ». Un membre protégé est accessible uniquement par le code au sein de la classe où il est déclaré aussi bien par le code dans les classes qui héritent de cette classe.

2.4.2.3 Héritage

La modélisation orientée objet permet d'étendre le comportement et les propriétés d'une classe existante. Cette dernière est étendue pour obtenir une classe plus spécifique, appelée souvent classe fille ou sous-classe. Le comportement et les

propriétés de la classe mère, décrit sous forme de variables, d'équations et d'autres contenus, sont hérités par la classe fille.

Dans l'exemple suivant nous présentons deux classes : classe Point et classe Color. La nouvelle classe ColoredPoint hérite des deux classes leurs propriétés position et couleur.

Exemple:

```
class Point
Real x ;
Real y;
end Point;
class Color
Real red;
Real blue;
Real green;
equation
red + blue + green =1;
end Color;

class ColoredPoint
  extends Point;
  extends Color;
end ColoredPoint;
```

Figure 2.3 Exemple d'héritage entre classes de Modelica

2.4.2.4 Equations

Modelica est basé sur les équations. Elles sont plus flexibles que les structures de programmations ordinaires telles que les assignations puisqu'elles ne précisent pas la direction du flux des données. Ceci augmente une réutilisation potentielle des classes Modelica. En effet, l'utilisation des équations est plus expressive que celle des

expressions d'assignation. Prenons le cas d'une résistance. Le comportement de ce composant est représenté par l'équation suivante:

$$V = R * i$$

Cette équation a trois expressions d'assignation possibles :

$$i := V / R$$

$$V := R * i$$

$$R := V / i$$

Les équations sous Modelica peuvent êtres classées en quatre catégories[8] :

- Les équations normales.
- Les équations de déclaration.
- Les équations de modification.
- Les équations d'initialisation.

Les équations dans la partie déclaration

Les équations de déclaration sont souvent utilisées dans la déclaration des constantes et des variables. Les équations de modifications sont utilisées quand il y a besoin de modifier la valeur par défaut de l'attribut d'une variable.

Les équations dans le bloc *equation*

Les équations normales sont utilisées dans ce bloc qui débute par le mot clé « *equation* ». Plusieurs sortes d'équations peuvent être présentes dans ce bloc :

- Les équations à égalité simple.
- Les équations répétitives : *for-equation*.
- Les équations de connection : *connect-equation*.
- Les équations conditionnelles avec la clause *if*.
- Les équations conditionnelles avec la clause *when*.

[8] Quelques exemples d'équations sont présents dans le code détaillé au chapitre 5.

Les équations à égalité simple sont les équations de base issues des mathématiques. Les autres types d'équations sont transformés par le compilateur de Modelica en des équations à égalité simple.

Les équations d'initialisation

Il existe trois moyens possibles permettant de spécifier les contraintes initiales nécessaires pour résoudre certains problèmes d'initialisation :

- Spécifier les valeurs initiales à travers l'attribut *start*.
- Spécifier les contraintes initiales par des équations dans le bloc *initial equation*.
- Spécifier les valeurs initiales à travers des assignations dans le bloc *initial algorithm*.

2.4.2.5 Les composants, les connecteurs et les connexions

Les composants

Un modèle Modelica est typiquement constitué de composants issus de domaines divers interagissant entre eux à travers des connexions qui les lient. Un composant est une classe qui doit être défini indépendamment de l'environnement où elle est utilisée. Ceci est essentiel pour la réutilisabilité. Dans la définition des équations d'un composant, seules les variables locales et les variables connecteurs doivent être utilisées. Le seul moyen pour la communication entre plusieurs composants est le passage par les connecteurs. C'est au niveau de ces connecteurs qu'ont lieu les échanges d'information entre les composants au sein d'un modèle.

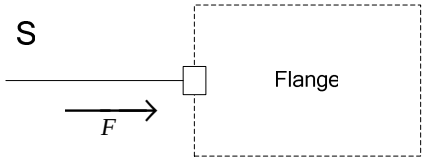
Les connecteurs

Un connecteur est une instance de la classe *connector*. Les connecteurs spécifient l'interface d'interaction entre les composants. Par exemple, la classe Pin est un connecteur utilisé comme interface pour les composants électriques. Cette classe a deux attributs, un de type Voltage et l'autre de type flow Current (flux).

```
connector Pin
Voltage v;
flow Current i;
end Pin;
```

Figure 2.4 Classe Pin

Les connecteurs peuvent êtres formulés pour plusieurs applications de différents domaines. Par exemple, dans le domaine de la mécanique, le connecteur *Flange* est utilisé pour décrire les interfaces pour une interaction à une dimension entre deux composants mécaniques en spécifiant la force *F* et la position *s* comme point d'interaction.




```
connector Flange
Position S;
flow Force f;
end Flange;
```

Figure 2.5 Un composant avec un connecteur pour la mécanique

Les connexions

Les composants peuvent êtres connectés via des connecteurs de type équivalent. Modelica supporte les connexions causales et les connexions acausales.

La direction du flux de données dans une connexion acausale est inconnue. Par contre, une connexion causale peut être établie en connectant un connecteur avec un attribut input à un connecteur avec un attribut output. Deux types de couplage peuvent être établis selon les variables des deux composants connectés. Ces variables peuvent être des variables de flux ou des variables non-flux :

- Couplage à égalité : Pour les variables non flux.
- Couplage (Somme=0) : Pour les variables flux selon la loi des noeuds de Kirchoff.

Le mot clé *flow* devant l'attribut i de type *Current* indique que tous les courants dans les pins connectés sont sommés à zéro. La connexion de deux pins se fait à l'aide du mot clé *connect*.

connect (pin1, pin2) produit deux équations :

$$pin1.v = pin2.v$$
$$pin1.i + pin2.i = 0$$

2.4.2.6 Modélisation des systèmes dynamiques hybrides sous Modelica

Le comportement des systèmes dynamiques hybrides est caractérisé par des sauts entre des états continus. Les fonctionnalités hybrides de Modelica permettent la modélisation des évènements discrets et des discontinuités des variables réelles.

Les variables discrètes

Les paramètres, les constantes et les variables ont une variabilité différente. Les paramètres et les constantes ont des valeurs constantes tout au long de la simulation alors que les variables de type *Real* peuvent changer continûment. Les variables *Integer* et *Boolean* sont constantes la plupart du temps mais peuvent changer discontinument vers de nouvelles valeurs.

Les variables sous Modelica sont discrètes si elles sont précédées par le mot clé *discret* ou de type *Boolean*, ou *Integer*, ou *String*. En appliquant le mot clé *discret* à une variable de type *Real*, cette dernière sera constante par morceaux puisque sa valeur ne sera plus modifiable seulement par une assignation au sein d'une clause *when*.

Génération d'évènements

Un événement est un fait qui survient à un moment donné. Il désigne un changement d'état ou de contexte lié à une modification substantielle de la valeur d'une variable mesurable dans un intervalle de temps. On peut distinguer deux sortes d'évènements : événement temporel et événement d'état.

a) Evènements temporels

Les évènements temporels sont générés par le passage du temps. Sous Modelica, ils sont spécifiés de manière absolue (date précise). Un événement temporel peut être généré par une expression conditionnelle à temps discret où le temps est utilisé avec une clause *when-equation* ou *when-expression*. Exemple[9] :

when *time*=<5 then

x :=x+1

end when

La condition *time* =< 5 dans la clause *when-equation* va générer un évènement temporel à temps=5. Un événement temporel peut être généré aussi par le passage d'une expression en fonction du temps comme argument à l'une des fonctions mathématiques prédéfinies suivantes : (*sign*, *rem*, etc.)**[Fritzon, 2003]**. Les fonctions *sample*, *initial* et *terminal* peuvent générer aussi des évènements temporels. Elles seront expliquées ci-après.

b) Evènement de changement d'état

Les évènements de changements d'état ne peuvent pas être programmés d'avance, ils sont reliés à des changements de valeurs des variables d'états du modèle. Un événement de changement d'état est généré par la satisfaction d'une expression

[9] *time* est une variable sous modelica qui renvoie le temps courant de la simulation

conditionnelle contenant au moins une variable d'état. Par exemple un événement est généré et détecté chaque fois la valeur de sin(x) dépasse 0.5.

when sin(x) > 0.5

….

end when;

Sous Modelica, on peut aussi générer des évènements d'une façon répétitive en utilisant la fonction *sample(valeur_initiale, intervalle_de_temps).* Cette fonction retourne vrai et peut être utilisée pour déclencher des évènements aux instants de temps *valeur_initiale + i * (i=0,1,..).* avec *valeur_initiale* est la date du premier événement et *intervalle_de_temps* est l'intervalle de temps entre les évènements périodiques. La fonction agit comme une sorte d'horloge qui retourne vrai périodiquement à chaque pas de temps déjà fixé. Elle retourne vrai entre les intervalles des évènements générés.

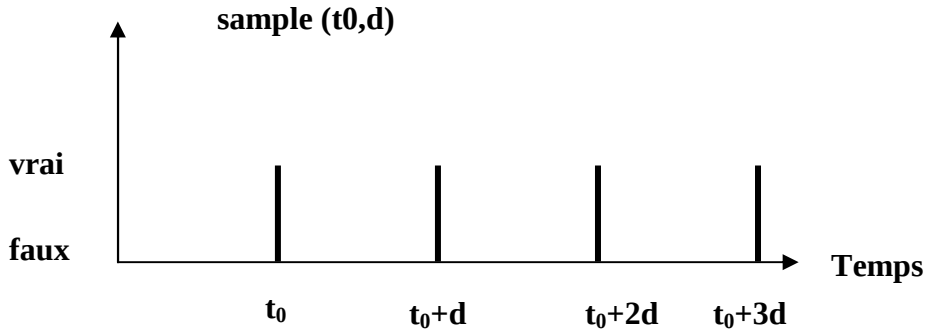


Figure 2.6 La fonction *sample()* de Modelica

Réinitialisations de variables

La réinitialisation des variables se fait par l'utilisation de la fonction *reinit*( ). L'effet de cette fonction est de stopper le temps de la simulation, changer les valeurs d'une ou plusieurs variables continues et reprendre la simulation.

Masquer des évènements

La fonction *noEvent(expr)* évalue l'expression *expr* et la retourne de telle façon qu'elle ne génère plus d'évènements. Dans certains cas, il est souhaitable d'empêcher certains événements de se produire. Dans la figure suivante l'expression *noEvent* évite de produire des événements qui seraient autrement déclenchés par (*expr*).

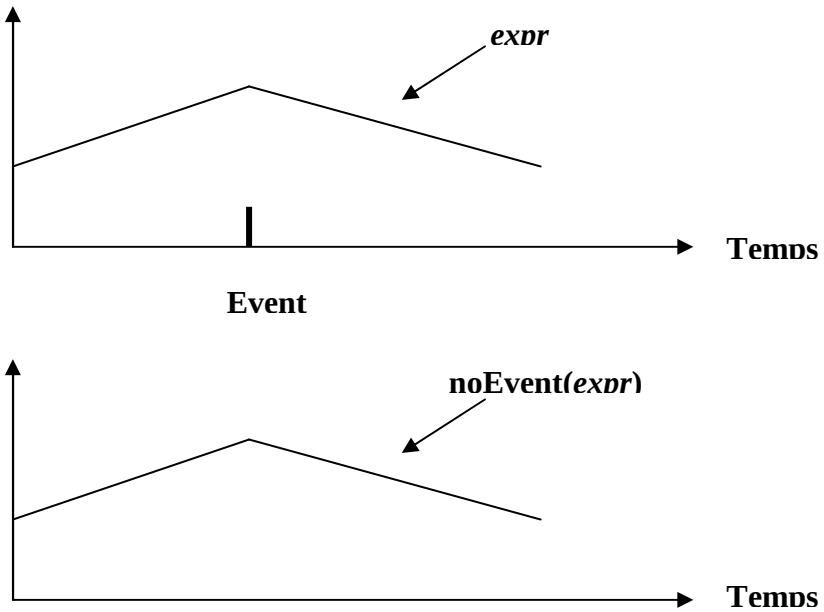


Figure 2.7 La fonction noEvent() de Modelica

Détection des changements

Pour détecter le changement des valeurs des variables sous Modelica, deux fonctions sont utilisées : *edge( )* et *change( ).* La fonction *edge* ne peut être utilisée qu'avec les variables booléennes. La fonction change ne peut être utilisée qu'avec les variables de type *Boolean*, *Integer* ou *String*. Chacune de ces fonctions a une expression conditionnelle équivalente.

edge(x) = x and not pre(x)

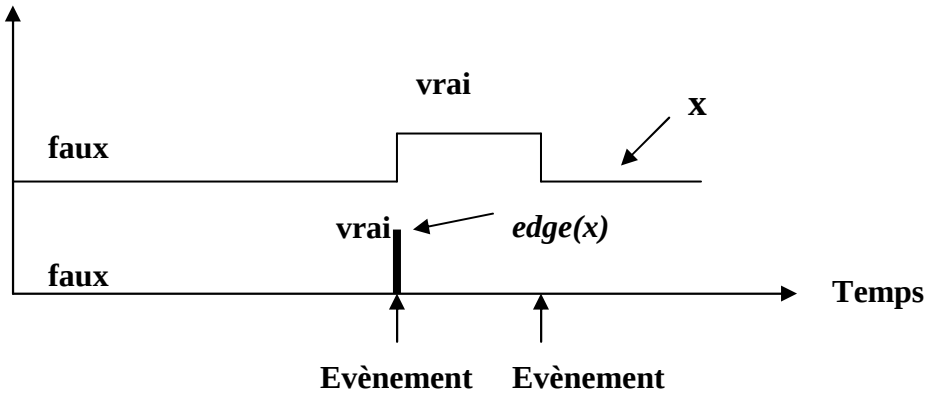


Figure 2.8 La fonction *edge()* retourne vrai quand x passe de faux à vrai.

La fonction *change* retourne vrai quand la valeur *x* est différente de sa valeur précédente.

change(x) = x <> pre(x)

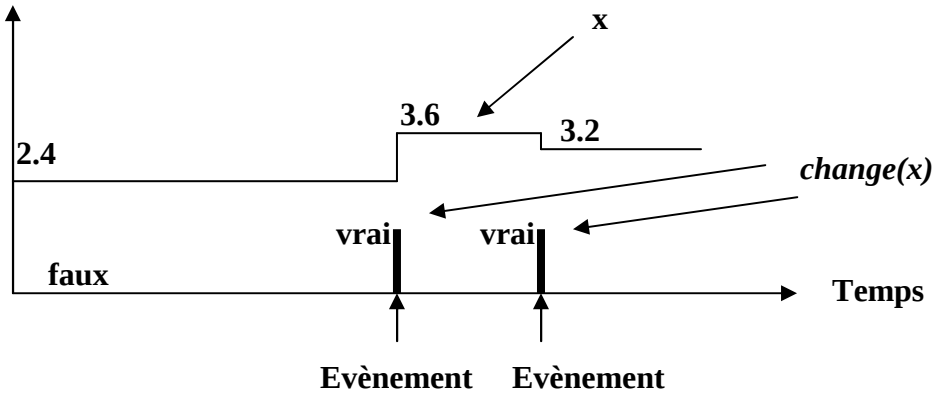


Figure 2.9 Comportement de la fonction *change()*

Identification du début et de la fin de la simulation

Pour identifier le début de la simulation, la fonction *initial( )* est utilisée. Elle renvoie vrai seulement à l'instant où la simulation démarre. La fonction *terminal( )* renvoie vrai seulement à l'instant où la simulation se termine avec succès. Elle est typiquement utilisée pour déclencher des évènements à la fin de la simulation, par exemple, appeler des fonctions externes pour enregistrer les valeurs de la simulation

dans un fichier. Si la simulation ne finit pas avec succès, elle sera interrompue prématurément.

Fin de la simulation

La fonction *terminate()* est utilisée pour mettre fin à la simulation. Cette fonction prend comme argument une variable de type String. L'une des raisons pour terminer une simulation est l'absence d'intérêt de continuer à simuler un comportement constant. Cette fonction est utilisée pour indiquer une fin réussie de la simulation.

2.4.3 OpenModelica

OpenModelica a deux buts à court terme et à long terme **[Fritzson, 2006]:**

- Le but à court terme est de développer un environnement informatique interactif efficace pour le langage Modelica ainsi que de fournir une implémentation plutôt complète du langage.
- Le but à long terme est d'avoir une implémentation de référence complète du langage Modelica et d'inclure également la simulation des modèles basés sur les équations et des facilités additionnelles dans l'environnement de programmation ainsi que le développement des spécifications complètement formelles pour le langage Modelica dont les sémantiques statique et dynamique.

La version courante de l'environnement OpenModelica permet l'exécution interactive de la plupart des expressions, algorithmes et des parties de fonctions de Modelica, ainsi que la génération du code C efficace à partir des modèles d'équations et des fonctions Modelica. Le code C généré est combiné avec une librairie de fonctions utilitaires, une librairie run-time et un solveur numérique DAE.

2.4.3.1 Vue d'ensemble du système

L'environnement OpenModelica consiste en plusieurs sous-systèmes connectés. Les sous-systèmes suivants sont actuellement intégrés **[Fritzon, 2007]**:

- Système de traitement interactif de session : analyse la syntaxe et interprète les commandes et les expressions Modelica pour l'évaluation, la simulation, l'affichage courbes, …etc.
- Un compilateur Modelica : traduit le code Modelica en code C avec une table symbolique contenant les définitions des classes, des fonctions et des variables. Ces définitions peuvent être prédéfinies, définies par l'utilisateur ou obtenues à partir des librairies. Le compilateur contient aussi un interpréteur Modelica pour l'usage interactif et l'évaluation des expressions constantes.
- Un module d'exécution : ce module exécute le code binaire compilé à partir des expressions et des fonctions traduites, aussi bien que le code de la simulation à partir des modèles basés sur les équations, lié avec les solveurs numériques.
- Un éditeur/explorateur d'un modèle textuel Emacs : N'importe quel éditeur de texte peut être employé. *Gnu-Emacs*[10] a été utilisé parce qu'il peut être programmé pour des futures extensions. Le mode Emac cache les annotations graphiques pendant la rédaction, ce qui rend la lecture plus facile.
- Un éditeur/explorateur plugin Eclipse : Le plugin Eclipse est appelé MDT (Modelica Development Tooling) fournit un explorateur hiérarchique des classes et des fichiers ainsi que des avantages lors de l'édition des modèles Modelica comme celles pour Emacs.
- Un éditeur de modèle OMNotebook DrModelica : Ce module fournit un éditeur calepin. Cette fonctionnalité permet de traiter le tutorial DrModelica.
- Un éditeur/explorateur de modèles graphiques : C'est un éditeur graphique de connexion pour la conception des modèles basés sur les composants en connectant les instances des classes Modelica et en explorant les librairies de modèles pour la lecture et la sélection des modèles de composants.

[10] GNU-Emacs est l'une des deux versions les plus populaires de l'éditeur de texte Emacs.

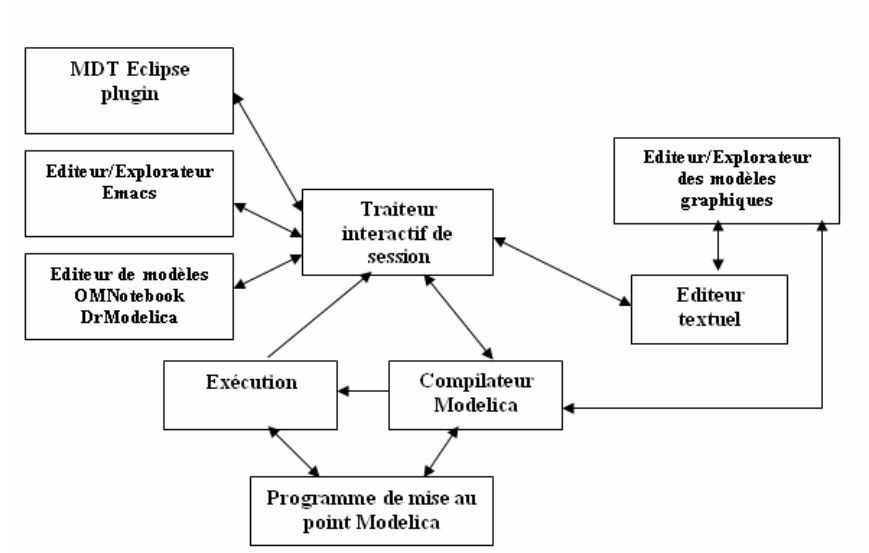


Figure 2.10 Vue d'ensemble d'OpenModelica

- Un programme de mise au point Modelica (debugger) : C'est le programme de mise au point conventionnel utilisant Emacs pour montrer le code source durant la progression, mettre des points de contrôle, etc. La trace arrière et les commandes d'inspection sont disponibles.

2.4.3.2 Les étapes de traduction du compilateur Modelica

Le processus de traduction du compilateur Modelica est décrit dans la figure 2.11. Le code source Modelica en entrée (des fichiers .mo) est transformé en un modèle plat. Dans cette étape, le compilateur vérifie les types des variables, effectue les opérations liées à la notion orientée objet tel que l'héritage et fixe l'inclusion des packages. Le modèle plat contient un ensemble d'équations de déclarations et de fonctions. Les deux étapes suivantes sont l'analyse des équations puis leur optimisation. Elles sont nécessaires pour compiler les modèles contenant des équations. En dernière étape, le code C généré est introduit au compilateur C pour produire un fichier exécutable.

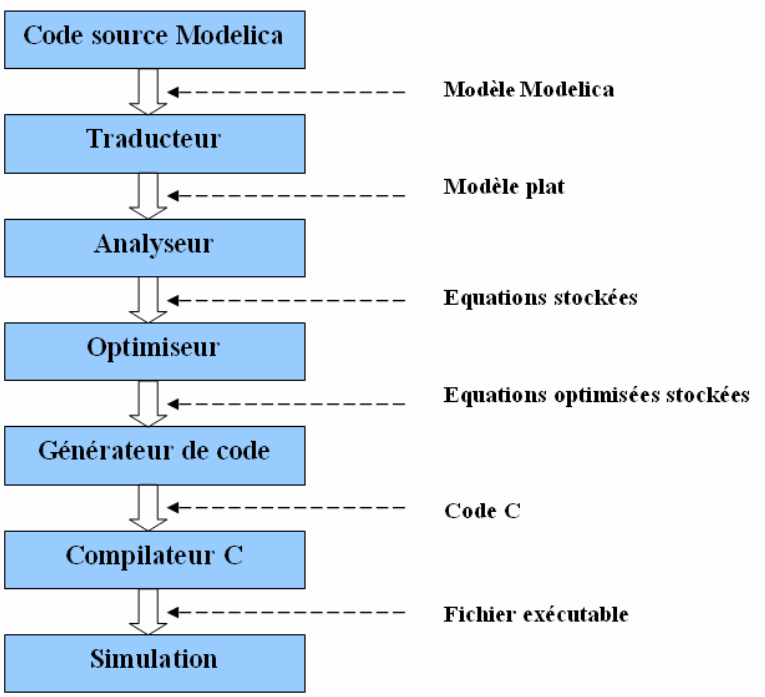


Figure 2.11 Etapes de traduction du code Modelica vers un fichier exécutable

2.4.4 Mathmodelica

MathModelica est un environnement de développement interactif pour la modélisation et la simulation des systèmes avancés. Il est composé par trois sous systèmes qui sont utilisés pendant les différentes phases du processus de modélisation et de simulation **[Site Web Mathmodelica]**.

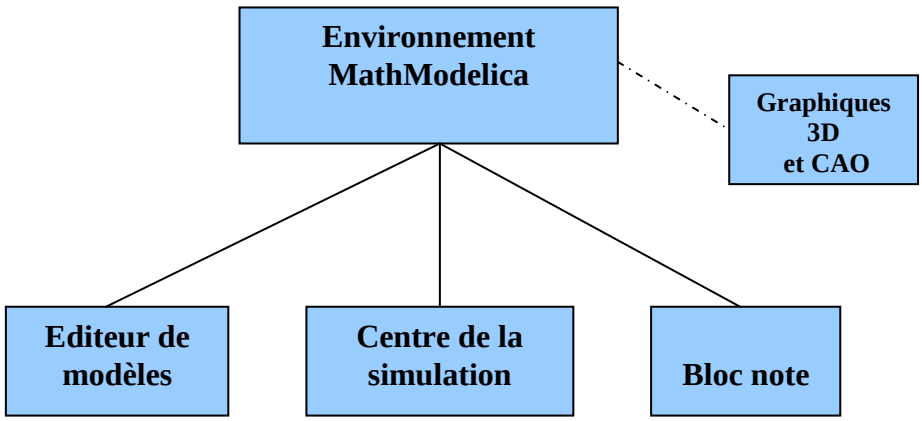


Figure 2.12 Architecture de l'environnement MathModelica

Les sous systèmes sont les suivants :

- Un éditeur graphique pour la conception des modèles à partir de la librairie des composants.
- Un bloc note interactif pour la programmation, la documentation et l'exécution des simulations.
- Le centre de la simulation pour l'initialisation des paramètres de la simulation, son exécution et l'affichage des courbes.

2.4.4.1 Editeur graphique des modèles

L'éditeur graphique des modèles MathModelica est une interface utilisateur où on peut construire des modèles en utilisant la méthode « glisser-déposer » à fin de glisser les classes des modèles de la bibliothèque standard de Modelica où à partir des composants des librairies définis par l'utilisateur. Ces composants sont souvent représentés par des icônes graphiques dans l'éditeur.

Cet éditeur peut être vu comme une interface utilisateur pour la programmation graphique sous Modelica.

2.4.4.2 Centre de simulation

Le centre de simulation est un sous système pour exécuter les simulations, initialiser des valeurs et des paramètres du modèle, afficher des résultats...etc.

2.4.4.3 Bloc note interactif

En plus de la programmation graphique utilisant l'éditeur des modèles, MathModelica fournit aussi un environnement de programmation pour créer des modèles Modelica textuels. Cet environnement textuel est le bloc note.

2.4.5 Dymola

Dymola (Dynamic Modeling Laboratory) **[Site Web Dymola]** est un environnement complet, basé sur le langage Modelica, pour la modélisation et la simulation des systèmes complexes intégrés dans différents domaines. Dassault Systèmes vient d'annoncer l'acquisition de Dynasim, société suédoise éditrice de la solution Dymola, ainsi que sa stratégie Catia Systems conçue pour placer la modélisation de systèmes embarqués au cœur de Catia. Le logiciel Dymola est notamment utilisé collectivement par les constructeurs automobiles allemands DaimlerChrysler, BMW, Audi et Volkswagen pour la modélisation et la simulation de systèmes d'air conditionné **[Communiqué Presse, 2005]**. Toyota qui utilise également Dymola au sein de ses divisions moteur, boîte de vitesse et châssis, a créé une bibliothèque pour le contrôle de ses moteurs diesel **[Soejima, 2000].**

La structure de Dymola est similaire aux autres environnements MathModelica et OpenModelica. Il est composé de :

- Un compilateur Modelica ainsi qu'un optimiseur pour réduire la taille des systèmes d'équations.
- Un solveur numérique pour les équations différentielles algébriques hybrides.
- Une interface graphique.
- Un éditeur texte pour les modèles Modelica.

2.4.5.1 Architecture de Dymola

Dymola **[Cours Dymola]** est basé sur l'utilisation des modèles Modelica stockés dans des librairies. Il peut aussi importer d'autres données et graphiques. Un traducteur contenu dans Dymola permet de générer du code C à partir des équations. Le code C généré peut être exporté à Simulink ou à des plateformes Hardware-In-The-Loop.

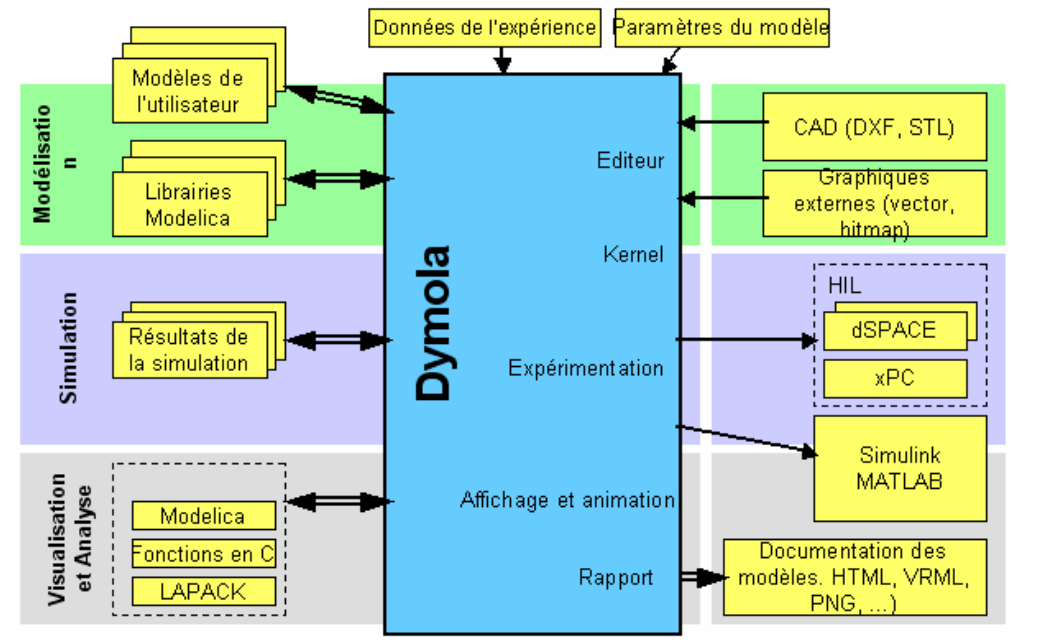


Figure 2.13 Architecture de Dymola

2.4.5.2 Interface graphique et textuelle

L'environnement Dymola possède deux éditeurs, l'un est graphique et l'autre est textuel.

Quelques caractéristiques de l'interface utilisateur :

- Les différents navigateurs de l'interface affichent des petites icônes des différents composants et classes. Une icône peut être glissée à partir de la liste hiérarchique du package vers l'éditeur graphique de connexion. En basculant vers l'éditeur textuel, le composant glissé apparaîtra comme du texte.
- Les icônes peuvent être créées avec l'éditeur graphique prédéfini. L'importation des fichiers bitmap créés par d'autres outils est possible.
- Dans le navigateur hiérarchique des composants, la structure hiérarchique des composants d'un modèle peut être vue.
- Un éditeur de texte prédéfini pour éditer les modèles est disponible. Il intègre des facilités de programmation : *Syntax highlighting*, utilisation des différents couleurs pour les mots clé…etc.)

2.4.5.3 Simulation temps réel HIL (Hardware-In-The-Loop)

Une des fonctions les plus puissantes de Dymola est la simulation temps-réel. Cette fonction est utilisée pour les systèmes HIL où un composant matériel est remplacé par un composant logiciel. En effet, une option temps réel est développée sous Dymola mais elle est payante.

La modélisation multi domaine mène souvent à des systèmes d'équations complexes qui ont deux types de solutions : des solutions qui varient lentement et d'autres rapidement. En effet, un système mécatronique est formé par des composants qui ont un comportement variable : Dynamique rapide comme les contrôleurs et dynamique lente comme les systèmes articulés et la mécanique des robots. Pour obtenir des simulations en temps réel, des méthodes pour la résolution des modèles et des méthodes pour la génération du code ont été développées. Par exemple, les composants dont les variables varient lentement sont discrétisées avec la méthode explicite d'Euler, alors que les composants dont les variables varient rapidement sont discrétisées avec la méthode implicite d'Euler. L'utilisation de cette dernière méthode nécessite la résolution d'un système d'équations à chaque pas de temps. Ce qui prend beaucoup de temps de traitement. Ceci peut être un inconvénient pour les simulations temps réel.

La méthode d'intégration « *inline* » a été proposée pour traiter ce cas. Les formules de discrétisation de la méthode d'intégration sont combinées avec les équations du modèle. Le système d'équation global est réduit en une forme plus efficace. Par exemple, pour un modèle robotique comportant 66 états, la taille du système d'équations non linéaires peut être réduite à 6. **[Dymola User Manual]**

2.4.6 Choix de l'outil

Afin d'avoir une architecture finale qui permet de communiquer deux simulateurs en utilisant HLA, le choix d'un simulateur à source ouverte est nécessaire

pour faciliter l'intégration des services HLA dans le simulateur. Par conséquent, nous avons choisi d'utiliser le simulateur OpenModelica.

2.5 Environnements virtuels

La demande sur les applications de réalité virtuelle est en croissance continuelle, et pour cela un grand nombre de nouveaux environnements virtuels ont été crées. La plupart des laboratoires travaillant dans le domaine de la réalité virtuelle ont décidé de créer leurs propres environnements virtuels pour répondre à leurs problèmes spécifiques, on peut noter par exemple la plate-forme VEMAT qui est un environnement virtuel pour la fabrication sur des machines industrielles complexes par le web, développé par TEMASEK polytechnique de Singapore **[Khon, 2002]**. La plate forme VFMS développé par KAIST (Korea advanced institute of Technology) qui est un environnement virtuel spécifique pour modéliser les systèmes industriels flexibles **[Sang, 2001]**. On peut citer aussi l'environnement VISUM développé à l'université de Karlsruhe en Allemagne pour simuler les systèmes mécatroniques et en particulier les robots humanoïdes **[Finkenzeller, 2003]**. Le fait de créer son propre environnement virtuel prend beaucoup de temps mais mène à l'utilisation spécifique d'un environnement virtuel qui convient à son application de réalité virtuelle. Une autre solution est d'utiliser un environnement virtuel générique. Ce type d'environnement doit répondre à certains critères de spécification : **[Bierbaum, 1998]**

- Une performance qui permet à une application de réalité virtuelle de tourner en temps réel.
- L'EV doit être flexible pour pouvoir interagir un grand nombre de hardware et de software.
- L'EV doit être simple à utiliser.

On va présenter dans ce qui suit une architecture générale d'un environnement virtuel et les environnements virtuels génériques les plus utilisés en mettant l'accent chaque fois sur les critères listés en dessus.

2.5.1 OpenMask

OpenMASK **[Site Web OpenMASK]** est une plate-forme pour le développement et l'exécution d'applications modulaires dans le domaine de l'animation, de la simulation et de la réalité virtuelle. OpenMASK est un software open source développé à l'Irisa de Rennes.

L'objectif principal d'OpenMASK est de fournir un noyau d'animation et de simulation : **[Margery, 2002]**

- Indépendant du niveau d'animation (descriptive, générative or comportementale) utilisé ;
- Indépendant du style de programmation de l'animation (réactive, orienté agent, objets actifs . . .) utilisé ;
- Indépendant de la bibliothèque de rendu utilisée ;
- Capable d'utiliser plusieurs activités en parallèle pour le calcul de la simulation ;
- Indépendant du type d'exécution multi activité utilisé (calcul distribué ou calcul parallèle).

OpenMASK a une architecture qui permet de programmer indépendamment de l'environnement (dispositifs externes,..). OpenMASK a une architecture modulaire ; Les modules de simulation sont reliés entre eux par un bus. Un module de simulation sert à décrire tous les éléments constitutifs de l'environnement virtuel que l'on veut réaliser. On peut décrire par exemple un objet virtuel, une interface d'interaction ou même le comportement d'un autre module.

La granularité du module de simulation n'est pas imposée par *OpenMASK*. En effet un module de simulation peut servir à décrire aussi bien un humanoïde et son comportement qu'une simple sphère. Il est laissé à l'utilisateur le choix de la granularité des modules de simulation. Le module *OpenMASK* a une architecture prédéfinie.

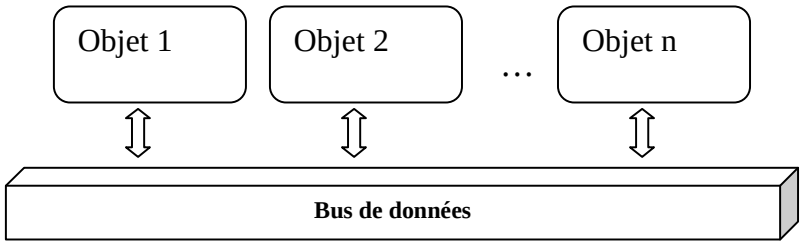


Figure 2.14 Les objets sous OpenMASK

Pour une application donnée, les objets de simulation sont structurés dans un arbre de simulation. L'objet racine de cet arbre est le contrôleur, mais le reste de la sémantique de l'arbre de simulation est laissé au concepteur d'application. **[Chauffaut, 2003]**

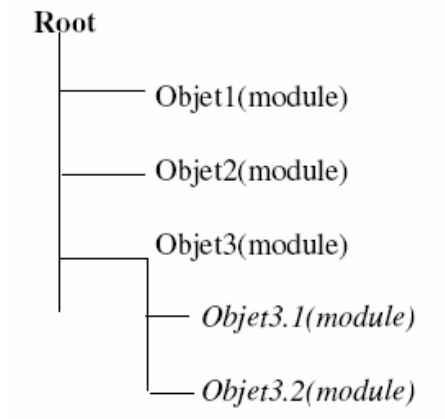


Figure 2.15 Arbre de simulation

Les plus importantes méthodes à décrire sont la méthode d'initialisation *init()*, la méthode du calcul du comportement *compute()*, et la méthode de traitement des évènements échangés par les modules OpenMASK *ProcessEvent()*. Le comportement est programmé en C++ dans la méthode *compute()*. La méthode de programmation du comportement est laissée au choix de l'utilisateur. Chaque pas de simulation, le comportement des modules est mis à jour par le contrôleur.

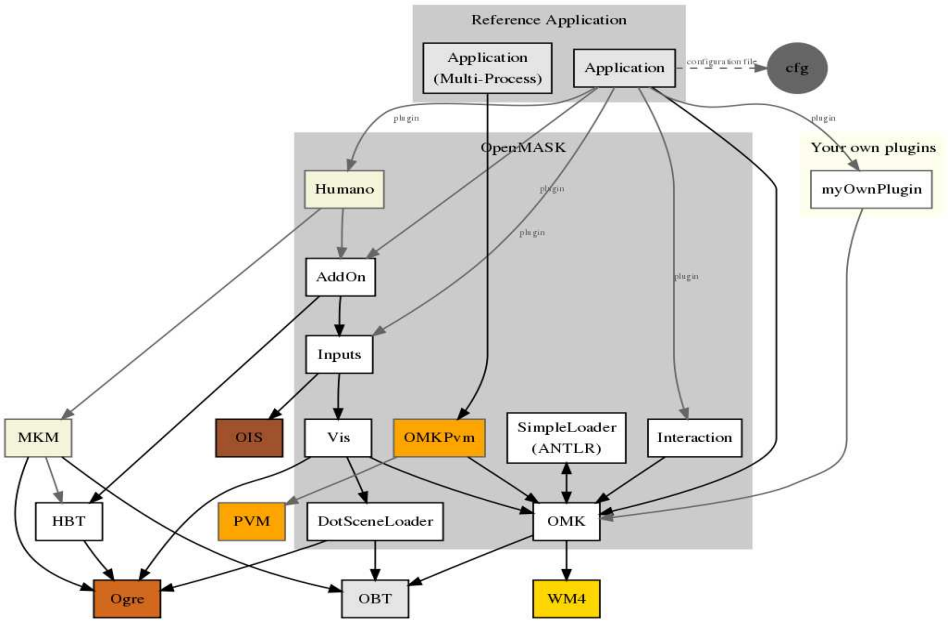


Figure 2.16 Architecture de OpenMASK

Synthèse

L'environnement virtuel OpenMASK a une très bonne performance temps réel grâce à la distribution via PVM (Parallel Virtual Machine). Pour les applications non distribuées, OpenMASK propose un module Clock, qui une fois instancié et paramétré synchronise la simulation 3D avec le temps réel. Un autre avantage est l'utilisation d'une approche modulaire pour créer une simulation sous OpenMASK. Cette approche se base sur les objets simulés et les objets visuels. Chaque objet simulé est une classe C++. Ceci favorise la réutilisation. Des extensions prédéfinies sous OpenMASK4 peuvent déjà être utilisées pour des animations simples. Un autre avantage est sa flexibilité: par exemple la possibilité d'interfacer matériel et/ou logiciel. L'inconvénient majeur de cette plateforme est la difficulté d'utilisation en l'absence de documentation suffisante ce qui rend son utilisation difficile.

2.5.2 VrJuggler

VRJuggler **[Site Web VRJuggler]** est un système logiciel pour le développement et l'exécution d'applications de réalité virtuelle. Son architecture fait appel à plusieurs techniques de technologie de la programmation ; entre autres les

Design pattern. Le but de *VRJuggler* est de présenter une couche logicielle entre l'application et le matériel pour fournir un modèle indépendant du matériel. Cette couche contient un ensemble d'interfaces bien définies aux composants de la réalité virtuelle basés sur leur fonctionnalité. Par exemple, un système de *tracking* à 6 degrés de liberté est manipulé par un composant d'interfaçage de position. De cette façon, si l'application doit employer un système *tracking* différent, il n'y aura pas besoin d'introduire des changements sur le programme. Aussi longtemps que le nouveau dispositif est commandé par l'interface de position, l'application n'aura besoin d'aucun changement de code. De cette façon, les applications sont concernées seulement par le type d'information qu'elles doivent recevoir et envoyer au système de réalité virtuelle. *VRJuggler* prend soin de convertir cette information en représentation fondamentale requise pour chaque composant particulier.

Pour réaliser l'indépendance vis-à-vis du matériel, l'architecture de VRJuggler est basée sur un micro noyau, qui utilise un certain nombre de managers. Chacun de ces managers est dédié à une tache bien précise. Le micro noyau tient le rôle de médiateur entre tous ces managers. Ce qui implique les taches suivantes: **[Bierbaum (a), 2001]**

- Maintenir l'exécution interactive du système et des applications.
- Coordonner les interactions entre les managers.
- Contrôler les communications entre les managers et les applications.
- Maintenir la synchronisation des composants du système
- Prendre en charge la reconfiguration d'exécution du système de VR.
- Diriger l'exécution des applications multiples

La figure suivante montre un diagramme du noyau de VRJuggler, les managers courants et les liaisons entre eux :

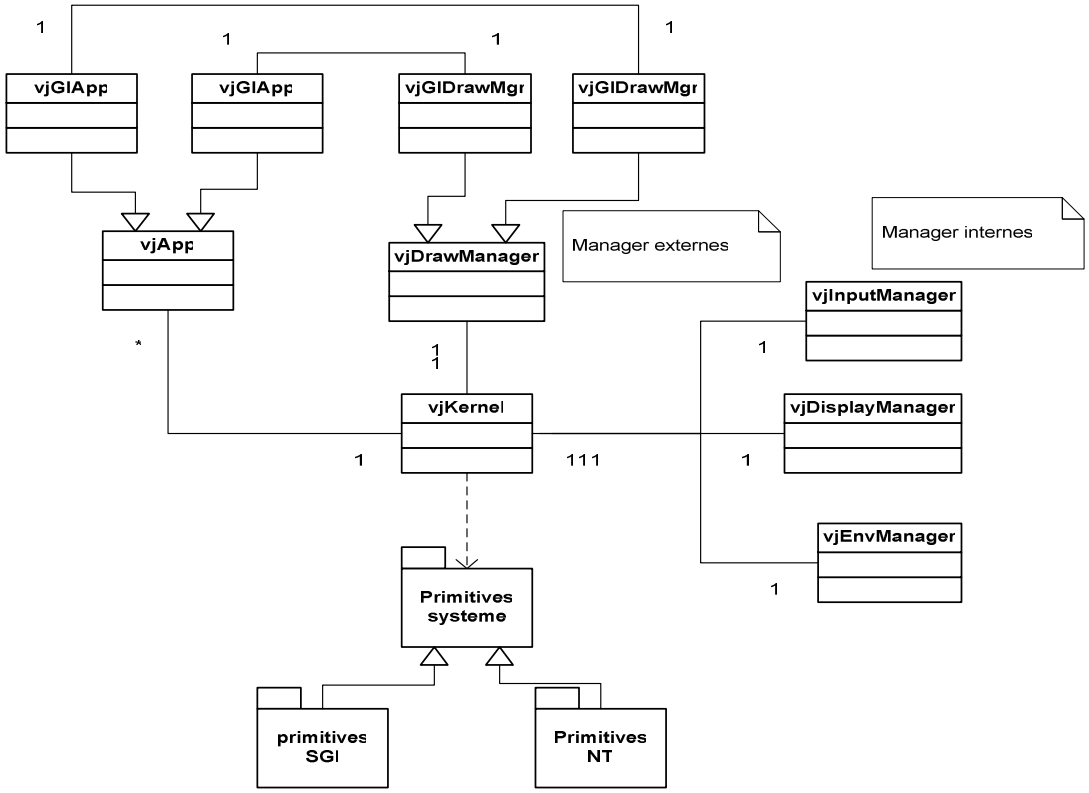


Figure 2.17 Architecture de VRJuggler

Les Managers dans le VRJuggler commandent les éléments spécifiques du système de réalité virtuelle. Ils sont séparés dans deux catégories **[Bierbaum (b), 2001]**: managers internes et managers externes. Les managers internes manipulent la fonctionnalité centrale au delà de la portée du noyau ; les directeurs externes traitent, quant à eux, les interfaces à l'application.

Synthèse

VRJuggler comporte un avantage majeur qui est sa flexibilité. En effet comme son nom l'indique (jongleur) il permet d'interfacer un grand nombre de logiciel et de matériel. Un autre avantage de cet outil est qu'il est à code ouvert. On peut donc facilement l'étendre ou accéder au fonctionnement du noyau. *VRJuggler* permet aussi une exécution temps réel de grandes applications grâce à la distribution. Il comporte, néanmoins, quelques inconvénients. En outre, le fait qu'il ne permet pas nativement

une modélisation modulaire des applications. En effet chaque application est modélisée dans une classe *C++*.

2.5.3 3dVia Virtools

L'environnement de réalité virtuelle 3dVia Virtools **[Site Web Virtools]** permet de développer des applications 3D temps réel et de définir des interactions avec l'utilisateur. Virtools est une technologie Dassault Systèmes. Il est très utilisé dans le milieu du jeu vidéo, de la réalité virtuelle et du multimédia en générale.

Virtools se base sur 5 composants clés: l'interface graphique (Graphical User Interface), le moteur de comportement (Behavior Engine), le moteur de Rendu (Render Engine), le langage de Script Virtools (VSL) et le kit de développement SDK.

L'interface graphique (GUI):

Cette interface est utilisée pour développer des applications 3D en assemblant visuellement les objets et les comportements. Elle est composée de panneaux de configurations, d'outils de graphiques de navigation, des vues schématiques...etc.

Le moteur de comportement

Le moteur de comportement sert à déployer et diffuser des applications interactives. Il gère à la fois des comportements standards et personnalisés. Virtools offre une collection de comportements réutilisables qui permet de créer à peu près tout type de contenu, sans une seule ligne de code, à partir de l'interface graphique de l'éditeur de schéma. Ses comportements réutilisables sont des blocks de comportement appelés Bbs (Behavior Building Blocks).

Le langage de Script Virtools (VSL)

C'est un langage performant pour créer des fonctions spécifiques de bas niveau sans utiliser le C++.

Le moteur de rendu

Le moteur de rendu fournit un rendu d'images et d'animations 3D temps réel de qualité. Il inclut les caractéristiques suivantes :

- Support des standards industriels : DirectX et OpenGL
- Support des Vertex et Pixel Shaders (DX9.c, OpenGL 2.0, HLSL, CgFX, Shader Model 3)
- Support de l'importation de modèles 3D et d'animation depuis 3ds Max®, Maya®, XSI® , Lightwave® et Collada®

Le Kit de développement SDK

Le Kit SDK de Virtools est un kit de développement qui fournit un accès à toutes les fonctionnalités bas niveau utilisées par Virtools. Avec le SDK, on peut créer de nouveaux comportements (DLL), modifier les opérations des comportements existants, écrire de nouveaux outils afin d'importer ou d'exporter de nouveaux formats, et enfin de modifier ou remplacer le moteur de rendu.

Il existe plusieurs autres packs ou librairies tels que: Le VR Pack qui permet de gérer la stéréoscopie, la librairie AI Library (Artificial Intelligence Library) qui permet d'intégrer une intelligence artificielle pour la gestion de la foule... etc.

Synthèse

Un des avantages majeurs de Virtools est qu'il réduit considérablement le temps de développement d'applications 3D temps réel. En effet, Virtools permet de modéliser le comportement des objets 3D dans une scène par des blocs fonctionnels en utilisant une approche à haut niveau d'abstraction. Un autre avantage de Virtools est sa flexibilité. Plusieurs librairies ont été développées pour permettre d'interfacer Virtools à des périphériques VR standards tel que la librairie VR Library. L'inconvénient majeur de cet environnement est le fait qu'il se base sur une approche interprétée et non compilée. Ceci peut induire sur la performance de l'exécution en mode temps réel surtout dans le cas d'un grand nombre d'objets virtuels. Un dernier inconvénient de Virtools est qu'il se base sur une approche non open source. Cet inconvénient rend

compliqué l'ajout de nouveaux blocs de comportement ou l'interfaçage avec des librairies externes.

2.5.4 Choix du simulateur 3D

Afin d’obtenir une architecture finale complètement à source ouverte, nous avons opté pour les environnements virtuels à code ouvert (Open Source) pour notre approche. De plus, ces derniers sont extensibles et la tâche de les rendre compatibles HLA est moins difficile. Entre OpenMASK et VrJuggler, le premier semble le mieux adéquat à notre application parce que la modélisation est orientée module par contre la modélisation sous VrJuggler est orientée application. En effet, sous VrJuggler, une application est représentée par un objet, alors que sous OpenMASK, une entité virtuelle est représentée par un objet. Il est évident qu’OpenMASK est plus modulaire. Un des principaux avantages de la modularité est de permettre la réutilisation de composants entre les applications. Un autre est d'autoriser la génération automatique de composants depuis des outils dédiés. Suite à l’étude précédente notre choix s’est porté sur l’environnement virtuel OpenMASK. En effet c’est celui qui répond le mieux aux critères les plus importants que nous nous sommes donnés.

2.6 Conclusion

Nous avons présenté à travers cet état de l’art les principaux formalismes pour la modélisation des systèmes dynamiques hybrides afin de modéliser le comportement des systèmes mécatroniques, ainsi qu’une comparaison brève de quelques formalismes selon des critères détaillés précédemment. Nous avons fait le tour des principaux outils de simulations des systèmes à comportement hybride. Une comparaison concise de ces outils nous a mené à choisir le simulateur OpenModelica, basé sur le langage Modelica, pour implémenter les automates hybrides. Nous avons ensuite présenté les environnements virtuels les plus pertinents, ainsi qu’une synthèse pour choisir l’environnement le plus adéquat à notre application qui est le simulateur OpenMASK.

3. Chapitre 3 : Simulation distribuée

3.1 Introduction

Les simulations distribuées ou parallèles sont très intéressantes dans plusieurs situations tels que :

- Les applications à temps critique où la simulation est utilisée comme un outil d'aide à la décision et les résultats doivent être communiqués rapidement.
- La conception des systèmes complexes où l'exécution de la simulation consomme beaucoup de temps.
- Les environnements virtuels pour l'entraînement où les ressources et participants sont répartis dans plusieurs parties géographiques.

Généralement on divise les simulations en deux catégories : simulations analytiques et environnements virtuels distribués. Cette distinction est intéressante parce que chaque catégorie présente plusieurs exigences et défis technologiques.

Notre travail s'inscrit dans le cadre de la simulation en vue d'utiliser une architecture pour la simulation distribuée afin de permettre à deux simulateurs de communiquer sur une même machine. Dans ce chapitre, nous présentons la simulation en général, ses intérêts, ses utilisations et ses dangers. Dans le cadre de notre étude, nous considérerons essentiellement les simulations à événements discrets. Nous verrons au chapitre 5 que notre application portera sur une simulation distribuée des systèmes dynamiques hybrides.

Nous mettons l'accent sur les simulations à évènements discrets en présentant les notions de base (temps réel, temps simulé, mécanismes d'exécution des évènements…etc.). Nous décrirons par la suite les simulations distribuées à évènements discrets. Nous faisons l'état de l'art des approches et algorithmes d'exécution de ces simulations.

3.2 Simulation : définition, intérêts et dangers

3.2.1 Définitions

Avant de présenter une définition de la simulation, nous présentons une définition de la notion de *modèle*. Un modèle est une abstraction de la réalité. C'est une vue subjective mais pertinente de la réalité. Le caractère abstrait d'un modèle doit notamment permettre de faciliter la compréhension du système étudié. Il réduit la complexité du système étudié, permet de simuler le système, le représente et reproduit ses comportements. Plus simplement, on peut définir un modèle comme suit :

Un modèle est toute chose qu'on peut expérimenter pour donner des réponses à des questions sur le système. **[Fritzon, 2003]**

Ceci implique qu'un modèle peut être utilisé pour répondre à des questions sur le système sans faire des expériences sur le système réel. Plusieurs sortes de modèles existent. Nous nous intéressons dans notre travail aux modèles mathématiques représentés de manières différentes : équations, fonctions, programme informatique…etc.

Lors de la définition d'un modèle nous avons mentionné la possibilité d'expérimenter un modèle au lieu du système réel lui correspondant. En effet, ceci est l'utilisation principale des modèles. Elle est appelée par le terme *simulation*. Nous pouvons ainsi définir la simulation comme suit :

Une simulation est une expérience effectuée sur un modèle. **[Fritzon, 2003]**

Il est important de se rendre compte que *la description d'une expérience* et *la description d'un modèle* sont deux parties séparées de point de vue conceptuelle. D'autre part, ces deux aspects d'une simulation vont de pair, même si elles sont séparées. Par exemple, un modèle est valide seulement pour certaines expériences. Il est peut être utile de définir un ensemble de conditions, que les expériences valides doivent remplir, avec chaque modèle.

3.2.2 Intérêts de la simulation

Il existe plusieurs raisons pour effectuer des expériences sur des modèles au lieu du système réel :

- Les expériences sont coûteuses, dangereuses, ou le système à étudier n'existe pas encore. Ce sont trois principales difficultés d'expérimentation sur des systèmes réels.
- L'échelle de temps des dynamiques du système est incompatible avec l'expérimentateur. Par exemple, cela prend des millions d'années pour observer des petites transformations dans le développement de l'univers, alors que les mêmes transformations peuvent êtres observées sur une simulation sur un ordinateur.
- Les variables peuvent êtres non observables : Dans une simulation, toutes les variables peuvent être contrôlées et modifiées même celles qui sont inaccessibles dans un système réel.
- La manipulation facile des modèles : en effet, certaines variables ou paramètres de la simulation peuvent être modifiées plus facilement. Par exemple, le changement d'un paramètre lié à un poids dans une simulation est facile alors que dans le système réel la manipulation est plus difficile.
- Suppression des perturbations : Dans une simulation, il est possible de supprimer des perturbations qui semblent inévitables lors des mesures effectuées sur le système réel. Ceci nous permet d'isoler certaines perturbations et ainsi de mieux comprendre leurs effets.
- Suppression des effets secondaires : souvent, les simulations sont parfaites si elles permettent la suppression des effets secondaires. Ceci peut aider à une meilleure compréhension des effets de premier ordre.

3.2.3 Dangers de la simulation

L'expression « dangers de la simulation » a été utilisée par Peter Fritzon dans **[Fritzon, 2003]**. En effet, la facilité d'utilisation de la simulation peut être un

inconvénient. En effet, il est fort possible que l'utilisateur oublie les limitations et les conditions pour que la simulation soit valide, et ainsi il tire de fausses conclusions de la simulation. La meilleure façon de résumer cette idée est la célèbre phrase de Alfred Korzybski « la carte n'est pas le territoire ». Ainsi, la simulation qui tend à devenir un outil indispensable dans l'industrie du fait de sa rapidité de mise en œuvre ne doit pas être utilisée sans prise de recul scientifique et prudence.

Pour réduire ces inconvénients, l'utilisateur doit toujours comparer quelques valeurs des résultats de la simulation à des valeurs des résultats de l'expérimentation sur le système réel. Ceci permet aussi d'éviter de tomber dans les trois problèmes suivants :

- Tomber en amour **[Fritzon, 2003]** avec le modèle : l'effet Pygmalion[11]. En effet, il est possible d'admirer un modèle tout en oubliant les conditions et les contraintes établies pour que la simulation soit valide. Exemple : l'introduction des renards dans le continent australien pour résoudre le problème des lièvres. Une supposition du modèle est que les renards vont attaquer les lièvres, ce qui est vrai dans d'autres parties de la planète. Malheureusement, les renards ont trouvé les faunes indigènes plus simples à attaquer et ont ignoré largement les lièvres.
- Forcer la réalité dans les contraintes du modèle : le syndrome de Procuste[12]. Un exemple est le façonnage de notre société après les théories économiques à la mode qui ont un point de vue simplifié sur la réalité et qui ont ignoré plusieurs aspects importants du comportement humain, de la société et de la nature.
- L'oubli du degré de précision du modèle : La plupart des modèles simplifient leurs hypothèses et nous devons prendre en compte leurs hypothèses lors de l'interprétation des résultats de la simulation.

[11] Pygmalion, sculpteur chypriote de l'Antiquité, a créé, d'après la légende, une statue de femme d'une telle beauté qu'il en est tombé amoureux. Ayant demandé aux dieux de donner vie à cette statue, la déesse Aphrodite l'a exaucé.

[12] "Procruste ou Procuste: dans la légende grecque, Procruste est un bandit qui hante la route près d'Eleusis. Quand des voyageurs acceptent son invitation d'être son hôte, il les étend sur un des deux lits qui meublent son logis, l'un très long, l'autre très court. Si l'hôte est trop grand pour le lit de petite taille, Procuste lui ampute les jambes, s'il occupe l'autre lit. Procuste l'étire jusqu'à ce que le hôte cadre avec le lit. Thésée tua le bandit.

3.3 Simulation à évènements discrets

3.3.1 Le temps

Il y a plusieurs sortes de temps quand on parle de simulation : Temps physique, temps simulé et temps d'horloge. Un programme de simulation peut, à tout moment, obtenir la valeur courante du temps d'horloge physique du système d'exploitation. Le temps simulé est un nouveau concept qui n'existe que dans le monde de la simulation et il est défini comme suit :

Définition :

Le temps simulé est défini comme un ensemble de valeurs totalement ordonnées où chaque valeur représente un instant du temps du système physique à modéliser. Pour deux valeurs de temps de simulation, T_1 représentant le temps physique P_1 et T_2 représentant le temps physique P_2 :
Si $T_1 < T_2$ alors P_1 se produit avant P_2 et $(T_2-T_1) = (P_2-P_1)*K$ où K est une constante.
Si $T_1 < T_2$, on dit alors que T_1 se produit avant T_2 et si $T_1>T_2$ on dit alors que T_1 se produit après T_2.

Pour n'importe quelle simulation, une seule échelle de temps de simulation et globale est utilisée et qui est reconnue par tous les composants de la simulation, comme tous les pays reconnaissent le temps de Greenwich. Ceci assure que toutes les parties de la simulation ont une compréhension unique des relations « avant et après » entre les actions simulées qui surviennent à des instants spécifiques du temps simulé.

3.3.2 Temps réel et temps réel mis à l'échelle

La progression du temps simulé pendant l'exécution d'une simulation peut ou non avoir une relation directe avec la progression du temps d'horloge. Pour les simulations utilisées dans les environnements virtuels, le temps simulé doit avancer en synchronisation avec le temps d'horloge sinon l'environnement simulé paraîtra non réel. Si le temps de la simulation avance lentement par rapport au temps d'horloge

alors l'environnement virtuel paraîtra lent et répondra aux interactions de l'utilisateur en retard. De même, si la simulation avance rapidement par rapport au temps d'horloge, les utilisateurs ne pourront pas interagir avec la simulation.
Les simulations synchronisées avec le temps d'horloge sont souvent des simulations temps réel. Les simulateurs qui opèrent avec ce mode sont appelés des simulateurs temps réel. A cause de cette relation entre le temps d'horloge et le temps simulé, souvent on mélange les deux dans la littérature des simulations distribuées. Cependant, il faut toujours garder ces deux concepts distincts.

Une variante des exécutions temps réel est l'exécution temps réel mis à l'échelle (*Scaled Real Time*). Dans cette exécution, le temps simulé avance lentement ou rapidement proportionnellement au temps d'horloge avec un facteur constant. Par exemple, si la simulation est réglée pour avancer 2 secondes de temps simulé pour chaque seconde du temps d'horloge, alors cela permet à la simulation de s'exécuter deux fois plus rapidement que le monde réel. Ceci est utilisé souvent pour omettre les parties « non intéressantes » de la simulation. De même, on peut ralentir la simulation par un facteur constant pour fournir une vue détaillée d'une partie de la simulation. Une exécution temps réel est un cas particulier d'une exécution temps réel mis à l'échelle où le facteur de proportion est égal à un.

La simulation temps réel et temps réel mis à l'échelle utilisent une fonction de mappage pour transformer le temps d'horloge en un temps simulé **[Fujimoto, 2000]**. Cette fonction est décrite par la transformation :

$$T_s = W2S(T_w) = T_{debut} + K \times (T_w - T_{wdebut})$$

où :

T_w est une valeur du temps d'horloge.

T_{debut} est la valeur du temps simulé au début de la simulation.

Tw_{debut} est le temps d'horloge au début de la simulation.

K est un facteur de proportion.

Si, par exemple, K =2 alors la simulation s'exécute deux fois plus vite que le temps d'horloge. Avancer une seconde en temps d'horloge correspond à avancer deux secondes en temps simulé.

Dans les simulations analytiques qui n'incluent pas les humains ou le matériel comme composant dans la simulation, la progression du temps est souvent non synchronisée par rapport au temps d'horloge. Ces simulations désignent parfois les simulations « *aussi-rapide-que-possible* » parce qu'on voudrait souvent finir la simulation le plutôt possible sans maintenir une relation fixe avec le temps d'horloge. On peut concevoir des simulations qui peuvent être exécutées en temps réel ou *aussi-rapide-que-possible*. En effet, on peut ralentir facilement une simulation mais il est souvent difficile de l'accélérer. Un mécanisme d'attente peut être utilisé pour empêcher la simulation d'avancer son temps simulé au delà du temps d'horloge.

3.3.3 Mécanismes d'exécution des évènements

Jusqu'ici on s'est focalisé sur l'aspect temporel concernant l'exécution de la simulation. Une autre méthode de classification correspond à la manière dont l'état du modèle varie pendant l'avancement de la simulation (*Time Flow Mechanism*). La figure suivante représente une classification des modèles de simulation.

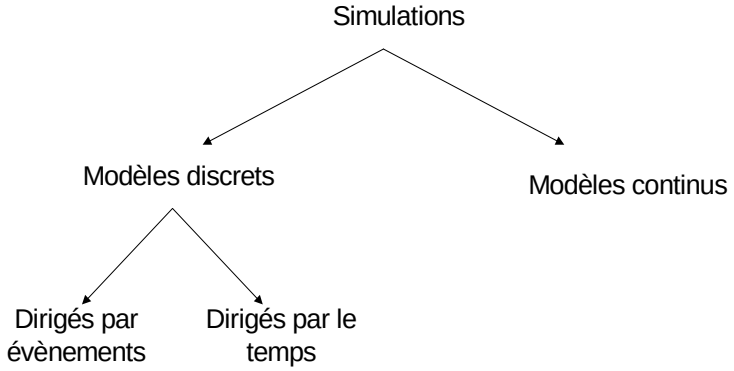


Figure 3.1 Classification des simulations

Les modèles de simulations peuvent être, de façon générale de deux types : les modèles continus et les modèles discrets. Dans une simulation continue, l'état du système change d'une manière continue au cours du temps. Le comportement d'un tel

système est celui décrit typiquement par un ensemble d'équations différentielles qui décrivent la variation de l'état du système en fonction du temps de la simulation. Dans une simulation discrète, le modèle de simulation voit le système physique comme un ensemble de changement d'état à des points discrets du temps de simulation. D'un point de vue conceptuel, le système est modélisé par un ensemble de sauts d'un état à un autre. On peut diviser les simulations discrètes en deux catégories : simulation dirigée par évènements et simulation dirigée par le temps. Ces deux catégories se différencient par le mécanisme d'avancement du temps (Time Flow Mechanism).

- *Exécution dirigée par le temps*

Dans une simulation dirigée par le temps, les variables d'état du système sont calculées à chaque pas de temps fixe. Par contre, une variable d'état peut ne pas changer de valeur d'un pas de temps à un autre.

Une simulation temps réel ou temps réel mise à l'échelle peut être une simulation dirigée par le temps. En effet, la simulation doit contrôler son avancement pour qu'elle soit synchronisée avec le temps d'horloge. Une simulation dirigée par le temps « non temps réel » doit calculer d'une façon répétitive l'état du système à la fin de chaque pas de simulation. Pour la version temps réel, on a la même chose sauf que la simulation attend jusqu'à ce que le temps d'horloge s'écoule avant d'avancer au pas du temps suivant. L'algorithme suivant représente une boucle de base pour une simulation temps réel typique.

```
Tant que la simulation est en progression

        Attendre jusqu'à W2S(temps d'horloge) >= temps simulé
        Calculer les états du système à ce pas de temps
        Avancer le temps simulé au pas de temps suivant.
```

Figure 3.2 Boucle de base pour une simulation temps réel dirigée par le temps

- Exécution dirigée par les évènements

L’idée de base des simulations à évènements discrets est la mise à jour des variables d’état seulement si un événement se produit. Un événement est une abstraction, utilisée dans les simulations, pour modéliser une action instantanée dans un système physique. Chaque événement a une estampille qui indique le moment dans le temps de simulation où l’événement se produit.

Une estampille est une *date* attribuée à un message. Cette date peut se réduire à un simple numéro d'ordre. On parle par exemple d'estampille dans les systèmes distribués asynchrones : les différents processus peuvent construire une horloge logique en étiquetant chacun de leurs messages par un nombre entier appelé *estampille*. Dans ce sens, le terme anglais équivalent est *timestamp*.

Un événement se produit en général lors d’un changement d’une ou plusieurs variables d’état définies dans la simulation. Dans une simulation dirigée par les évènements, le temps simulé avance par saut de durée variable d’un événement au prochain événement à traiter.

Une simulation dirigée par les évènements peut émuler une simulation dirigée par le temps en définissant les évènements qui se produiront à chaque pas de temps fixe. Pour cela, le pas de simulation utilisé doit être égal au plus petit commun diviseur de toutes les estampilles des évènements. Ceci va garantir qu’aucun événement ne se produira entre deux pas de temps. Ceci peut être inefficace en pratique puisqu’il n’y aura aucun calcul à effectuer pour plusieurs pas de simulation.

Une simulation dirigée par les évènements peut être une simulation temps réel. En effet, l’idée est de retenir le temps simulé d’avancer au temps de l’estampille de l’événement jusqu’à ce que le temps d’horloge avance au temps de l’estampille de cet événement. Si l’estampille du prochain événement est T_s, le temps simulé n’avance à T_s que si W2S (T_w) atteint T_s où T_w est la valeur courante de l’horloge.

Les limitations de ce type de simulation peut être liées à la performance dans le cas d'un grand nombre d'évènements car ces derniers ne sont pas regroupés et traités en paquet.

3.4 Simulation distribuée à évènements discrets

Nous avons présenté jusqu'à présent les éléments de base de la simulation séquentielle à évènements discrets. L'inconvénient majeur de ces simulations est leur limite due à la puissance de calcul disponible sur la machine d'exécution. La simulation distribuée est apparue pour répondre à plusieurs attentes ; parmi elles l'utilisation au mieux de la puissance des ordinateurs afin de traiter des modèles et des exécutions plus complexes.

Nous avons vu précédemment, qu'un système physique est perçu comme un ensemble de processus physiques qui interagissent d'une certaine manière. Chaque processus physique est modélisé par un processus logique. Les interactions entre les différents processus physiques sont modélisées par des échanges de messages estampillés (datés) entre les processus logiques. Le calcul effectué par chaque processus logique est un ensemble de calculs d'évènements, où chaque calcul doit modifier les variables d'état et/ou générer des évènements locaux ou des évènements destinés à d'autres processus logiques.

A première vue, ce modèle semble parfaitement adapté pour une exécution parallèle ou distribuée. Malheureusement, ce n'est pas le cas. Cette distribution nécessite une synchronisation entre processus logiques afin que l'exécution demeure cohérente. En effet, l'échec du traitement des évènements selon leurs estampilles peut causer l'envoi d'un événement qui possède une estampille de valeur inférieure au temps logique du processus logique destinataire. Le traitement des évènements en désordre évoque le problème de *causalité* et le problème d'assurer le traitement des évènements selon l'ordre de leurs estampilles évoque le problème de *synchronisation*. Un exemple proposé dans **[Fujimoto, 2000]** pour expliquer le problème de causalité est le suivant :

Trois processus s'exécutent chacun suivant son temps simulé, mais à un instant donné du temps réel, tous ne sont pas au même point du temps simulé. Le premier processus déclenche un événement T qui représente le tir d'un projectile. Un message est envoyé alors aux deux autres processus logiques (2 et 3). La réception du message est marquée par (T). Le processus logique 3 reçoit le message avant le processus logique 2. En conséquence de la réception du message par le processus logique 3, une destruction de la cible gérée par ce dernier est exécutée. Un message est envoyé alors aux deux processus logiques 1 et 2. La réception de ce message est marquée par (D). L'événement D est détecté par le processus logique 2 avant qu'il détecte l'événement T. Pour le processus logique 2, la destruction de la cible a eu lieu avant le tir du projectile. Ce traitement en désordre des évènements induit un problème de causalité.

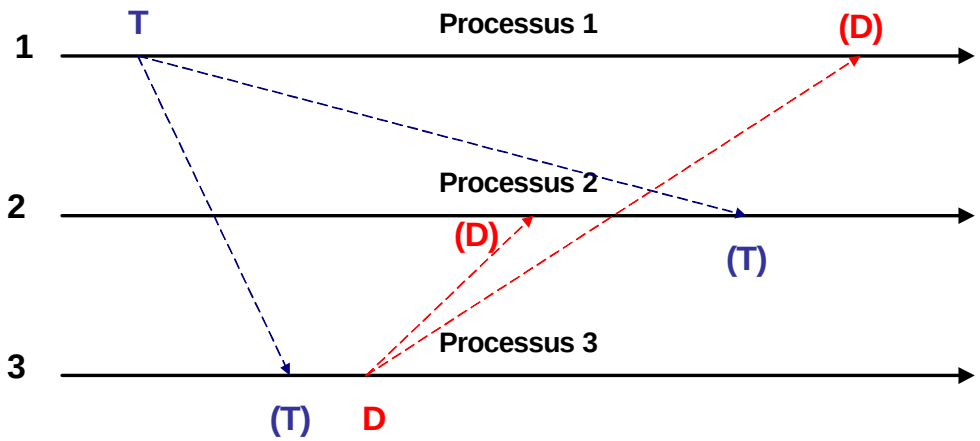


Figure 3.3 Problème de causalité

Une exécution distribuée doit s'assurer de reproduire les relations de causalité temporelles existantes dans l'exécution séquentielle équivalente.

Le problème de synchronisation

Le problème de synchronisation a été un point fondamental de la plupart des travaux de recherche à ce jour concernant l'exécution des simulations analytiques à évènements discrets sur des ordinateurs distribués ou parallèles. Si on adhère à la

contrainte de causalité locale on peut s'assurer que des erreurs de causalité ne se produiront pas.

Contrainte de causalité locale : une simulation à évènements discrets composée par des processus logiques qui interagissent uniquement en échangeant des messages datés obéit à la contrainte de causalité localement si et seulement si chaque processus logique traite les évènements en un ordre non décroissant.

L'adhésion à cette contrainte est suffisante, mais pas toujours nécessaire, pour garantir qu'il n'y aura pas des erreurs de causalité. En d'autres termes, la violation de cette règle n'induit pas forcément des erreurs de simulation. En effet, on peut avoir dans un seul processus logique deux évènements qui peuvent être indépendants l'un de l'autre. Dans ce cas, leur traitement hors de cette contrainte de causalité locale ne conduit pas à des erreurs de causalité.

Le défi des simulations parallèles à évènements discrets est d'exécuter des processus logique d'une façon concurrente et procéder à corriger les erreurs de la simulation.

Dans la partie Annexes, nous présentons quelques algorithmes de base pour fixer le problème de synchronisation et de causalité. Ces algorithmes représentent deux approches dites conservative et optimiste. Historiquement, les premiers algorithmes de synchronisation se sont basés sur une approche conservative.

3.5 Conclusion

Nous avons présenté dans ce chapitre un ensemble des notions de base de la simulation. Nous avons mis l'accent en particulier sur la simulation distribuée à évènements discrets. Quelques techniques et algorithmes d'exécution de ces simulations distribuées ont été présentés. Nous avons aussi présenté des méthodes pour synchroniser des simulations avec le temps réel. Ces méthodes nous seront utiles plus tard dans ce travail pour aboutir à la fin à une simulation distribuée temps réel.

4. Chapitre 4 Les architectures pour la simulation distribuée

4.1 Introduction

La simulation est un des moyens qui permet de maîtriser la complexité des systèmes mécatroniques, elle nous permet de :

- Simuler pour comprendre le fonctionnement d'un système mécatronique afin de trouver les lois d'évolution ; le rôle du simulateur ici est de permettre la mise au point d'un modèle.
- Simuler pour apprendre à manipuler un système, le rôle du simulateur ici est de reproduire un phénomène.
- Simuler pour évaluer d'une manière prédictive le comportement d'un système mécatronique et de prévoir ses réactions lors de son interaction avec le monde extérieur. Le rôle du simulateur ici est celui d'un outil de conception et de validation du système lui même.

Des logiciels de simulation ont été conçus afin de simplifier la tâche du concepteur, et de lui permettre de se consacrer à la partie métier du simulateur. Nous nous intéressons aux architectures qui permettent aux différents simulateurs de différentes disciplines d'inter opérer pour pouvoir simuler des systèmes mécatroniques pluridisciplinaires dépendants de différents domaines physiques.
Le besoin de simuler est apparu très tôt dans le domaine militaire. En effet, la simulation rend désormais possible :

- l'entraînement de " troupes " numériquement importantes dans un environnement réaliste jamais atteint auparavant ;
- la planification de missions opérationnelles ;
- le développement de nouvelles stratégies et tactiques ;
- le test de l'efficacité de nouveaux systèmes à une étape avancée de leur développement.

Pour pouvoir réaliser ces larges mondes virtuels, il était nécessaire de penser à une infrastructure ou architecture normalisée qui pourra rendre différentes plates-formes de simulation inter opérables.

En 1989, un groupe dépendant de la Défense américaine organisa une série de réunions dans le but de créer des architectures pour la simulation distribuée. Ces réunions ont abouti à la mise en place de la norme DIS (Distributed Interactive Simulation). Pendant de nombreuses années, plusieurs simulateurs ont été développés pouvant s'interfacer avec le protocole DIS. Malgré son grand succès, la norme DIS manque d'extensibilité et de réutilisation. Ainsi, une nouvelle architecture, HLA, a été développée pour combler à ce manque.

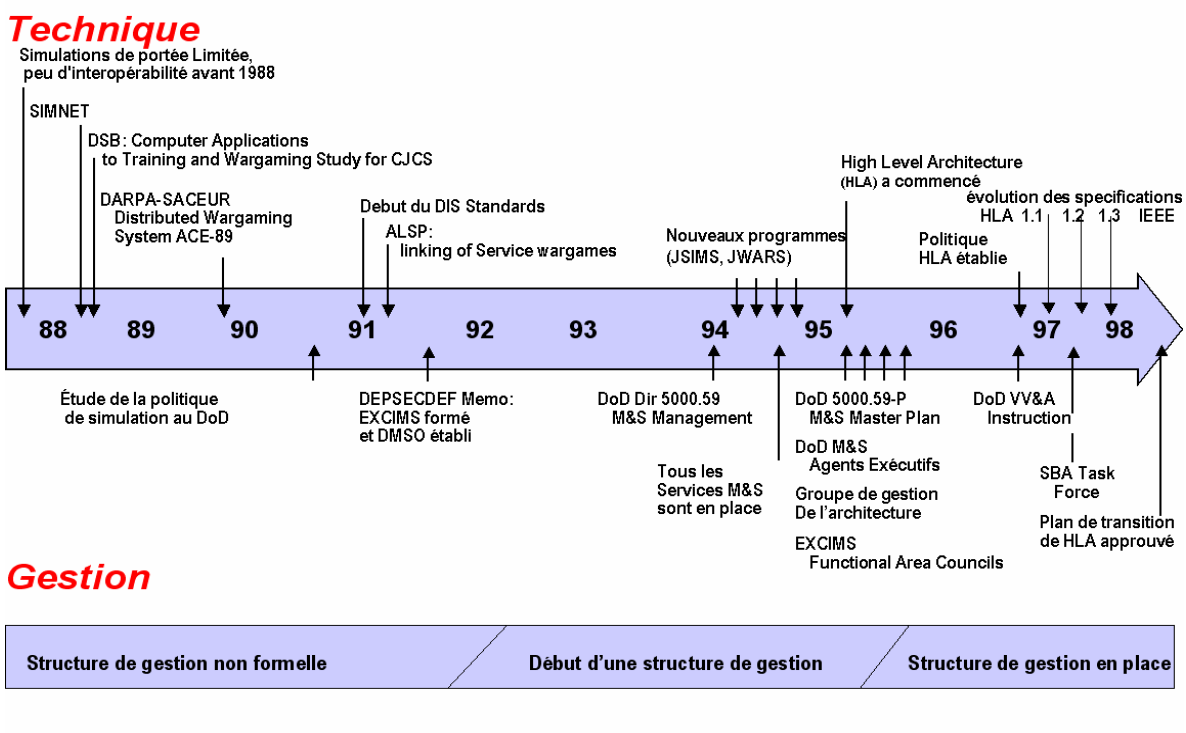


Figure 4.1 Standards de simulation distribuée

Les trois architectures que nous allons présenter, dans l'ordre chronologique, dans ce chapitre sont d'origine militaire. Nous effectuerons par la suite une comparaison entre la norme DIS, ALSP (Aggregate Level Simulation Protocol) et la norme HLA, nous montrerons les avantages de HLA par rapport à DIS et ALSP. Nous présenterons, par la suite, une brève comparaison entre les deux approches HLA et

CORBA. La norme CORBA s'oriente de plus en plus vers le temps réel en intégrant des notions de qualité de service dans les échanges, et la norme HLA a une orientation plus poussée vers la simulation. Cette dernière norme est mieux adaptée pour notre type d'application.

De 1983 à 1990, le projet SIMNET (SIMulation NETworking) de l'ARPA a mené à la réalisation d'un protocole de communication utilisé pour échanger des données entre différents simulateurs de véhicules militaires, placés dans un environnement virtuel commun **[Miller, 1995]**. Ces travaux se sont poursuivis jusqu'à la mise en place du protocole DIS.

4.2 DIS

DIS (Distributed Interactive Simulation) est une architecture pour la simulation distribuée. Elle permet de relier des simulations afin de créer un monde virtuel. DIS a été établi dans le but de définir une norme pour les simulations militaires. Dans DIS, le monde est modélisé comme un ensemble d'entités qui interagissent les unes avec les autres par le biais d'événements, qu'elles créent et qui peuvent être perçus par les autres entités. Pour cela, des échanges de données s'effectuent de manière asynchrone et permettent de maintenir un environnement synthétique consistant. En effet, ces échanges se basent sur des protocoles qui permettent d'envoyer des messages aux entités et des évènements sur un réseau reliant les plate-formes de simulation. DIS ne sert pas seulement à interconnecter plusieurs plates-formes de simulation mais il peut aussi influencer sur les applications de ces simulations en définissant des entités par exemple.

4.2.1 Architecture de DIS

La mise en place du standard DIS a pour but d'interconnecter plusieurs simulations distribuées. Cette connectivité est basée sur le fait que les différentes simulations disposent d'une représentation commune de l'environnement fixe. Les activités dynamiques et les évènements sont échangés entre les simulations à travers

les PDUs (Protocol Data Units). L'architecture de DIS, représentée dans la figure suivante, se décompose en 4 grandes parties qui seront présentées ci-après.

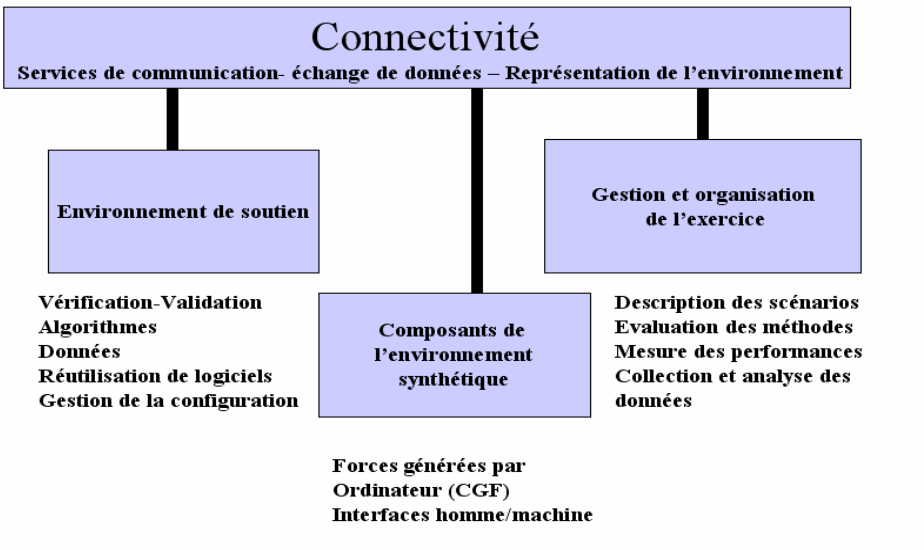


Figure 4.2 Structure de l'architecture DIS

Connectivité

La connectivité représente l'ensemble des mécanismes d'interconnexion entre les différentes applications de simulation. Les informations qui transitent entre les plate-formes de simulation sont échangées à travers des messages formatés appelés PDU (Protocol Data Unit).

Quatre domaines ont été étudiés pour la normalisation : les protocoles d'échange, l'architecture de communication, la sécurité et l'environnement.

- **Les protocoles d'échange de données** :

Cet aspect concerne l'identification des données échangées et les conditions de leurs utilisations :

- Identification des éléments de données à échanger;
- Représentation commune des données échangées ;

- Assemblage de ces éléments donnés au sein de messages formatés appelés PDU (Protocol Data Units).
- Détermination des circonstances d'émission de ces PDUs.
- Détermination du comportement à adopter suite à leur réception ;
- Les algorithmes critiques qui doivent absolument exister dans chaque système (ex : le Dead Reckoning[13]).

Toutes ces définitions ont été regroupées et décrites en détails dans un la norme IEEE 1278.1 **[IEEE 1278.1].**

- **Architecture de communication** :

Les PDUs définissent l'information échangée entre les différentes plates-formes de simulation. La normalisation des communications touche :

- Les procédures d'adressage (unicast, multicast, broadcast).
- Les contraintes de fiabilité (meilleur effort, fiable).
- Le choix des protocoles et des fonctions de communication.
- Des conduites en matière de bande passante/débit etc...

L'architecture est décrite en détail dans la norme IEEE 1278.2 *Standard for Distributed Interactive Simulation – Communication Architecture Requirements*.

- **La sécurité :**

Il s'agit de la protection des données sensibles échangées. Les quatre points abordés sont :

- La politique de sécurité DIS.
- Document d'instructions proposant une méthodologie de sécurisation.
- Accréditions.

[13] Mécanisme recommandé par la norme DIS et qui signifie « calcul par déduction ». Le but de ce mécanisme est de minimiser le nombre de messages échangés entre les sites, prédire le mouvement des objets des sites distants et d'utiliser uniquement les paramètres connus localement.

- Des exigences de performances en terme de sécurité.

- **L’environnement :**

Il s’agit de fournir une représentation intégrée et complète des composants du monde réel. Cette partie traite les points suivants :

- Identification de sources communes ;
- Création d’une norme ;
- Création d’une base de donnée pour y stocker les données communes ;
- Distribution des données vers les simulateurs ;
- Identification des besoins des exercices

Les composants de l’environnement synthétique

L'environnement synthétisé doit fournir une représentation intégrée et complète des composants du monde réel. Deux conditions doivent êtres prises en compte :

- Fidélité de la représentation de l’environnement.
- La corrélation de la représentation d'un système à l'autre.

Les composants de l’environnement synthétique sont les forces générées par l’ordinateur CGF (Computer Generated Forces), afin de représenter un nombre suffisant de combattants, les interfaces homme-machine…etc

Gestion/Supervision

Afin d’améliorer l’interopérabilité des simulateurs, les procédures d’organisation doivent être normalisées. Ce domaine de normalisation comporte trois aspects :

- *Gestion de l'exercice en cours*

Ceci concerne la normalisation des PDUs de contrôle (arrêt, réinitialisation..) et la surveillance de l’exercice.

- Gestion du réseau

Ceci a pour but de faciliter la gestion du réseau de communication entre les différentes plate-formes de simulation.

- Gestion de la sécurité

Ceci concerne spécialement les moyens d'établissement et vérification des niveaux de sécurité requis.

4.2.2 PDU (Protocol Data Unit)

Le PDU représente le cœur de l’architecture DIS. Dans cette dernière on trouve 27 PDUs, représentés dans le tableau ci après. Parmi ces PDUs, quatre sont souvent utilisés pour interagir entre les nœuds (état de l’entité, tir, détonation et collision).

Nom du PDU	Explication
Acknowledge	Acquitte la réception des PDUs Start/Resume, Stop/Freeze, Create Entity et Remove Entity.
Action Request	Envoie une requête à une entité pour effectuer une action.
Action Response	Acquitte la réception d'un PDU action Request.
Collision	Quand une entité détecte une collision avec une autre entité, elle produit ce PDU.
Comment	Permet de transmettre un message arbitraire.
Create Entity	L'information sur la création d'une nouvelle entité est effectuée avec ce PDU.
Data	La réponse d'un PDU Data Query ou Set Data est transmise avec ce PDU.
Data Query	Effectue une requête à une entité pour obtenir des données.
Designator	Opération désignant une opération à une entité.
Detonation	Transmet la détonation ou l'impact d'une munition.
Electromagnetic Emission	Transmet l'information sur l'émission électronique et la contre-mesure.
Entity State	Comporte les informations concernant l'état d'une entité.
Event Report	Permet à une entité gérée de rapporter l'occurrence d'un événement important au gestionnaire de la simulation. (dommage, mort, essence épuisée)
Fire	Communique le tir d'une armée.
Receiver	Communique l'état du récepteur. (allumé éteint et/ou en réception)
Remove Entity	Informe du retrait d'une entité de la simulation.
Repair Complete	Notifier à l'émetteur d'un Service Request que la réparation est terminée.
Repair Response	Acquitte la réception d'un message Repair Complete.
Resupply Cancel	Annule un service, soit par le récepteur, soit par le fournisseur.
Resupply Offer	L'offre de ravitaillement est transmise par ce PDU.
Resupply Received	La réception d'un ravitaillement est transmise par ce PDU.
Service Request	Permet d'effectuer une requête pour obtenir de la logistique.
Set Data	Initialise ou change l'information de l'état interne.
Signal	Transmet la voix, le son et d'autres données.
Start/Resume	Communique le départ et la reprise d'un exercice.
Sto/Freeze	Communique l'arrêt ou le gel d'une entité ou de l'exercice.
Transmitter	Fournit des informations détaillées sur un émetteur radio.

Tableau : Ensemble des PDUs du protocole DIS.

Sa structure est présentée dans le tableau suivant. Le PDU fournit une structure composée par plusieurs champs internes. Ces champs sont utilisés pour transmettre les données d'une simulation à une autre. Les données à l'intérieur de ces champs sont définies comme des multiples de 8 bits comme le montre le tableau ci-après.

L'entête du PDU fournit des informations sur le contenu des données à tous les nœuds de réception. Ces informations contiennent la version du protocole en cours d'utilisation, l'identifiant de l'exercice de simulation, le type du PDU envoyé, le temps de génération des données contenues dans le PDU et la longueur du flux de données en cours d'envoi.

Les champs qui viennent après l'entête sont Entity ID, Force ID, Number of articulated parts, Entity type et Alternate Entity type. Ces champs fournissent des informations sur l'exercice de la simulation et sur l'entité qui a généré les données contenues dans le PDU. Le groupe des champs suivant fournit des informations sur l'état courant de l'entité. Ces données fournissent la position, la fréquence de mouvement, orientation et des paramètres utilisés par l'algorithme « Dead Rekoning ».

Le dernier groupe de champs permet le transfert d'information sur les parties articulées qui sont attachées à la plate-forme représentée par l'entité. Ces champs nécessitent 128 bits pour chaque partie articulée. Ainsi, la taille totale du ESPDU est égale à (1152 + 128 n) avec n égal au nombre de paramètres d'articulations.

Ces choix techniques risquent malheureusement d'encombrer le réseau. En effet, le PDU Entity State est très volumineux et comprend des données qui ne changent pas ou très peu. De plus, ce PDU est le plus utilisé pour la communication entre les simulateurs comme le montre la démonstration d'interopérabilité de DIS effectuée en 1993 lors de la conférence annuelle I/ITSEC **[Pullen, 1995]** . Tout le trafic réseau a été enregistré et analysé. L'analyse des différents PDUs montre que 96% du trafic totale consiste en des ESPDUs. La répartition des PDUs restants (4%) est la suivante :

- 4% tir
- 4% détonation
- 1% collision
- 0% logistique
- 0% gestion de la simulation
- 39% émission
- 50% transmetteur
- 0% signal
- 2% acoustique
- 1% observateur (PDU à l'écoute)
- 0% autres

Taille des champs (bits)	Les champs du PDU Entity State		
96	PDU Header	Protocol Version	8-bit enumeration
		Exercice ID	8-bit unsigned integer
		PDU Type	8-bit enumeration
		Protocol Family	8-bit enumeration
		Time Stamp	32-bit unsigned integer
		Length	16-bit unsigned integer
		Padding	16-bit unused
48	Entity ID	Site	16-bit unsigned integer
		Application	16-bit unsigned integer
		Entity	16-bit unsigned integer
8	Force ID		8-bit enumeration
8	# des paramètres d'articulation		8-bit unsigned integer
64	Entity Type	Entity kind	8-bit enumeration
		Domain	8-bit enumeration
		Country	8-bit enumeration
		Category	8-bit enumeration
		Subcategory	8-bit enumeration
		Specific	8-bit enumeration
		Extra	8-bit enumeration
		Entity kind	8-bit enumeration
		Domain	8-bit enumeration
		Country	8-bit enumeration

64	**Alternative Entity Type**	**Category**	**8-bit enumeration**
		Subcategory	**8-bit enumeration**
		Specific	**8-bit enumeration**
		Extra	**8-bit enumeration**
96	**Entity linear velocity**	**x-Component**	**32-bit floating point**
		Y-Component	**32-bit floating point**
		Z-Component	**32-bit floating point**
192	**Entity Location**	**X-Component**	**64-bit floating point**
		Y-Component	**64-bit floating point**
		Z-Component	**64-bit floating point**
96	**Entity Orientation**	**Psi**	**32-bit floating point**
		Theta	**32-bit floating point**
		Phi	**32-bit floating point**
32	**Apparence de l'entité**		**32-bit record of enumerations**
320	**Dead Reckoning parameters**	**Dead Reckoning algorithm**	**8-bit enumeration**
		Other parameters	**120 bit unused**
		Entity linear acceleration	**3*32-bit floating point**
		Entity angular velocity	**3*32-bit floating point**
96	**Entity marking**	**Character set**	**8-bit enumeration**
		String record	**11 8-bit unsigned integer**
32	**Capacités**		**32 Boolean fields**
N° 128	**Articulation parameters**	**Parameter type designator**	**8-bit enumeration**
		Change	**8-bit unsigned integer**
		ID-attached to	**16-bit unsigned integer**
		Parameter type	**32-bit parameter type record**
		Parameter value	**64-bit**
Taille totale ESPDU = (1152 + 128n) avec n = nombre de paramètres d'articulations			

Tableau : Entity State PDU **[Hofer, 1995]**

Comme on peut le constater, DIS a été créé pour des raisons purement militaires. Ceci est un inconvénient majeur pour son utilisation dans d'autres domaines.

4.3 ALSP

ALSP est un protocole de simulation de niveau agrégé, décidé par la DARPA en 1990, dont l'objectif est de permettre l'interconnexion plusieurs simulations constructives non temps réel déjà existantes. Contrairement à DIS, orienté entraînement, où l'homme agit directement dans la simulation, ALSP permet l'étude de scénarios exécutés en temps coordonné.

ALSP propose une partie logicielle, une interface pour l'échange de messages et un ensemble de simulations déjà existantes. Le principe de fonctionnement de ALSP repose sur ces éléments :

- Les acteurs qui représentent les simulations.
- Une confédération qui représente l'ensemble des acteurs interconnectés.
- Les objets qui représentent les entités de la simulation.
- Un ensemble d'attributs qui décrivent ces objets.
- Les fantômes qui représentent les objets vus comme locaux mais contrôlés par un autre acteur de la confédération.

L'infrastructure AIS (ALSP Infrastructure Software) repose sur la figure suivante :

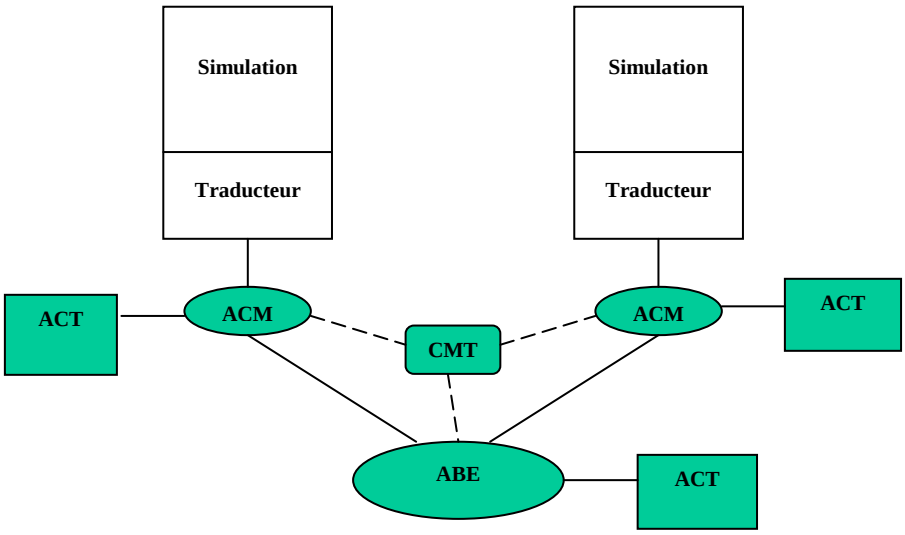


Figure 4.3 Architecture ALSP

Cette figure représente une confédération globale ALSP. Les composants qui forment cette architecture sont les suivants **[Site Web ALSP]**:

- Les ACM(ALSP Common Module) : Ces modules permettent de gérer le temps local de l'acteur ainsi que les objets.
- ABE (A Broadcast Emulator): module chargé de la communication entre les ACM.
- Traducteur : il assure la liaison entre acteur et confédération en convertissant les données de la simulation au format proposé par ALSP.
- Les ACT (ALSP Control Terminal) : ils sont attachés à un seul ACM ou ABE, et permettent notamment de visualiser ou modifier le statut de celui ci.

Cette architecture est bien différente de celle de DIS et annonce déjà celle de HLA.

4.4 HLA

HLA (High Level Architecture) constitue une architecture de haut niveau pour la simulation distribuée, proposée par le DMSO (Defense Modeling and Simulation Office) pour soutenir la réutilisation et l'interopérabilité à travers un grand nombre de différents types de simulations développées et maintenues par le DoD. La définition de base de HLA a été officielle le 21 août 1996. Elle a été approuvée par le secrétaire de la défense pour l'acquisition et la technologie (USD(A&T)) comme architecture technique standard pour toutes les simulations de DoD le 10 septembre 1996. Le HLA a été adopté comme service pour les systèmes répartis 1,0 de simulation par le groupe de gestion d'objet (OMG) en novembre 1998 et mis à jour en 2001 pour refléter les changements résultant de l'étalonnage commercial des spécifications sous l'IEEE. Le HLA a été approuvé comme norme ouverte par l'institut des ingénieurs électriques et électroniciens (IEEE) - la norme 1516 d' IEEE - en septembre 2000. En novembre 2000 les services et le personnel commun ont signé le mémorandum identifiant le HLA comme architecture préférée pour l'interopérabilité de simulation dans le DoD. **[Site Web DMSO-HLA]**

Ces deux spécifications (HLA 1.3 et IEEE 1516) qui coexistent sont incompatibles, en raison des importantes modifications apportées. Les principes initiaux de HLA sont conservés, certains services sont ajoutés, d'autres sont simplifiés. Une synthèse des différences entre HLA 1.3 et HLA IEEE 1516 est présenté dans **[Lightner, 2000]** . Nous nous intéressons principalement dans la suite de ce chapitre à la version IEEE 1516 de HLA.

4.4.1 Architecture générale de HLA

HLA est fondée sur l'hypothèse qu'aucune simulation ne peut satisfaire tous les besoins. L'objectif de cette architecture est triple:

- Faciliter la réutilisation de simulateurs élémentaires
- Faciliter l'interopérabilité entre simulateurs distribués
- Réduire les coûts de modélisation et de simulation

Une simulation globale HLA est appelée fédération, et un simulateur y participant est appelé fédéré. Un fédéré peut aussi recevoir des données extérieures correspondant à un objet réel ou un humain. Un autre composant essentiel d'une simulation HLA qui s'ajoute est le RTI (Run-time Infrastructure). Il constitue une implémentation informatique des spécifications de l'interface de programmation. Il s'agit donc d'un ensemble de logiciels offrant des services communs à un ensemble de fédérés, éventuellement répartis, et participant à une même fédération. Cet ensemble se comporte comme un système d'exploitation distribué. Tout échange de données entre des fédérés doit passer par le RTI.

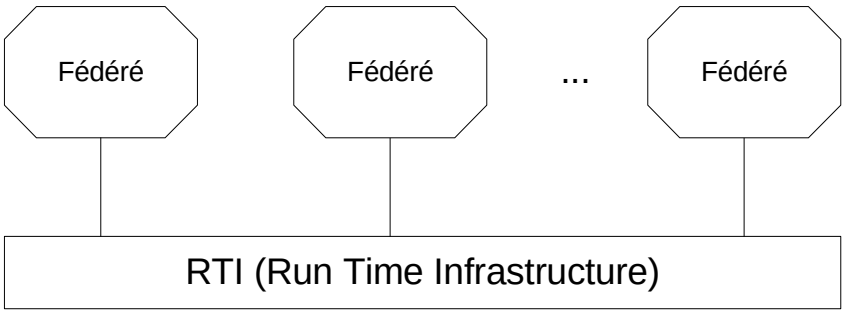


Figure 4.4 Fédérés et RTI d'une simulation HLA

Quelques définitions sont nécessaires pour la suite de cette présentation de HLA :

- Fédération : Une fédération est un ensemble de fédérés ayant un modèle objet commun, c'est la représentation d'un ensemble de simulateurs inter-opérant.

- Fédéré : Un fédéré est un membre d'une fédération HLA, c'est la représentation de chaque simulateur élémentaire.

- RTI : Le RTI (Run-Time Infrastructure) constitue une implémentation informatique des spécifications d'interface HLA. Il s'agit d'un processus informatique assurant les communications entre les fédérés d'une même fédération, en offrant les services de HLA au travers d'une API[14].

La communication se décompose en deux parties : le RTIambassador et le FederateAmbassador. Le RTIambassador est une interface permettant au fédéré d'envoyer des messages au RTI (demandes ou transmissions d'informations). Son API est entièrement définie dans différents langages et une spécification décrit comment l'implémentation doit se comporter lorsqu'un message est reçu. Le FederateAmbassador dispose d'une interface définie et normalisée ; spécifique à l'application et son implémentation est à la charge du développeur du fédéré.

4.4.2 Les spécifications de HLA

L'architecture HLA est décrite par des spécifications composées : **[IEEE 1516]**

- d'un ensemble de règles définissant les responsabilités des fédérés et de la fédération.
- d'une modélisation orientée objet des fédérés (OMT)
- des spécifications de l'interface de programmation (API)

[14] Application Program Interface

Les règles

Le département américain a défini dix règles qui régissent toutes les simulations HLA. Leur respect est indispensable au bon déroulement d'une simulation. Les cinq premières concernent le fonctionnement global des fédérations.

Règle 1 : Les fédérations doivent avoir un modèle objet de fédération (FOM : Federation Object Model) conforme à l'OMT de HLA. Le FOM doit décrire toutes les données échangées au cours de l'exécution d'une fédération, et doit indiquer les conditions de ces échanges.

Règle 2 : Les représentations d'objets associés à la simulation doivent être dans les fédérés et non dans le RTI.

Règle 3 : Au cours de l'exécution de la fédération, les données décrites dans le FOM ne peuvent faire l'objet d'échanges entre fédérés que par l'intermédiaire du RTI.

Règle 4 : Au cours de l'exécution de la fédération, les fédérés doivent interagir avec le RTI conformément à la spécification de l'interface HLA.

Règle 5 : Au cours de l'exécution de la fédération, l'attribut d'une instance d'un objet ne peut être la propriété que d'un seul fédéré à n'importe quel instant.

Les cinq dernières règles s'intéressent aux fédérés.

Règle 6 : Les fédérés doivent avoir un modèle objet de simulation (SOM : Simulation Object Model) conforme à l'OMT de HLA. Le SOM correspond aux informations qui pourront êtres rendues publiques au cours de l'exécution de la fédération.

Règle 7 : Les fédérés doivent mettre à jour et prendre en compte des modifications d'attributs des objets, ainsi que envoyer et/ou recevoir des interactions conformément à la spécification de leur SOM.

Règle 8 : Les fédérés doivent être capables de transférer ou d'accepter la propriété des attributs dynamiquement pendant l'exécution d'une fédération, conformément aux spécifications indiquées dans leurs SOM.

Règle 9 : Les fédérés doivent pouvoir faire varier les conditions sous lesquelles ils fournissent des mises à jours des attributs des objets, conformément aux spécifications de leur SOM.

Règle 10 : Les fédérés doivent pouvoir être capables de contrôler le temps local d'une manière qui leur permettra de coordonner l'échange de données avec les autres membres de la fédération.

À ce stade nous ferons une vue d'ensemble rapide pour avoir une idée de la portée des règles. Les règles de fédération établissent les règles de base pour créer une fédération, y compris des conditions de documentation (règle 1), représentation d'objet (la règle 2), échange de données (règle 3), conditions d'interfaçage (règle 4) et propriété d'attributs (règle 5). Les règles des fédérés couvrent la documentation (règle 6), commande et transfert d'attributs appropriés d'objet (règles 7, 8 et 9), et gestion du temps (règle 10).

Le modèle objet : OMT[15]

La réutilisabilité et l'interopérabilité exigent que tous les objets et interactions contrôlées par un fédéré doivent être indiqués en détail et avec un format commun. Le modèle d'objet (OMT) fournit une norme pour documenter l'information de modèle

[15] Object Model Template

d'objet de HLA. L'OMT décrit le formalisme qui doit être employé pour définir le FOM d'une fédération, ou le SOM d'un fédéré.

D'après les règles 1 et 6, on doit avoir un OMT qui décrit tous les objets et toutes les interactions. La spécification d'un format standard pour l'OMT est une partie du standard HLA.

Règle 1 : Les fédérations doivent avoir un modèle objet de fédération (FOM : Federation Object Model) conforme à l'OMT de HLA. Le FOM doit décrire toutes les données échangées au cours de l'exécution d'une fédération, et doit indiquer les conditions de ces échanges.

Règle 6 : Les fédérés doivent avoir un modèle objet de simulation (SOM : Simulation Object Model) conforme à l'OMT de HLA. Le SOM correspond aux informations qui pourront être rendues publiques au cours de l'exécution de la fédération.

Principes fondamentaux de l'OMT

Un pattern, pour spécifier les modèles objets de HLA, est un composant essentiel de HLA pour les raisons suivantes :

- Il fournit un mécanisme commun compréhensible pour spécifier les échanges de données et la coordination générale entre les membres d'une fédération.
- Il fournit un mécanisme commun et standard pour décrire les capacités des membres d'une fédération.
- Il facilite la conception et l'application d'un ensemble d'outils pour le développement des modèles objets de HLA.

FOM[16]

Les modèles objets HLA peuvent être utilisés pour décrire un membre d'une fédération (fédéré) ou pour décrire un ensemble de fédérés qui interagissent

[16] Federation Object Model

(fédération) en créant un FOM. La première fonction du FOM est de spécifier, dans un format commun et standard, la nature des données échangées entre les fédérés. Ces données incluent une énumération de toutes les classes d'objets et d'interactions appartenant à la fédération, ainsi qu'une spécification des attributs ou paramètres caractérisant ces classes.

SOM[17]

Le SOM de HLA est une spécification des capacités intrinsèques fournies aux fédérations HLA par une simulation individuelle. Les formats de l'OMT qu'on va décrire sont applicables aux FOMs et aux SOMs. Les SOMs sont caractérisés par leurs objets, attributs, interactions et paramètres. Le bénéfice de l'utilisation commune des formats OMT pour FOMs et SOMs est la référence commune pour décrire les modèles objets dans la commune HLA.

LES MODELES OBJET

- Modèle Objet de Fédération (FOM) :
  - Un par fédération
  - Présente toutes les informations partagées (Objets, interactions…)
- Modèle Objet de Simulation (SOM) :
  - Un par fédéré
  - Décrit les caractéristiques d'un fédéré
  - Présente les objets et les interactions qui peuvent être utilisés extérieurement.
- Modèle objet de contrôle (MOM) :
  - a une définition universelle
  - Identifie les objets et les interactions utilisées pour contrôler la fédération.

[17] Simulation Object Model

HLA et les concepts de l'orienté objet

Il faut noter les différences entre la définition des modèles objets sous HLA et la définition des modèles objets par les techniques de la conception orientée objet. Dans la conception orientée objet, un modèle objet fournit une description plus large d'un objet, incluant des détails complets des relations internes qui caractérisent l'objet. Les modèles objets HLA s'intéressent plus précisément à l'échange d'information entre fédéré et fédération.

Les composants de l'OMT

Les informations suivantes sur la composition de l'OMT s'appliquent également aux FOMs et aux SOMs. L'OMT spécifie que les modèles objets HLA doivent être documentés sous la forme d'un nombre de tables qui spécifient les informations au propos des classes d'objets, leurs attributs et leurs interactions. **[IEEE 1516.2]**

Les tables sont les suivantes :

- Table d'identification : Le but de cette table est de définir les caractéristiques de la fédération ou du fédéré.

Catégorie	Information
Nom	Mechatronic Simulation (SOM)
Version	
Date	
But	
Domaine d'application	
Sponsor	
Nom POC	
Détails POC	

- Table des classes objets : Cette table contient des informations sur la hiérarchie classe/sous classe des classes objets. Les classes objets sont classées selon leurs capacités de publication et de souscription (P/S).

Véhicule (N)	Avion (N)	A 380 (PS)
		B 747 (PS)
	Navire (N)	
	Char (N)	
Opérateur (N)	Pilote (N)	
	Contrôleur (N)	

P : la classe est publiée.

S : On s'abonne à la classe

N : ni l'un ni l'autre (classe abstraite)

Pour le SOM : {P, S, PS, N}

Pour le FOM : {PS, N}

- Table des classes d'interactions : Une interaction est une action déclenchée par un objet (ou un ensemble d'objets) qui peut avoir un effet sur des objets dans un autre fédéré. Ces interactions sont définies dans la table de structure des classes d'interactions. Les classes d'interactions sont classées selon leurs capacités. Quatre catégories de base désignées par les lettres I, S, R et N qui indiquent qu'un fédéré est capable de :

I (Initiates) : Initialiser et envoyer cette interaction.

S(Senses) : Souscrire à cette interaction et d'utiliser son information sans être nécessairement capable d'effectuer des changements aux objets (S).

R(Reacts) : Souscrire à cette interaction et de réagir en effectuant des changements aux objets (R).

N(Neither..) : Ni initialiser, ni s'inscrire et ni réagir à cette classe d'interaction (N).

Interaction Air-Air (ISR)	Missile1(ISR)
	Missile2(ISR)
	Missile3(ISR)
Interaction Sol-Air (ISR)	Missile1(ISR)
	Missile2(ISR)

- Table des attributs : Cette table permet de spécifier les fonctions des attributs des objets dans une fédération.

- Table des paramètres : Pour chaque classe d'interaction identifiée dans la table de structures des classes d'interactions, l'ensemble des paramètres associés avec cette classe d'interaction est spécifiée dans la table des paramètres. Les entrées de la table sont les classes d'interactions, leurs paramètres, et des informations spécifiant leurs types de donnée, cardinalités, unités, résolutions, précisions, conditions de précision et l'espace de routage.

- Table des routages : Cette table spécifie les espaces de routage des attributs des objets et des interactions dans une fédération.

- Un lexique FOM/SOM doit être crée : il définit tous les termes utilisés dans les tables.

La spécification OMT exige que les fédérés et les fédérations utilisent les sept composants de l'OMT bien que quelques tableaux peuvent être vides dans certains cas.

L'interface de programmation API

L'interface HLA définit un ensemble de services permettant l'échange d'informations entre fédérés participant à une même fédération. Ces informations

échangées doivent passer par le RTI conformément à la règle 3. Les services de l'interface HLA appartiennent donc à deux catégories :

- Les services à l'initiative des fédérés : Les services fournis par le RTI, auxquels peuvent accéder les fédérés.
- Les services à l'initiative du RTI : Les services fournis par les fédérés, auxquels peut accéder le RTI.

Ces services permettent la meilleure interopérabilité et réutilisabilité des composants de simulation développés. Indépendamment de ce sens d'utilisation, ces services sont organisés en six groupes **[IEEE 1516.1]**: gestion de la fédération, gestion des déclarations, gestion des objets, gestion du temps, gestion de la propriété et gestion de la distribution des données.

- Gestion de la fédération

Ce groupe gère l'accès d'un fédéré à une fédération (créer, rejoindre, quitter ou détruire une fédération), il gère aussi les mécanismes internes de la fédération (sauvegarde, restauration, synchronisation).

Le cycle de vie d'une fédération est décrit dans la figure 4.5. Le premier fédéré doit créer la fédération. Chaque fédéré voulant participer à cette fédération doit s'enregistrer en appelant le service *JoinFederationExecution*. A la fin de l'exécution, les fédérés voulant se retirer de la simulation doivent invoquer le service *ResignFederationExecution* et le dernier fédéré peut détruire la fédération.

Le service de gestion de la fédération est indispensable car les autres services ne peuvent être invoqués que si le fédéré a rejoint la fédération.

- Gestion des déclarations (publication/ souscription)

Ce groupe gère les déclarations : les déclarations par les fédérés des messages qu'ils sont susceptibles d'envoyer et l'information par le RTI des messages que les fédérés

sont susceptibles de recevoir. Le RTI se sert des déclarations faites par les fédérés pour gérer les flots de données.

Publication :

Chaque fédéré doit publier les classes d'objets et d'interactions qu'il prévoit de produire. Il peut ne publier qu'un sous ensemble des attributs de l'une de ses classes. Il doit dire de manière explicite tous les attributs qu'il veut publier. Un attribut spécifique PrivelegeToDelete peut être inclus. Il donne la propriété de l'instance au fédéré qui l'a créée.

Souscription :

Tout fédéré voulant des informations concernant certaines classes doivent l'indiquer de manière explicite. Pour les classes ils font appel à subscribeObjectClassAttributes. A partir de ce moment, le fédéré sera averti de toutes les créations d'instances de cette classe qui se produisent dans la fédération. Pour les classes d'interactions, les fédérés ne peuvent pas s'inscrire à un sous ensemble des paramètres. Ils doivent obligatoirement s'inscrire à tous les paramètres.

- Gestion des objets

Ce groupe gère les objets et les interactions. Une simulation évolue par la création des objets, leur mise à jour ou leur destruction et par les envois d'interactions. Ces évènements sont créés et observés par les fédérés à l'aide de ce groupe de service.

- Gestion du temps

Ce groupe gère l'avancement du temps en particulier des fédérés. Ce service est détaillé plus tard.

- Gestion de la propriété

Ce groupe gère la propriété des attributs d'objet. La règle numéro huit permet le transfert des attributs des objets, les services de ce groupe permettent de réaliser ces transferts.

- Gestion de la distribution des données

Ce groupe gère la distribution des données. Il permet d'affiner les déclarations du groupe de gestion des déclarations. Il permet de publier/souscrire à des objets suivant certaines valeurs de ses attributs. Pour échanger les données, le RTI gère des souscriptions et des publications à des classes d'objets. Ainsi, les mises à jour ne sont émises aux destinataires que s'ils ont explicitement souscrit ce type de classe. Ce routage effectué par le RTI permet de filtrer les messages et de n'envoyer les mises à jour qu'aux fédérés concernés. La gestion de la distribution des données permet d'étendre ce principe en ajoutant la gestion des zones (spaces). Ce service est optionnel et peut ne pas être mis dans une simulation. Il permet d'étendre le filtrage en n'émettant les messages qu'aux fédérés se trouvant dans la même zone que le message émis.

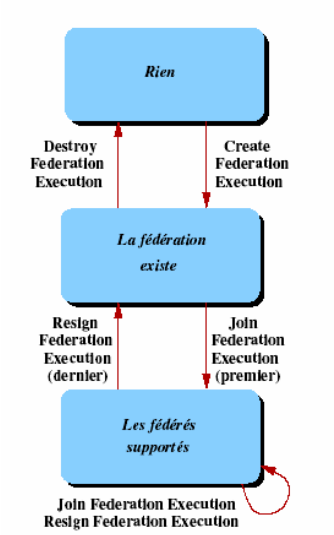


Figure 4.5 Le cycle de vie d'une simulation HLA

4.4.3 L'infrastructure de simulation RTI

Le RTI constitue une implémentation informatique des spécifications HLA, et en particulier de son interface de programmation (API). Il s'agit donc d'un ensemble de logiciels offrant des services communs à des fédérés répartis et participant à une même fédération.

La distribution RTI offerte par le DMSO est constituée de trois composantes:

- Le processus FedExec : Contrôle la fédération. Il permet au fédérés de joindre et de démissionner d'une fédération. Il facilite l'échange de données entre les fédérés d'une même fédération.

- L'exécutif RTIExec : Un processus global qui contrôle la création et la destruction des FedExecs.

- Les librairies libRTI du RTI.

Une vue logique des composants d'un RTI est présentée dans la figure suivante :

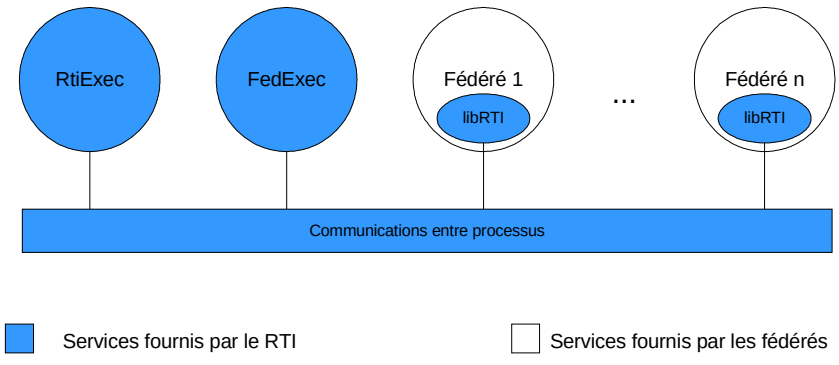


Figure 4.6 Vue logique des composants d'un RTI

4.4.4 Composants d'un fédéré HLA

Les composants d'un fédéré sont décrits dans la figure suivante :

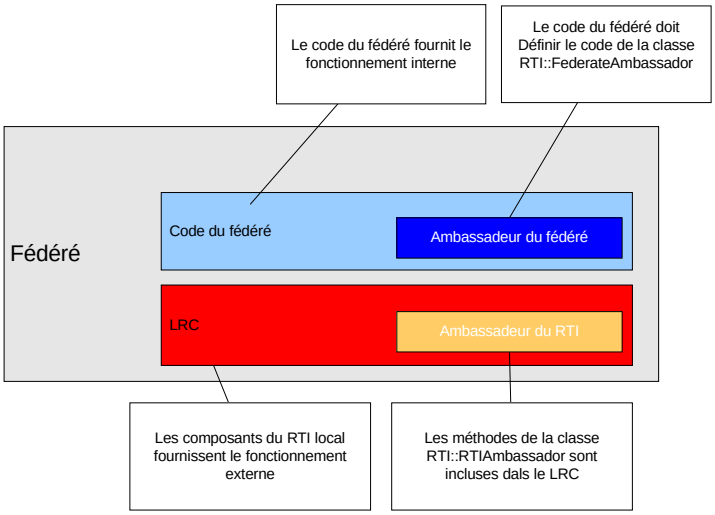


Figure 4.7 Composants d'un fédéré

- Ambassadeur RTI : le code du fédéré nécessite des services du RTI en appelant des fonctions membres de la classe RTI ::RTIAmbassador qui est contenue dans le libRTI.

- Ambassadeur du fédéré : le RTI envoie des messages et des réponses au code du fédéré en appelant des fonctions implémentées dans le fédéré. Ces fonctions sont appelées des callbacks, et elles sont implémentées comme une sous classe de la classe RTI ::FederateAmbassador.

4.4.5 Exécution d'une fédération

Les étapes dans le processus d'exécution d'une fédération :

1- Le premier fédéré à se lancer doit créer la fédération en lui donnant un nom.

2- Tous les fédérés suivant ainsi que le fédéré créateur doivent rejoindre la fédération.
3- Les fédérés désirant émettre des données publient les différents attributs des objets et des interactions qu'ils vont créer pendant la simulation.
4- Ces mêmes fédérés vont créer des instances d'objets déclarés et vont les enregistrer auprès du RTI.
5- Les fédérés désirant recevoir les informations des autres fédérés doivent souscrire auprès du RTI. Si des informations les concernent, le RTI leur fait découvrir ces instances.
6- Pendant la simulation ces deux étapes s'effectuent continuellement :

-Le fédéré envoie des mises à jour concernant ses propres instances.

-Le RTI transmet les différents mises à jour vers les fédérés ayant souscrit ce type d'objet.

7- Lorsqu'une instance disparaît de la simulation, le fédéré transmet l'information au RTI.
8- Lorsqu'un fédéré retire toutes ses instances, il peut se retirer de la simulation et en informe le RTI qui le retire de la liste des fédérés participants.
9- Le dernier fédéré de la simulation doit détruire l'exécution.

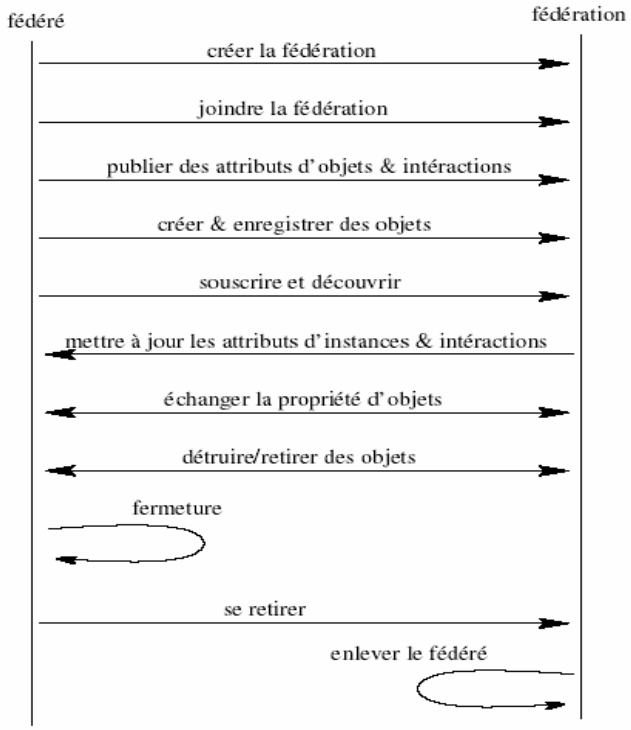


Figure 4.8 Exécution d'une fédération

4.4.6 Caractéristiques de l'architecture HLA

HLA suit trois styles d'architectures logicielles : **[Kuhl, 1999]**

- HLA possède des aspects d'architecture organisée en couches hiérarchiques : chaque couche propose des services à la couche supérieure, et inversement utilise les services de la couche inférieure. En effet, la couche RTI permet de proposer au concepteur de fédéré des mécanismes de simulation de haut niveau, et de cacher dans les couches inférieures des aspects de bas niveau (Protocoles de communications, TCP/IP…)

- HLA propose une architecture d'abstraction des données : la spécification impose une interface pour le RTI et une interface pour les fédérés. Derrière cette interface, le travail du fédéré est caché au RTI, et inversement le fonctionnement du RTI est caché aux fédérés.

- HLA est une architecture orientée événement : le RTI diffuse les données aux fédérés qui se sont enregistrés en conséquence. Ainsi un fédéré, après avoir indiqué les types d'événements qu'il souhaite recevoir, n'a pas à rechercher activement l'apparition de tels évènements : ses propres services seront directement appelés par le RTI.

4.5 Comparaison des architectures de simulation distribuée

4.5.1 Comparaison générale

Nous allons dans le tableau suivant comparer les quatre architectures de simulation distribuée d'après certains critères : SIMNET, ALSP, DIS, et HLA.

Domaines d'études	SIMNET	ALSP	DIS	HLA
Domaines d'application	Simulation de troupes en temps réel	Simulation constructive non temps réel	Simulation d'entraînement en temps réel	Tout type de simulation
Interopérabilité	Oui	Oui	Oui	Oui
Réutilisation	Non	Non	Non	Oui
Multi sites et utilisateurs	Oui	Oui	Oui	Oui
Architecture	Egal à égal	Egal à égal	Egal à égal	Egal à égal
Systèmes d'exécution	Système informatique	AIS	PDU de gestion de simulation	RTI
Interface informatique		Traducteur	Coupleur	API
Modèle de communication	Diffusion	Diffusion	Diffusion	Diffusion
Méthodologie				FEDEP
Base de données	Dupliquée	Dupliquée	Dupliquée	Distribuée
Transport d'informations	PDU	Chaînes ASCII	PDU	Modification des attributs
Filtrage messages	Non	Non	Non	Oui (DDM)
Représentation d'objets distants	Dead Reckoning	Objets fantômes	Dead Reckoning	Repercussion d'attributs
Gestion du temps	Temps réel	Temps coordonné	Temps réel	Temps réel/coordonné

Tableau : Comparaison des quatre architectures de simulation distribuée

Les différentes architectures de simulation n'utilisent pas les mêmes termes et n'ont pas les mêmes fonctionnalités. Néanmoins, pour les comparer, certains critères peuvent êtres utilisés tels que : l'interopérabilité, la réutilisation, la méthode de transport de données, les moyens pour la gestion du temps...etc.

L'architecture SIMNET est utilisée pour entraîner des troupes en temps réel. Pour faire interagir plusieurs simulations, cette architecture se base sur les PDUs qui sont tous d'ordre militaire. Cette architecture reste donc inaccessible par le milieu civil, d'où son inconvénient majeur. Quant à l'architecture ALSP, elle est utilisée pour construire des simulations constructives non temps réel. Une simulation constructive est une simulation où on trouve des simulacres de personnes qui utilisent des équipements simulés dans des conditions simulées ; les jeux de guerre ainsi que les simulations analytiques en sont des exemples. L'architecture DIS s'adresse aux simulations militaires d'entraînement en temps réel. Elle est limitée au domaine militaire. Enfin, l'architecture HLA traite tout type de simulation, du temps réel et du temps coordonné. De plus, elle est la seule architecture qui favorise la réutilisation. Elle sera dorénavant retenue pour la simulation distribuée.

4.5.2 Comparaison entre HLA et DIS

Il existe plusieurs similitudes entre DIS et HLA. Prenons l'exemple du cycle de vie d'un objet sous HLA. En première étape, un modèle objet de la fédération FOM est défini. Ce FOM décrit toutes les données échangées au cours de l'exécution d'une fédération et doit indiquer les conditions de ces échanges. Pour DIS, les documents et les PDUs décrivent déjà ces détails.

Le tableau suivant montre les étapes nécessaires sous HLA et DIS que le programme doit exécuter. Ainsi, ce tableau montre une comparaison entre le cycle de

vie d'un exercice sous HLA et sous DIS. Il y a quelques différences entre DIS et HLA qui ressortent de ce tableau :

- Quand un attribut d'un objet varie, DIS envoie tous les attributs de l'objet (PDU Entity State) alors que sous HLA, le fédéré transmet seulement l'attribut qui a été modifié en faisant appel à `Update Attribute Values`. Le RTI reçoit la valeur de l'attribut mis à jour et la transmet aux autres fédérés intéressés.
- Les évènements sous DIS sont gérés d'une façon implicite. En effet, si un simulateur reçoit un ESPDU d'une entité non connue alors il a découvert une nouvelle entité. Les protocoles de DIS se basent sur l'approche « j'envoie des données, donc j'existe ». Alors que sous HLA, le RTI exige que chaque objet soit instancié explicitement.

En dernier lieu, on peut noter que DIS est plus utilisé dans les simulations militaires vu son architecture qui se base sur un ensemble de messages (les PDUs) spécifiques au domaine militaire. De l'autre, côté HLA présente une architecture plus ouverte qui se base sur un ensemble de services qui sont communs à tous les fédérés. Ces groupes de services ne sont spécifiques à aucun domaine ; il suffit de déclarer les objets, les classes et les interactions spécifiques à ce domaine sans avoir à modifier l'architecture.

Action	DIS	HLA	Commentaires
Créer un exercice	Définir (ou utiliser) un ID d'exercice	`Create Federation Execution`	
Joindre l'exercice	Ecouter et envoyer des PDUs comme convenu	Join Federation Execution	Implicite vs Explicite
Obtenir l'ID d'un objet	L'application crée un ID unique	Demander l'ID au RTI	
Créer un objet	Commencer à émettre des ESPDUs	Instancier un objet	
Découvrir des objets	Recevoir des ESPDUS d'entités inconnues.	Instancier des objets découverts	Appel du RTI vers le fédéré
Le char se déplace	Envoyer un ESPDU	`Update Attribute Values` (position)	Le fédéré ne transmet que la valeur modifiée.
Le char déplace la tourelle	Envoyer un ESPDU	`Update Attribute Values` (orientation de la tourelle)	Le fédéré ne transmet que la valeur modifiée.
Le char tire	Envoyer un PDU de type Fire.	`Send Interaction`	
Suppression d'un objet	Arrêter d'envoyer des ESPDUs	`Delete Object`	Implicite vs Explicite
Quitter l'exercice	Arrêter de recevoir et d'envoyer	`Resign Federation Execution`	Implicite vs Explicite
Terminer l'exercice	Toutes les simulations s'arrêtent	`Destroy Federation Execution`	Implicite vs Explicite

Tableau : Comparaison du cycle de vie d'un exercice sous DIS et HLA **[Calvin, 1995]**

4.6 Outils et méthodologies HLA

HLA est une spécification et non une implémentation. Il existe donc plusieurs logiciels relatifs à HLA : des RTI, des outils pour la conception de fédération, des modèles objets, etc.

- Des RTI commerciaux : RTI-NG (développé par le DMSO), pRTI de Pitch, MaK RTI de MaK technologies.

- Des RTI libres tel que CERTI de l'ONERA de Toulouse, yaRTI et XRTI, poRTIco.

4.6.1 CERTI

CERTI est une infrastructure d'exécution pour les simulations distribuées à événements discrets développée à l'ONERA (Office National d'Etudes et de Recherches Aérospatiales) **[Site Web CERTI]**. Il fournit un ensemble de services basé sur les spécifications HLA 1.3.

Le développement du CERTI a débuté en 1996 et a été destiné à l'étude d'une architecture de simulation fournissant des extensions sécuritaires **[Bieber, 1998]**.

3.6.1.1 Architecture

CERTI est un prototype d'une RTI développée au CERT (Centre des Etudes et des Recherches à Toulouse). Le RTI est un système distribué contenant :

- Un processus local (RTIA).
- Un processus global (RTIG)
- Une librairie libRTI qui est liée à chaque fédéré.

L'architecture du CERTI est résumée sur la figure suivante : **[Bréhôlée, 2002]**

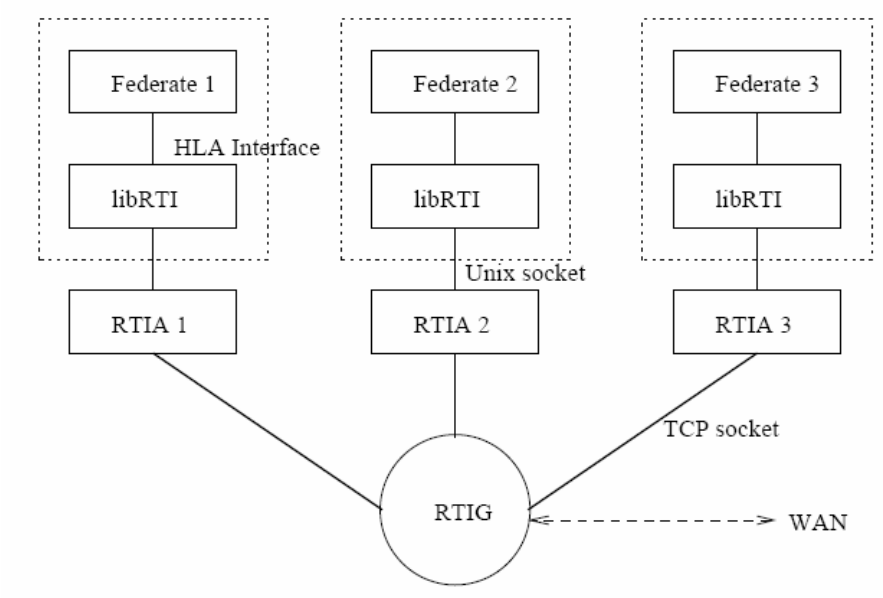


Figure 4.9 L'architecture du CERTI

Le RTIA

Chaque processus fédéré interagit localement avec le RTIA (RTI Ambassador) via un socket UNIX. Le processus RTIA échange des messages avec le RTIG à travers le réseau via des sockets TCP (et/ou) UDP. Il joue le rôle d'intermédiaire entre le fédéré et le RTI. On peut le comparer au LRC (Local RTI Component) du RTI du DMSO. Le rôle principal du RTIA est de satisfaire immédiatement les requêtes des fédérés. D'autres requêtes nécessitent l'envoi de messages au RTIG.

Le RTIG

Le RTIG, ou RTI Gateway, est le serveur principal jouant le rôle du RTI. La fonction principale du RTIG est de contrôler les communications entre les RTIAs, et entre les fédérés. Pour le mode fiable, le RTIG est un point de passage obligé entre deux fédérés.

La deuxième fonction principale du RTIG est de simplifier l'exécution de certains services HLA, parce que c'est un point central dans l'architecture. Il contrôle la création et la destruction de l'exécution de la fédération. Il enregistre les identités des

fédérés voulant publier ou s'inscrire pour les attributs d'une classe objet ou une classe interaction.

La librairie libRTI

La librairie libRTI est liée avec l'application. Elle fournit toute l'API de l'interface spécification de HLA. Son rôle est de transformer chaque appel au RTI Ambassador en un appel de socket UNIX contenant la requête ainsi que tous les paramètres associés. Ce message est transmis au RTIA et attend la réponse en retour d'appel dans le cas où une exception est levée ou si la méthode dispose de paramètres de retour.

Le scénario de transfert des données

La figure suivante illustre l'échange de mise à jour d'attributs **[Siron, 1998]**:

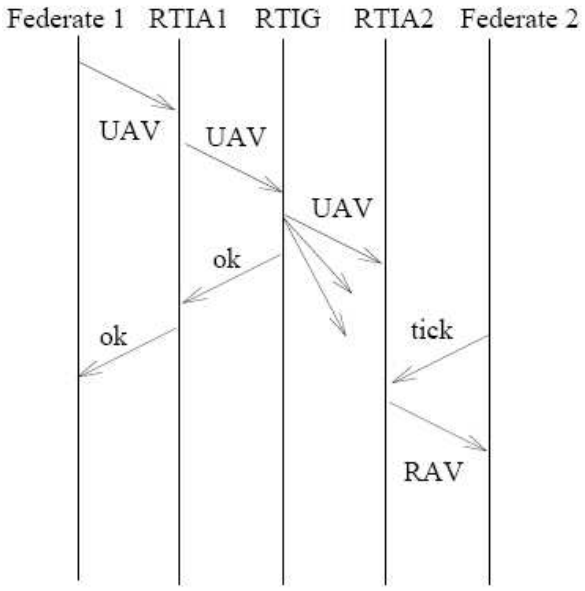


Figure 4.10 Scénario de transfert des données

UAV représente Update Attribute Values
RAV représente Reflect Attribute Values

L’appel de mise à jour est lancé par le fédéré n°1, la librairie libRTI génère le message et le transmet au RTIA1 du fédéré n°1 à travers le socket UNIX. Le RTIA1 reçoit le message et extrait les données. Le RTIA1 peut utiliser ces données pour mettre à jour des informations locales. Il génère un nouveau message réseau TCP et le transmet au RTIG. Ce dernier reçoit le message et renvoie un acquittement au RTIA1 qui génère un autre acquittement à destination du fédéré n°1. A partir de cet instant, l’appel Update Attribute Values du fédéré n°1 se termine. Il peut alors continuer l’exécution de la simulation.

Le RTIG émet un message Reflect Attribute Values vers tous les fédérés concernés soit par multicast soit par une socket TCP point à point.

Le message du RTIG est reçue par le RTIA2 du fédéré n°2 et est conservé dans une file d’attente. Le fédéré n°2 fait appel à la méthode tick(), la librairie libRTI demande au RTIA2 de recevoir les messages en attente. L’appel Reflect Attribute Values est effectué sur le Federate Ambassador du fédéré puis un acquittement est transmis au RTIA2.

4.6.1.2 Portabilité

Le CERTI fournit un ensemble de services basé sur les spécifications HLA 1.3. Le langage C++ est utilisé pour le code CERTI et pour le développement des fédérés. Plusieurs protocoles standard sont utilisés pour les communications entre composants comme le TCP/IP. Pour la gestion des processus, le CERTI utilise les bibliothèques standard d’UNIX. Il a un nombre très faible de problèmes de portabilité. Une version beta de CERTI sous Windows est sous test.

4.6.1.3 Applications avec CERTI

La première application était une application de test, elle consiste à simuler le jeu de billard. Dans cette première application, il y a une seule classe objet, la balle avec les attributs de position. La fédération est composée de plusieurs fédérés. Chaque

fédéré simule une instance balle. La collision est simulée par les interactions. Chaque fédéré publie ses attributs et s'inscrit pour les autres attributs.

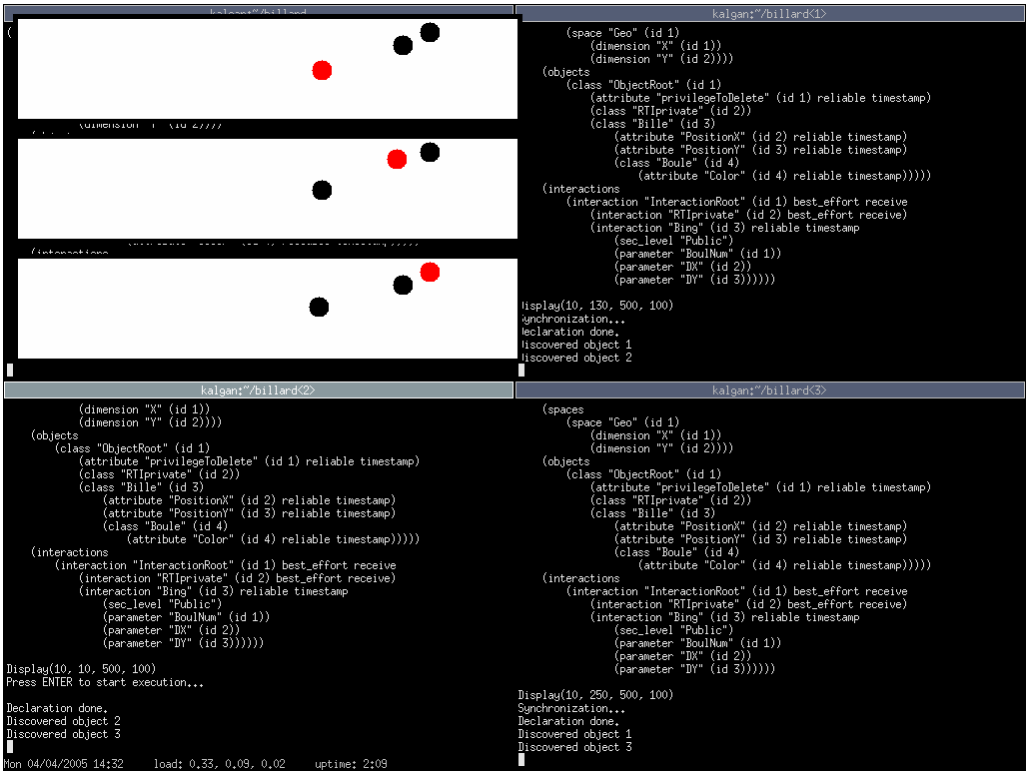


Figure 4.11 Application billard avec CERTI

Une autre application est la simulation d'un avion attaquant des unités de défense **[Adelantado, (a) 2001]**. D'autres applications ont été réalisées pour la simulation des aéroports de futur **[Adelantado, (b) 2001]**, combat de tanks (jeu) et pour la simulation des radars qui détectent les missiles et les avions.

4.6.2 Travaux HLA

4.6.2.1 Outils pour la simulation HLA

La plupart des applications de HLA sont liées au domaine militaire.

GENESIS [Site Web GENESIS]

C’est un projet dont le but principal est l’aide à la conception et aux développements des fédérés et de la fédération HLA. Il est développé à L’ONERA de Paris et financé par la DGA.

Le projet consiste à développer un cadre de travail pour les utilisateurs de HLA, composé de méthodologie et d’outils logiciels.

L’outil logiciel GENESIS est composé de trois parties :

- Un parseur qui est un analyseur lexical et syntaxique. Il lit les fichiers textuels fournis ; il les vérifie, les analyse, collecte les données et les stocke dans une mémoire, dans des structures appropriées.
- La seconde partie de l'outil est constituée par ces structures de données internes. Ce sont des classes C++ qui représentent en mémoire tous les éléments lus par le parseur.
- La troisième partie de l'outil GENESIS est constituée par les différentes actions de génération que l'on peut réaliser à partir des données collectées.

L’outil fonctionne en deux temps : D’abord, il lit une description textuelle de la simulation. Ce texte permet de représenter des besoins préliminaires (je veux simuler un vérin commandé par un automate programmable), puis par enrichissements successifs, il contient des informations de plus en plus précises (La tige du vérin effectue un mouvement de translation, décrit par une variable de déplacement x). L’étape ultime de la description consiste à rattacher les fonctionnalités HLA aux modèles physiques qu’on veut simuler (l’équation dynamique du vérin).

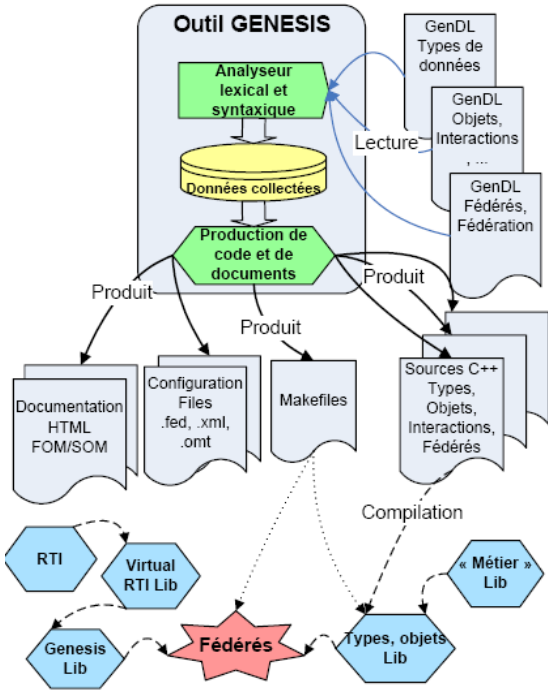


Figure 4.12 Vue d'ensemble de l'outil GENESIS

A partir de cette description, l'outil peut accomplir des actions très variées :

- Il peut vérifier la cohérence de la description.
- Il peut produire la documentation suivant la méthodologie définie par la norme.
- Il peut produire le code source (C++) des objets, types de données, simulateurs composants la simulation, ainsi que les moyens de compiler ce code source (Makefiles).

4.6.2.2 Méthodes et domaines d'application expérimentés

a) Dans le domaine des systèmes de production, on trouve dans **[Hibino, 2002]** un environnement pour relier des simulateurs de systèmes de production au RTI de HLA. Un adaptateur, appelé adaptateur de production, peut être relié au RTI et accéder aux services utilisés pour évaluer le système de production. La communication entre l'adaptateur et les simulateurs est réalisée par le transfert des messages sous le format XML. L'adaptateur traduit le message en format XML reçu

du simulateur en un appel d'un service HLA et le fournit au RTI. Il traduit aussi le message reçu par le RTI en un message sous format XML et le fournit au simulateur. Un fichier RDD est proposé qui définit le contenu des messages échangés entre les simulateurs. L'échange des données entre le simulateur et l'adaptateur en utilisant les fichiers XML peut entrainer une perte du temps. Cette méthode n'est pas applicable au niveau des simulateurs mécatroniques parce qu'elle ne favorise pas la simulation en temps réel.

b) Dans le domaine des systèmes de contrôle, on peut citer deux applications majeures :

Le simulateur hybride Anylogic dispose actuellement des fonctionnalités de HLA **[Borchshev, 2002]**. Une interface universelle appelée StepHook est employée en particulier par le module HSM (HLA Support Module) qui permet de relier Anylogic et une fédération HLA. Pour le simulateur MATLAB, plusieurs méthodes ont été aussi proposées qui utilisent un «wrapper » entre MATLAB et le RTI en utilisant des bibliothèques externes. L'implémentation d'une interface MATLAB/HLA a été développée dans le projet de recherche appelé « la boite à outils HLA » **[Pawletta, 2000]**. Cette méthode n'est applicable que pour les simulateurs disposant d'une interface de librairie.

c) Dans le domaine des systèmes 3D de CAO, l'approche en **[Kanai, 2003]** propose une méthode pour transformer un système CAO 3D en un système compatible avec HLA. Cette méthode est appliquée seulement aux simulateurs qui ont une interface API (Application Program Interface). Dans chaque fédéré, deux parties du code sont implémentées :
La première partie du code détecte les événements, obtient les valeurs des attributs d'objets du modèle interne du fédéré en employant l'interface spécifique du simulateur et les traduit en rappels de service vers le RTIAmbassador. Cette partie de code s'appelle LRC (Local RTI Component).

La seconde partie du code, appelée Fed-Code, détecte les évènements ou les valeurs à modifier des attributs d'objets à partir du RTI et les traduit en des appels de mise à jour vers l'API en utilisant les fonctions de rappel dans la classe FederateAmbassador. Cette méthode n'est applicable que pour les simulateurs qui ont une interface API ainsi qu'une documentation complète.

d) Dans le domaine des systèmes de prototypage virtuel, une approche pour relier la plateforme oRis, orientée agent, à HLA a été expérimentée **[Raulet, 2003]**. Oris a été développée par Fabrice Harrouet comme thème de sa thèse sur la réalité virtuelle **[Harrouet, 2000]**. C'est une plate-forme de prototypage interactif développée par le laboratoire d'ingénierie informatique de Brest. Il dispose d'un langage interprété éponyme permettant de modéliser des agents et de les faire évoluer au sein d'un même environnement. L'architecture oRisDis propose de coupler la plate-forme Oris avec l'architecture HLA dans le but d'offrir un environnement interactif, collaboratif et dynamique. Le principe retenu est qu'une plate-forme oRis distribuée ne peut participer qu'à une seule fédération à la fois. Une plate-forme oRis représente donc un seul fédéré, dans la terminologie HLA. Une plate-forme oRis contient plusieurs agents qui collaborent ensemble. Certains des agents peuvent être locaux à la plate-forme, tel qu'un agent chargé d'effectuer des traitements. D'autres agents, par contre, participent activement à la collaboration. Les agents dans une plate forme oRis collaborent avec un agent spécifique, faisant partie de l'oRisRTI, qui va collecter les modifications à échanger. Du côté du récepteur, tous les évènements reçus depuis le RTI vont être traités par l'oRisRTI. Un agent se charge de transmettre les nouvelles valeurs aux agents distribués concernés. Tous les échanges entre le RTI et oRis passent par l'oRisRTI.

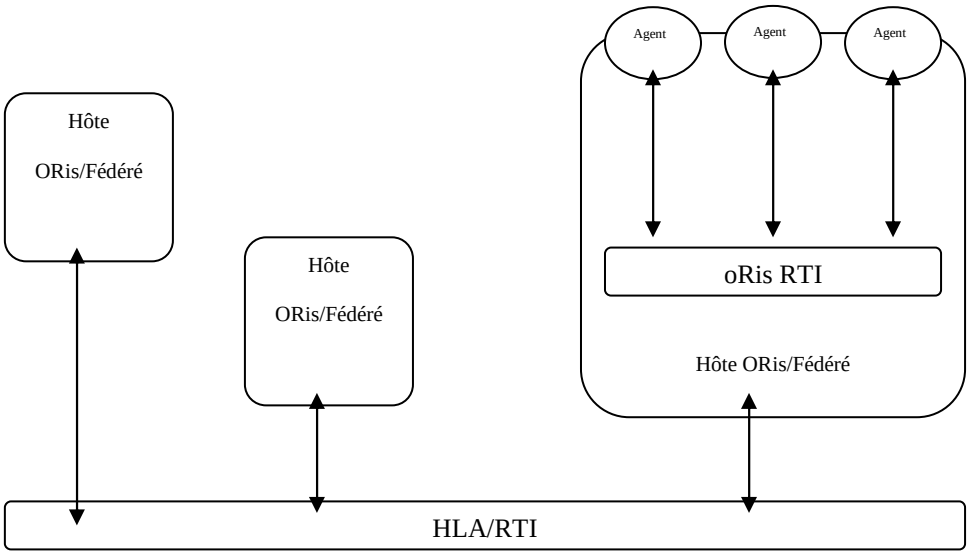


Figure 4.13 Architecture *oRisDis*

La plate-forme oRisDis permet de réaliser des applications de prototypage interactif à plusieurs utilisateurs. Cette architecture propose une double infrastructure composée de l'architecture HLA et d'une seconde architecture proposant l'interface entre la plate-forme oRis et le RTI HLA. L'oRisRTI joue un rôle important dans le couplage des plates-formes oRis participantes. Le rôle du RTI est d'effectuer le transport des informations et d'assurer certains services comme la gestion de la distribution des données ou la gestion du temps. L'oRisRTI assure l'interconnexion des plates-formes oRis.

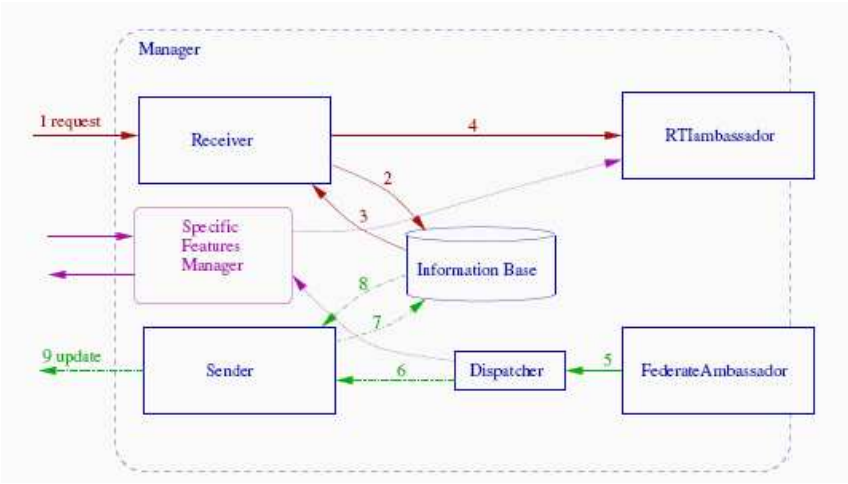


Figure 4.14 Composants du gestionnaire oRisRTI

L’interface avec le RTI est effectuée à l’aide de RTIambassador et FederateAmbassador. L’interface avec oRis est effectuée à l’aide de Sender et Receiver.

- RTIambassador : permet de transmettre des informations vers le RTI en utilisant une API bien définie.
- FederateAmbassador : reçoit les messages venant du RTI, extrait les données, et effectue le transfert vers le Dispatcher.
- Receiver : c’est un agent chargé de recevoir les requêtes provenant de différents agents partagés dans la plate-forme locale.
- Sender : c’est l’agent qui reçoit un certain nombre de mises à jour provenant du FederateAmbassador.
- Information Base : c’est une base de données locale à la plate-forme. Elle stocke les dernières valeurs d’attributs reçues depuis les agents de la plate-forme locale ou depuis le réseau (agents de plates-formes distantes).
- Dispatcher : Son rôle est de transmettre l’information reçue depuis le FederateAmbassador vers le composant chargé du traitement de l’information reçue.

4.7 La gestion du temps sous HLA

Le temps dans le système modélisé par le fédéré peut être représenté comme des points le long d’un axe de temps de la fédération. Chaque fédéré peut alors avancer le long de cet axe pendant l’exécution. Son avancée est soit contrainte par l’avancée des autres fédérés soit non contrainte. Le service de gestion du temps propose les mécanismes permettant de contrôler l’avancement des fédérés afin de garantir un ordre causal entre les différents évènements émis.

La perception du temps courant peut être différente suivant les fédérés mais l’avancement du temps est coordonné par la fédération. L’avancement du temps d’un

fédéré peut être soit *contraint* par la progression des autres fédérés soit il peut être *régulateur* et gère alors l'avancée de la simulation.

Fédéré régulateur : Un fédéré régulateur est capable d'émettre des évènements TSO. Les fédérés non régulateur peuvent émettre des évènements mais aucune date n'y sera associée.

Fédéré contraint : Un fédéré contraint est capable de recevoir des évènements TSO. Ceux n'étant pas contraints reçoivent quand même l'événement mais sans l'information de la date. Ils les reçoivent dans l'ordre uniquement (RO).

4.7.1 Les évènements TSO et RO

Les évènements reçus sont soit RO (Receive order) ou TSO (Time Stamp Order). Seuls les évènements TSO portent une signification temporelle. Ils portent un temps estampillé et ils ont l'ordre de leur arrivée garanti par le RTI.

Les messages sont caractérisés par un type d'ordre qui indique la manière de les transmettre :

- RO : transmission des messages au fédéré dans l'ordre où ils ont été reçus : file FIFO (First In First Out).
- TSO : transmission des messages aux fédérés selon leur ordre d'estampille. Les messages n'arrivent jamais dans le passé du fédéré destinataire.

Quand un événement est reçu comme RO ou TSO alors deux questions se posent :

- Comment il a été envoyé ?
- Comment il a été reçu ?

Pour un événement envoyé TSO, il faut vérifier pour le fédéré émetteur de l'événement :

- Le fédéré émetteur de l'événement doit être régulateur.
- Les attributs et les classes d'interactions doivent avoir l'ordre de type TSO. (définis dans le fichier FED)
- L'invocation de UPDATE ATTRIBUTE VALUES ou SEND INTERACTION doit inclure un temps estampillé.

Pour qu'un événement soit reçu comme TSO, il faut vérifier :

- L'événement doit être envoyé TSO.
- Le fédéré récepteur doit être contraint.

Un fédéré contraint peut recevoir un événement comme TSO. Un évènement envoyé comme TSO va être reçu comme RO par un fédéré non contraint.

4.7.2 Lookahead

HLA demande aux fédérés **régulateurs** de fournir un *lookahead*.

Supposons que F est un fédéré qui peut envoyer et recevoir des messages TSO et L un nombre d'unités de temps. Exigeons qu'aucune simulation ne va générer un événement avec une estampille inférieure à T+L, avec T= temps logique de F.
Si le temps logique de F est T, alors tout événement généré par F aura une estampille supérieure ou égale à T+L. Ceci va permettre à un autre fédéré d'avancer son temps logique à T+L.
L sera le *lookahead* du fédéré F.
En effet, un fédéré de temps logique T et de *lookahead* L ne doit pas envoyer de message ayant une estampille de valeur inférieur à T+L.

Chaque fédéré doit déclarer un lookahead non négatif. Chaque TSO envoyé par un fédéré doit avoir une estampille égale **au moins** à son temps logique courant plus son lookahead. Le lookahead peut changer pendant l'exécution :

- L’augmentation du lookahead prend effet immédiatement.
- La diminution du lookahead ne prend effet que lorsque le fédéré avance son temps logique.

4.7.3 LBTS (Lower Bound of the Time Stamps of messages)

La clef pour implémenter les services de gestion du temps dans le RTI est de calculer le LBTS pour chaque fédéré. Pour un fédéré F, le RTI doit assurer que :

- Les messages TSO doivent être délivrés à F dans l’ordre de leurs estampilles.
- Aucun message ne doit être délivré à F avec une estampille inférieure à son temps logique.

Le LBTS signifie la date minimale possible à laquelle le fédéré peut recevoir un événement TSO. Cette valeur est déterminée en regardant le message le plus proche dans le temps qui peut être généré par les fédérés régulateurs. Un fédéré contraint ne peut aller au delà de cette date.

Rôle des fédérés dans le calcul du LBTS

Le RTI doit identifier les fédérés qui doivent participer au calcul du LBTS et ceux qui exigent le résultat du calcul du LBTS. **Les fédérés régulateurs** génèrent les messages TSO donc ils doivent participer au calcul des LBTS. **Les fédérés contraints** reçoivent les messages TSO donc ils exigent les résultats du calcul des LBTS.

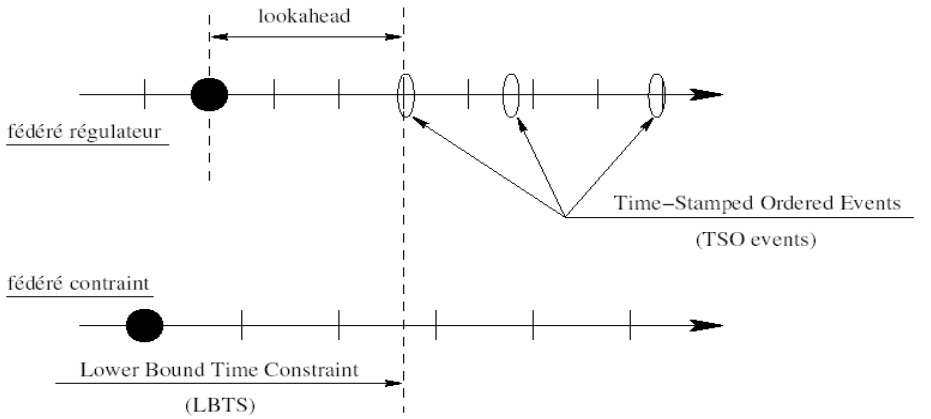


*Figure 4.15 L*ookahead et LBTS [Raulet]

4.7.4 Avancement du temps

Un fédéré ne peut avancer dans le temps qu'avec l'autorisation du RTI. Un fédéré émet une requête d'avancement dans le temps et en fonction des messages à venir, il peut être autorisé par le RTI à effectuer cet avancement.
Trois manières d'avancer existent (cf. pages 64-65) : exécution dirigée par le temps temps (time-step), exécution dirigée par événements (event-based) ou optimiste. **[Fujimoto, 1998]**

Exécution dirigée par le temps (time-step)

Les fédérés dits time-stepped calculent les valeurs par rapport à un moment donné en traitant les évènements qui se produisent jusqu'au temps courant + time step.

Le fédéré appelle le service **timeAdvanceRequest(T)** pour faire avancer son temps logique jusqu'à T. Tous les messages RO dans les files internes du RTI, et tous les messages dont l'estampille est inférieure ou égale à T sont délivrés au fédéré. Quand il n'y a plus de messages TSO avec une estampille inférieure ou égale à T qui vont être générés par d'autre fédérés, le RTI appelle la procédure

timeAdvanceGrant() avec le paramètre T pour indiquer que le temps logique du fédéré a été avancé à T.

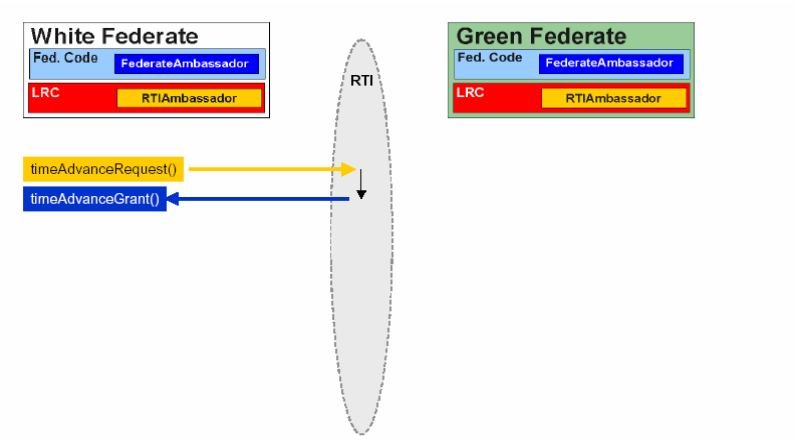


Figure 4.16 Avancement du temps logique : time-step

Exécution dirigée par événements

Les fédérés basés événement calculent leur temps en fonction de chaque événement reçu. Le fédéré appelle le service **nextEventRequest(T),** le RTI va délivrer tous les messages RO dans sa file d'attente interne. S'il n'y a pas de messages TSO avec une estampille inférieure ou égale à T et aucun message qui va être reçu dans le futur, le RTI appelle la procédure **timeAdvanceGrant** qui indique que le temps logique du fédéré a été avancé jusqu'à T, sinon le RTI va délivrer le prochain plus petit message TSO destiné au fédéré (avec une estampille T' <= T) ainsi que les autres messages qui ont une estampille T'. Le RTI appelle **timeAdvanceGrant** avec le paramètre T'. Dans ce dernier cas, le temps logique est avancé jusqu'à T'. (Voir exemple d'exécution)

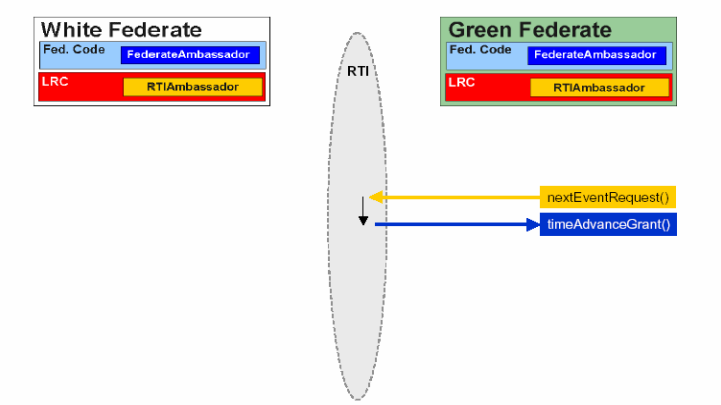


Figure 4.17 Avancement du temps logique : basé évènement

Optimiste

Les fédérés optimistes ont plus de liberté dans l'avancement de leurs temps logiques. Ils créent des évènements qui se produiront dans le futur. Le risque est de recevoir un message qui annule un message envoyé par ce fédéré, il aura alors la charge de se rétracter et de retirer ce message.

4.8 Conclusion

Nous avons présenté dans ce chapitre les architectures de simulation distribuée les plus connues dans un ordre chronologique. Nous avons choisi d'utiliser l'architecture HLA qui est de facto l'architecture la plus utilisée dans les simulations distribuée. Cette architecture est la mieux adaptée à notre application. Par la suite, nous avons présenté en détail cette architecture ainsi que les travaux autour de cette architecture dans différents domaines. Nous avons présenté aussi l'outil CERTI qu'on va utiliser par la suite dans notre application.

5. Chapitre 5: Choix de distribution des modèles sous OpenModelica et implémentation des automates hybrides

5.1 Introduction

Nous présentons dans ce chapitre deux approches pour la modélisation et la simulation d'un système mécatronique en utilisant les deux simulateurs OpenModelica pour la partie comportementale et OpenMASK pour la partie géométrique. Un choix de l'une de ces approches est effectué. Suite à ce choix, nous présentons une méthode générale et orientée objet pour implémenter le formalisme des automates hybride avec le langage Modelica indépendamment de l'outil utilisé. Nous présentons par la suite une méthode pour modéliser le couplage entre les automates hybrides des parties commande et opérative. Nous présentons à la fin une bibliothèque Modelica que nous avons développée, nommée HybridAutomataLib, qui permet la modélisation des automates hybrides.

5.2 Choix de modèles et d'architecture sous OpenModelica

La mécatronique fait intervenir plusieurs techniques de différents domaines. L'ingénierie de tels systèmes mécatroniques exige la conception simultanée et pluridisciplinaire de trois sous systèmes :

- Une partie opérative.
- Une partie contrôle.
- Une interface Homme/Machine.

La simulation d'un système mécatronique par conséquent doit supporter d'une part l'évolution dynamique représentée par les automates hybrides et d'autre part sa représentation graphique. Pour modéliser et simuler de tells systèmes nous proposons les deux approches suivantes:

- Première approche : Le simulateur OpenModelica est utilisé pour modéliser la partie opérative par ses composants prédéfinis dans ses librairies. La partie contrôle est modélisée aussi sous OpenModelica par les automates hybrides. Une liaison entre l'automate hybride de contrôle et les composants de la partie opérative est nécessaire. De l'autre côté, nous utilisons l'environnement virtuel OpenMASK pour la représentation graphique 3D. La communication entre ses deux simulateurs est maintenue par le RTI.

- Deuxième approche : Nous modélisons les deux parties contrôle et opérative par les automates hybrides qui sont implémentés sous OpenModelica. De l'autre côté, nous utilisons l'environnement virtuel OpenMASK pour la représentation graphique 3D. La communication entre les deux simulateurs est assurée par le RTI.

Notre choix s'est porté sur la deuxième approche parce que nous préférons avoir un modèle comportemental global utilisant les automates hybrides. En effet, la librairie Modelica n'est pas complète et plusieurs bibliothèques ne sont pas gratuites.

Nous identifions deux fédérés : un fédéré pour l'animation des objets 3D et un autre fédéré de comportement qui représente la partie contrôle et la partie opérative du système mécatronique. Le premier fédéré va être implémenté sous l'environnement virtuel OpenMASK, le deuxième fédéré sera implémenté sous le simulateur OpenModelica.

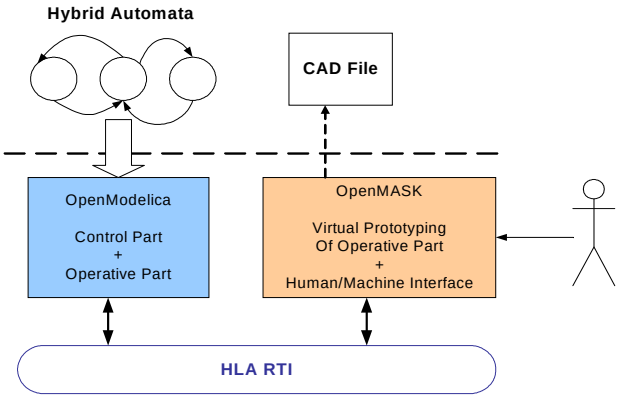


Figure 5.1 Approche globale

5.3 Implémentation des automates hybrides sous OpenModelica

En conséquence de nos choix, nous avons intégré les automates hybrides dans Modelica car cela n'existait pas et nous donnons ici les principes suivis **[Hadj-Amor, 2007]**. Ce langage de modélisation physique orienté objet, fournit les notions de classe et objet. Les fonctionnalités hybrides de Modelica permettent la modélisation des évènements discrets et des sauts de variables. Plusieurs fonctions permettent de générer et de masquer des évènements, de détecter la variation d'une variable continue ou discrète, réinitialiser des variables quand un évènement apparaît, d'identifier le début et la fin d'une simulation. Plusieurs formalismes de représentation des systèmes hybrides ont été intégrés à Modelica comme les Bonds Graphs **[Zimmer, 2006]**, les réseaux de Petri **[Mosterman, 1998]** et les statecharts **[Otter, 2005]**.

Considérant l'analogie entre une classe Modelica et une situation de l'automate hybride, on a pu modéliser chaque situation et chaque transition de ce dernier par une classe Modelica. Nous présentons dans ce qui suit une méthode orientée objet pour l'implémentation des automates hybrides avec Modelica.

5.3.1 Modélisation Orienté Objet

L'approche orientée objet sous Modelica est vue comme un concept de structuration utilisé pour traiter la description complexe des systèmes. Un modèle Modelica est essentiellement une description mathématique déclarative. Les propriétés des systèmes dynamiques sont exprimées d'une façon déclarative à travers des équations. Le concept de programmation orienté objet déclaratif du point de vue Modelica peut être résumé en ces points:

- L'approche orientée objet est utilisée essentiellement comme un concept de structuration encourageant la structure déclarative et la réutilisation des modèles mathématiques.
- Les propriétés dynamiques du modèle sont exprimées d'une manière déclarative

à travers des équations.

- Un objet est une collection de variables et d'équations qui partagent un ensemble de données.
- La notion orienté objet dans la modélisation mathématique n'est pas vue comme le passage de messages dynamiques entre objets.

L'approche orientée objet déclarative offerte par Modelica pour décrire les systèmes et leurs comportements est d'un niveau d'abstraction plus haut que celui de l'approche orientée objet usuelle puisque plusieurs détails d'implémentation peuvent être exclus. Par exemple, ce n'est pas nécessaire d'écrire du code explicite pour décrire le passage de données entre les objets. Ceci est généré automatiquement par le compilateur Modelica. Nous allons nous baser sur ces caractéristiques pour modéliser un automate hybride.

5.3.2 Modélisation d'une situation

Le comportement continu dans un automate hybride est modélisé par une équation algébro-différentielle. La déclaration de cette équation se fait dans le bloc ***equation*** présent dans chaque modèle Modelica.

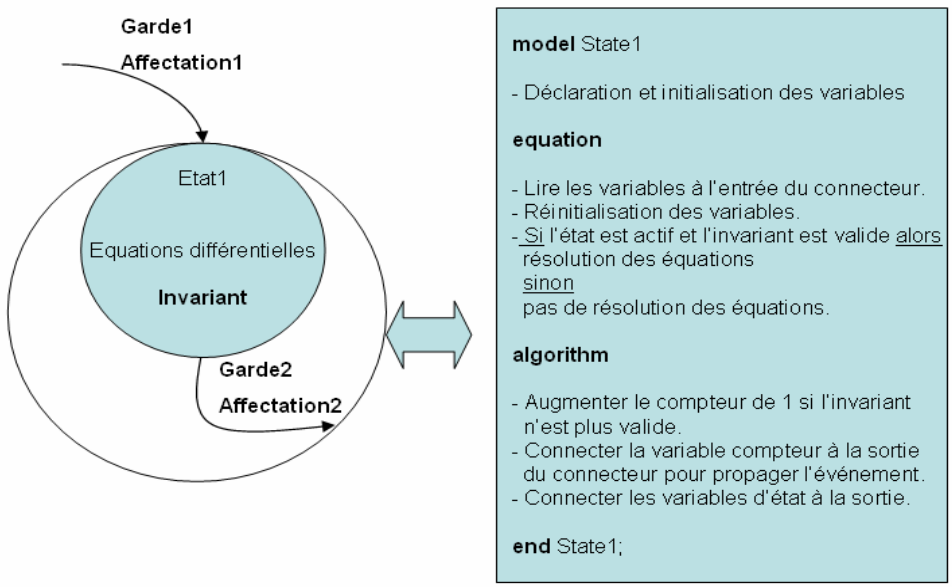


Figure 5.2 Modèle d'une situation

L’invariant qui représente les conditions pour rester dans une situation est modélisé par un test conditionnel (équation booléenne) qui doit être exécuté dans le bloc ***equation***. La situation reste active si le test est vrai et elle est désactivée quand le test devient faux. Chaque modèle *Etat* peut avoir une ou plusieurs connexions de sortie et d’entrée. On déclare deux connecteurs pour chaque situation: un connecteur d’entrée et un connecteur de sortie. Chaque connecteur peut avoir un ou plusieurs attributs. On crée des attributs pour les variables de la simulation et un attribut entier pour propager l’événement aux transitions sortantes.

5.3.3 Modélisation de la transition

Chaque transition reliant deux situations est modélisée par une classe Modelica qui doit tester les conditions de franchissement et commuter d’une situation à une autre (Figure 5.3). Une transition est franchissable seulement si la garde est vraie et l’invariant de la situation cible est vraie. La classe Modelica représentant le modèle *Transition* comporte aussi un connecteur d’entrée et un autre de sortie. Les variables transmises à partir d’un modèle de situation aux différentes transitions sortantes sont récupérées en connectant les sorties des modèles de situation aux entrées des modèles de chaque transition sortante. On utilise les équations ***connect*** pour établir des connexions entre les composants à travers les connecteurs. Le test des gardes est modélisé par un ensemble d’équations conditionnelles en utilisant la clause if-Equations.

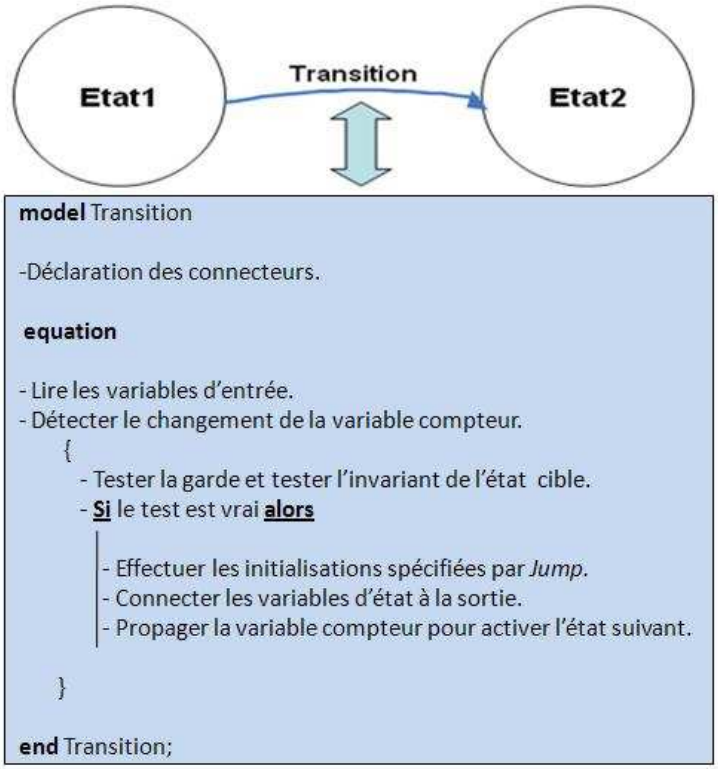


Figure 5.3 Modèle d'une transition

Une affectation est associée à chaque transition pour attribuer de nouvelles valeurs aux variables d'environnement dans la nouvelle situation avant le commencement de l'évolution continue. L'affectation est le transfert de données de l'ancienne situation vers la nouvelle situation. Le modèle *Transition* prend en compte les modifications des variables d'état lors d'une transition (Jump) et met ces valeurs sur le connecteur de sortie. Le passage de variable d'une situation à une autre se fait par la connexion de la sortie du modèle transition à l'entrée de la nouvelle situation en utilisant les équations ***connect*** . (Figure 5.4)

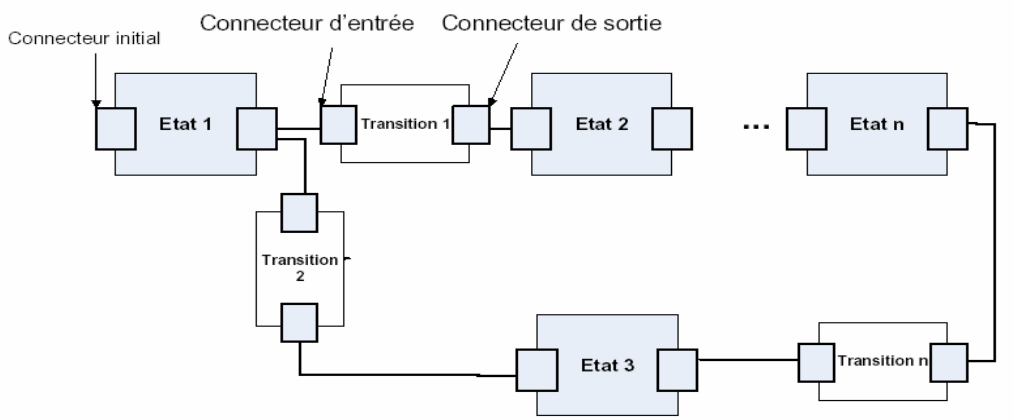


Figure 5.4 Modélisation orienté Objet

5.3.4 Description de la dynamique

On décrit dans cette partie l'évolution dynamique d'un automate hybride modélisé par des modèles Modelica. Une fois actif, l'objet associé à l'état continue son évolution et reste actif tant que le test de l'invariant est vrai. Si l'invariant est faux l'objet se désactive et ne continue pas son évolution.

Quand la situation se désactive, elle met une valeur entière sur son connecteur de sortie. Cette valeur est un compteur du nombre de fois où la situation se désactive. Les modèles des transitions sortantes lisent en continuité la valeur de ce compteur (Figure 5.3). Si cette variable change de valeur, ceci veut dire que la situation ancienne vient de se désactiver et donne le contrôle à chaque transition sortante qui doit vérifier elle même si elle est franchissable ou pas. Si la garde est vraie et l'invariant de la situation cible est vrai, le modèle transition communique les nouvelles valeurs aux variables d'environnement dans la nouvelle situation (On utilisera les évènements plus tard dans le cas des automates hybrides couplés).

5.3.5 Exemple

Nous avons effectué une expérimentation pour valider notre approche d'implémentation des automates hybrides sous OpenModelica **[Hadj-Amor , 2007]**.

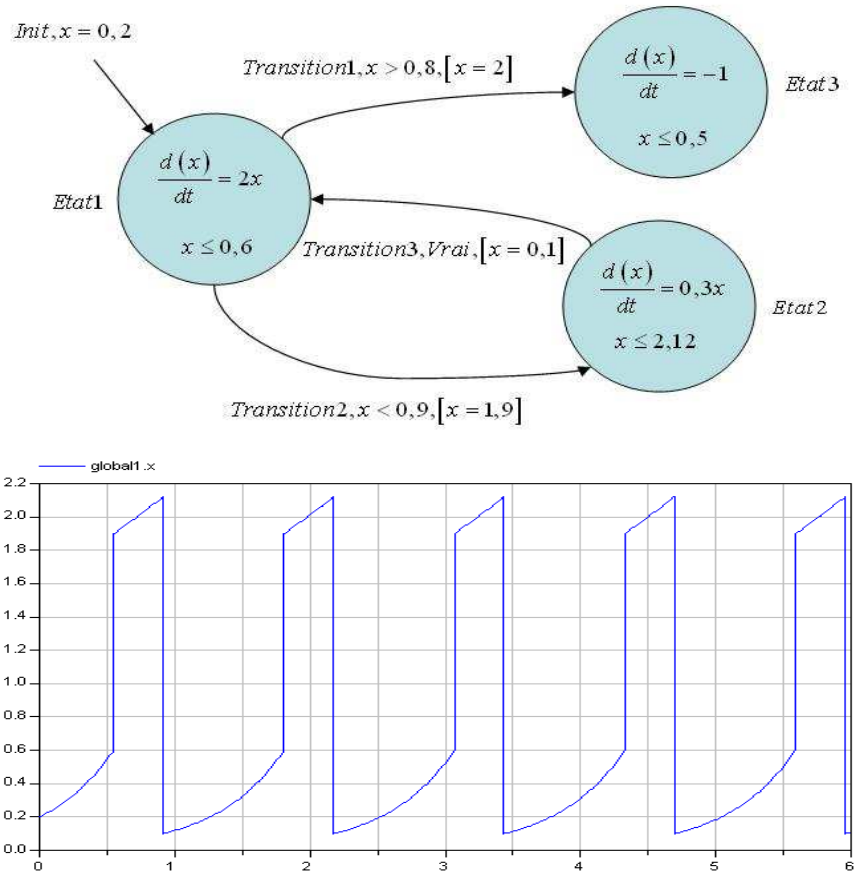


Figure 5.5 Exemple d'automate hybride simulé sous Modelica

La transition 1 ne sera jamais franchie car la garde ne sera jamais vraie. Les résultats de la simulation montrent bien que l'état 3 n'est pas atteignable.

5.3.6 Modélisation du couplage

Il est possible de coupler des automates hybrides en se basant sur le mécanisme de composition synchrone. Ce couplage a pour but de séparer la partie contrôle de la partie opérative. Une première définition est la suivante: « Si l'on considère deux automates et un événement qui appartient à l'ensemble des évènements de chacun d'eux, une transition étiquetée par cet événement ne peut être franchie dans l'un des automates que s'il existe une transition étiquetée, par ce même événement, franchissable dans l'autre automate. Les deux transitions sont alors franchies simultanément. »

Prenons le cas de l'exemple général suivant :

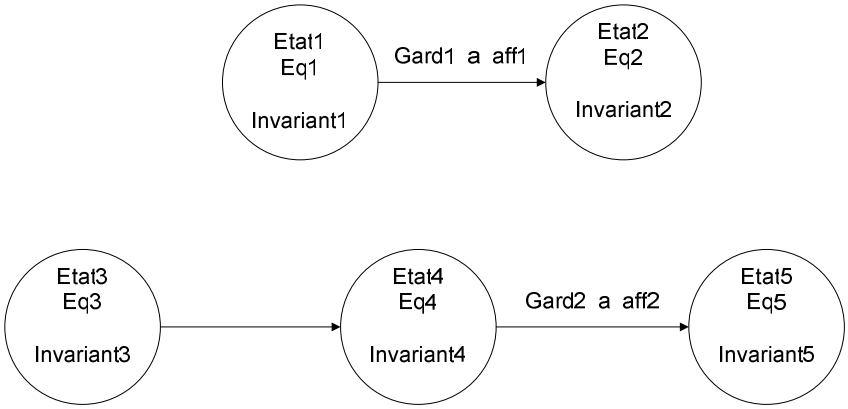


Figure 5.6 Exemple de couplage

La transition de l'automate hybride en haut est étiquetée avec l'évènement *a* qui appartient également à l'automate hybride du bas. Cette transition n'est donc franchissable que si la situation *Etat1* est active et sa condition de garde *Gard1* vraie, mais aussi si la situation *Etat4* de l'automate du bas est active et que la condition de garde de la transition de *Etat4* vers *Etat5* (*Gard2*) vraie. Dans tous les autres cas, la transition n'est pas franchissable. Ceci peut être modélisé par le produit booléen des deux gardes sur les deux transitions qui sont étiquetées avec le même évènement. De même pour les invariants des états de départ (*Etat1* et *Etat4*). Nous obtenons ainsi les automates équivalents suivants.

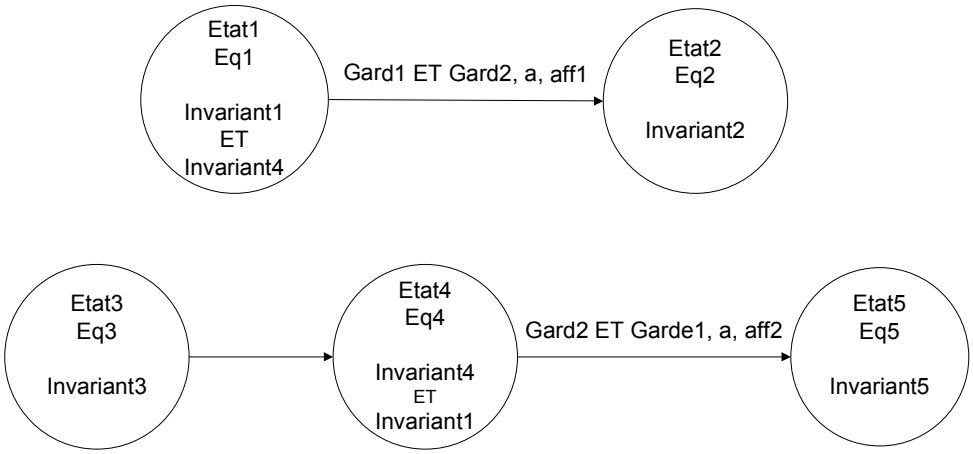


Figure 5.7 Exemple de couplage transformé

Dans la situation *Etat1*, l'invariant global, qui est *Invariant1 ET Invariant4*, fait intervenir des variables d'états hors de l'automate partiel du haut. De même pour les transitions étiquetées avec l'évènement *a* dans les deux automates, la garde, qui est *Garde1 ET Garde2*, fait intervenir des variables d'états hors des variables de l'automate partiel. Ceci est possible avec le langage Modelica. En effet, il est possible, à partir d'un automate hybride partiel 1, d'accéder à des variables d'états d'un automate hybride partiel 2. Par exemple, dans l'*Etat1*, il est possible de lire les variables d'états contenues dans *Invariant4*. Ainsi, à tout moment de la simulation, on peut évaluer la condition *Invariant1 Et Invariant4*. De même pour la condition *Garde1 ET Garde2*, on peut l'évaluer à tout moment de la simulation puisque on peut accéder à tous les variables d'états des deux automates hybrides partiels. Notre méthode sous Modelica n'utilise pas explicitement les notions d'évènements et d'étiquettes du formalisme des automates hybrides. Le franchissement sera déclenché sur l'occurrence *Garde1 ET Garde2* vraie. Pour cela, nous utilisons une notion très intéressante dans Modelica ; les connecteurs.

Nous pouvons établir une connexion entre *Etat1* et *Etat4*. Il suffit de créer un connecteur sur chacun des deux situations. Les deux connecteurs sont de même type. Dans chaque connecteur, on déclare l'ensemble des variables d'états contenues dans les deux situations. Ainsi, dans chaque situation *Etat1* ou *Etat4* et à tout moment de la simulation, la lecture des variables d'états de l'autre automate est possible. Dans la figure suivante, nous établissons une connexion entre *Etat1* et *Etat4* en supposant que *Etat1* a comme variable d'état *A1* et que *Etat4* a comme variables d'état *A4* et *B4*. Ainsi, nous utilisons un connecteur qui déclare trois variables.

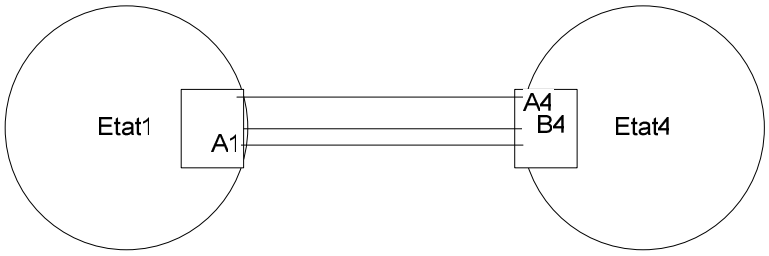


Figure 5.8 Exemple de connecteur entre deux situations

Nous avons modélisé les transitions d'un automate hybride par des classes (modèles) Modelica. Ainsi nous pouvons aussi, d'une façon similaire aux situations, établir une connexion entre deux classes *Transition*. En utilisant cette méthode, nous pouvons ainsi modéliser le couplage des automates hybrides grâce à l'approche, orientée objet, exploitée par Modelica et grâce la notion des connexions utilisée entre les différents modèles Modelica.

5.3.7 Implémentation de la méthode et description de la librairie HybridAutomataLib

Nous proposons dans ce paragraphe une implémentation de notre approche orientée objet pour la modélisation et simulation des automates hybrides avec Modelica. Nous nous basons donc sur les fonctionnalités hybrides du langage Modelica présentées dans le paragraphe 2.4.2.6. Nous présentons une classe (modèle) Modelica pour la modélisation d'une situation en général d'un automate hybride. Cette classe est nommée *state*. Nous nous limitons à une situation d'un automate hybride avec une seule variable d'état et une équation différentielle de premier ordre. Nous présentons à la fin de ce paragraphe une méthode pour l'implémentation des équations de second ordre ainsi qu'une méthode pour le cas d'une situation avec plusieurs variables d'état.

5.3.7.1 La classe *state*

Dans la figure 5.12, nous avons découpé le modèle *state* en plusieurs parties afin de mieux le décrire.

Déclaration des connecteurs d'entrée et de sortie : dans cette partie nous déclarons deux connecteurs : un connecteur d'entrée *din1* et un connecteur de sortie *dout1*. Les connecteurs sont de type *connector2*. En effet, nous avons créé une classe connector2 qui contient deux variables : une variable *s* de type **Real** qui correspond à la variable

d'état de l'automate et une variable *numbevent* de type **Integer** pour propager les évènements d'activation ou de désactivation d'une situation.

```
connector connector2
  Real s;
  Integer numbevent;
end connector2;
```

Figure 5.9 Classe connector2

Déclaration des variables d'état : dans cette partie nous déclarons les variables d'état de la situation.

Déclaration des variables pour le déclenchement d'évènements : nous déclarons dans cette partie deux variables : *eventVisualizer1* de type Integer (entier) et *state* de type Boolean (booléen).

Déclaration des paramètres de l'équation différentielle : dans cette partie nous déclarons les paramètres de l'équation différentielle. Ils sont de type *parameter*. Nous déclarons aussi la variable *INV* qui représente l'invariant. Dans le cas où l'invariant est représenté par une condition sous forme d'un intervalle (INV1 <= x < INV2), nous devons déclarer deux variables *INV1* et *INV2*. Nous déclarons aussi une variable *Init* de type *parameter*. Cette variable représente l'affectation de la transition **Init** dans un automate hybride.

Initialisation : dans cette partie nous initialisons la variable d'état juste au lancement de la simulation. Nous utilisons les fonctions *initial()* et *reinit()* qui sont décrites précédemment dans le paragraphe 2.4.2.6.

Réinitialisation : dans cette partie nous réinitialisons la variable d'état au cours de la simulation. En effet, si au cours de la simulation la situation est non active et qu'un évènement d'activation est reçu à travers le connecteur d'entrée, alors la variable d'état est réinitialisée avec l'affectation faite au niveau de la transition qui est représentée par din1.s. Pour détecter l'évènement d'activation reçu au niveau du

connecteur d'entrée un test est effectué sur la variable *numbevent* en utilisant la fonction *change()* présentée dans le paragraphe 2.4.2.6. Nous utilisons une expression équivalente à la fonction *change()* puisque dans la version actuelle de OpenModelica cette fonction n'est pas encore implémentée. Dans cette expression nous utilisons alors la fonction *pre()*.

Résolution des équations : Dans cette partie, l'équation différentielle est résolue tant que l'invariant est vérifié, sinon la variable d'état est maintenue constante.

Propagation des variables d'état vers les transitions sortantes : Nous mettons la variable d'état sur le connecteur de sortie.

Vérification de l'invariant : cette partie est implémentée dans le bloc *algorithm*. Nous utilisons l'expression *when* pour vérifier si l'invariant est vérifié ou non. Si ce dernier n'est plus vérifié la variable booléenne *state* passe de faux à vraie.

Déclenchement d'un évènement : dans cette partie nous déclenchons un évènement dès que la variable *state* est vraie. La variable *eventVizualiser1* est incrémentée de 1.

Propagation de l'évènement : nous mettons la variable *eventVizualiser1* sur le connecteur de sortie pour faire propager l'évènement.

Cette classe Modelica développée représente une situation d'un automate hybride avec une équation différentielle de premier ordre. En mécanique, la plupart des équations de dynamique sont des équations de deuxième ordre. Nous avons donc intégré la résolution des équations différentielles de deuxième ordre dans Modelica. Nous ne pouvons pas utiliser la fonction *der()* de Modelica directement. En effet, la résolution de ce type d'équations n'est pas supportée par Modelica. Nous utilisons donc un changement de variable afin de transformer une équation de second ordre en deux équations de premier ordre.

Pour une équation de type a*der²(x) +b*der(x) +c*x = d, nous proposons le changement de variable suivant :

a*der(y) + b*y + c*x = d
der(x)=y

Dans le cas d'une situation comportant une équation différentielle de second ordre, il suffit d'implémenter ce changement de variables. Un test sur les paramètres de l'équation est effectué. L'implémentation de la résolution des équations différentielles de deuxième ordre est présentée dans la figure suivante :

```
// Résolution des équations

if x <= INV then
   if (a<>0) then
     a*der(y)+b*y+c*x = d;
     der(x)=y;
   else
      if b<>0 then
      der(x)= -c/b * x -d/b;
      else
           if c<>0 then
           c*x=d;
           else
           x =0;
           end if;
      end if;
   end if;
else
der(x)=  0;
der(y)=0;
end if;
```

Figure 5.10 Résolution des équations de deuxième ordre sous Modelica

Dans le cas où la situation comporte deux variables d'état ou plus, quelques modifications doivent être apportées à cette classe générique : déclaration de deux variables d'état ou plus, résolution des équations différentielles, modification des connecteurs en déclarant deux variables d'état ou plus.

La classe *state* traite le cas d'une situation d'un automate hybride avec une seule transition entrante et une ou plusieurs transitions sortantes. En effet, dans le bloc *Réinitialisation* nous effectuons un test sur la variable de propagation d'évènement provenant d'un seul connecteur d'entrée. Effectivement, nous ne pouvons pas connecter à un connecteur d'entrée plusieurs connecteurs équivalents d'autres classes Modelica sinon un problème d'écriture sur le connecteur d'entrée surviendra. Par contre, le connecteur de sortie de la classe state peut être connecté à plusieurs connecteurs d'entrée équivalents d'autres classes ou modèles Modelica.

Pour une situation avec deux transitions entrantes ou plus, nous créons pour chaque transition entrante un connecteur d'entrée qui lui correspond. Les transitions entrantes sont raccordées à la classe *state* via les connecteurs d'entrée correspondant. Au cours de la simulation, une seule transition est franchissable. Nous avons donc modifié la partie *Réinitialisation*. Nous effectuons un test pour chaque connecteur d'entrée pour vérifier si une transition est en train de propager un évènement d'activation. Dans le cas où une situation est raccordée par exemple à trois transitions entrantes nous modifions la partie *Réinitialisation* de la manière suivante :

```
  // Réinitialisation
when (pre(din1.numbevent)<> (din1.numbevent)) then
reinit(x,din1.s);
end when;

when (pre(din2.numbevent)<> (din2.numbevent)) then
reinit(x,din2.s);
end when;

when (pre(din3.numbevent)<> (din3.numbevent)) then
reinit(x,din3.s);
end when;
```

Figure 5.11 Une situation avec 3 transitions entrantes

```
model state "Situation d'un automate hybride\""

  // Déclaration des connecteurs d'entrée et sortie
  connector2 doutl;
  connector2 dinl;

  // Déclaration des variables d'état
  Real x;

  // Déclaration des variables pour le déclenchement des évènements
  Integer eventVisualizerl( start=0);
  Boolean state(start=false);

  // Déclaration des paramètres de l'équation différentielle
  parameter Real a= 2;
  parameter Real b= 0;
  parameter Real INV = 0.2 "Invariant";
  parameter Real Init = 0
    "Variable d'initialisation seulement pour le premier état. Pour les autres états Init =0";

equation
  // initialisation

  when initial() then
  reinit(x, Init);
  end when;

  // Réinitialisation
  when (pre(dinl.numbevent)<> (dinl.numbevent))  and not initial() then
  reinit(x,dinl.s);
  end when;

  // Résolution des équations

  der(x)= if x<=INV then   a*x + b else 0;

  // Propagation des variables d'état vers les transitions sortantes
  doutl.s = x;

algorithm

  // Vérification de l'invariant
  state :=false;
  when x>INV then
  state :=true;
  end when;

  // Déclenchement d'un évènement
  if state then
  eventVisualizerl:=eventVisualizerl + 1;
  end if;

  // Propopagation de l'évènement
  doutl.numbevent :=eventVisualizerl;

end state;
```

Figure 5.12 La classe générique s*tate*

5.3.7.2 La classe *Transition*

Nous décrivons la classe Transition dans la figure suivante :

```
model Transition

// Déclaration des paramètres de la Transition

  parameter Real Gard=0.9 " exemple: x<0.9";
  parameter Real INVetatCible=2.12
    "Condition sur l'invariant de l'état cible pour franchir la transition";
  parameter Boolean Vrai=false "Cette variable est utilisée quand la garde est toujours = VRAI
     (il faut l'initialiser à 0 si on a une garde sous forme de condition).";
  parameter Real Affectation "Variable d'affectation lors de la transition";

// Déclaration des connecteurs

  connector2 transdin;
  connector2 transdout;

algorithm
// Détection de la franchissabilité de la transition

  when (pre( transdin.numbevent) <>  transdin.numbevent) and  ((transdin.s < Gard) or Vrai) and (transdout.s < INVetatCible) then
  transdout.numbevent :=transdin.numbevent; // Propagation de l'évènement
  transdout.s :=Affectation; // Affectation au niveau de la transition
  end when;

end Transition;
```

Figure 5.13 La classe générique *Transition*

La classe *Transition* est divisée en 3 parties : Déclaration des paramètres de la transition, déclaration des connecteurs et la détection de la franchissabilité de la transition.

Déclaration des paramètres de la transition : une transition est caractérisée par sa garde. Nous déclarons un paramètre Gard. Nous supposons, que la condition de la garde est de la forme (variable d'état < Gard). Cette garde peut être modifiée pour l'adapter à d'autres formes de conditions. De même nous déclarons un paramètre représentant l'invariant de la situation cible qu'on a nommé INVetatcible. Comme pour le paramètre Gard, nous faisons la même supposition. Néanmoins, l'invariant de la situation cible peut être modifié afin de l'adapter à d'autres formes de conditions. Nous trouvons souvent dans les automates hybrides une garde sur une transition qui est toujours vraie. Nous déclarons pour cela un paramètre de type booléen pour traiter ce cas que nous appelons Vrai. Enfin, nous déclarons le paramètre Affectation de type réel pour l'affectation de la variable d'état quand la transition est franchissable.

Déclaration des connecteurs : on déclare deux connecteurs de type *connector2* (figure 5.9) : un connecteur d'entrée et un connecteur de sortie. Nous supposons dans

notre cas qu'il n'y a qu'une seule variable d'état. Dans le cas contraire, nous pouvons augmenter les connecteurs par d'autres variables.

Détection de la franchissabilité de la transition : cette partie est implémentée dans le bloc *algorithm*. Un test conditionnel est implémenté en utilisant l'expression *when*. Si un évènement de propagation est détecté, et si la condition de garde est vérifiée et la variable d'état est contenue dans l'invariant de l'état cible alors l'évènement de propagation est mis sur le connecteur de sortie ainsi que la variable d'affectation.

5.3.7.3 HybridAutomataLib

Les deux modèles présentés précédemment *state* et *Transition* composent la librairie HybridAutomataLib ainsi que le connecteur nommé *connector2*. OpenModelica n'implémente pas les spécifications complètes du langage Modelica. Nous avons utilisé les fonctions supportées par OpenModelica pour l'implémentation de cette bibliothèque. Ainsi, cette dernière peut être utilisée à la fois sous Dymola et OpenModelica[18]. Nous créons pour cela des interfaces graphiques pour chaque composant de la bibliothèque afin de les paramétrer facilement et avoir la possibilité de modéliser graphiquement et simuler des automates hybrides « simples ». Dans les figures suivantes nous présentons les modèles de la librairie HybridAutomataLib ainsi que leurs interfaces graphiques.

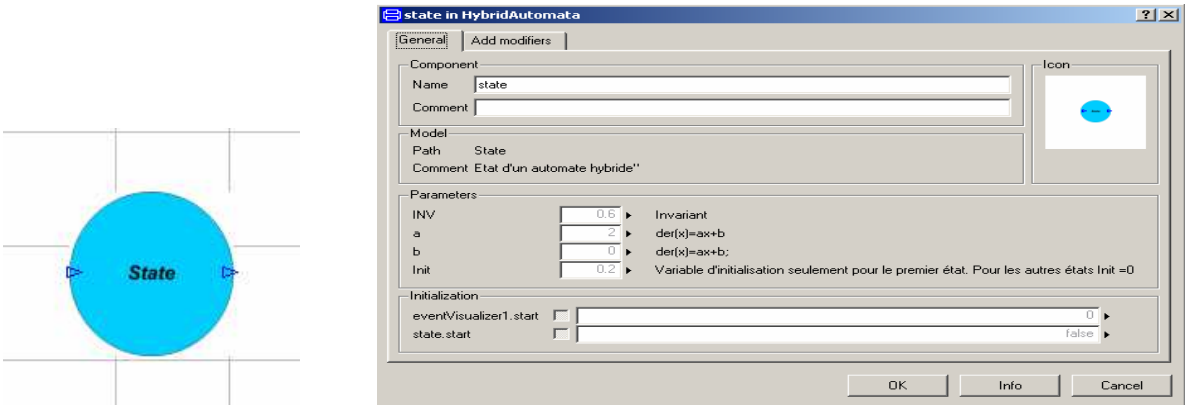


Figure 5.14 Le modèle *state* et son interface graphique sous Dymola

[18] Nous avons utilisé ici Dymola car OpenModelica ne permet pas encore une modélisation graphique.

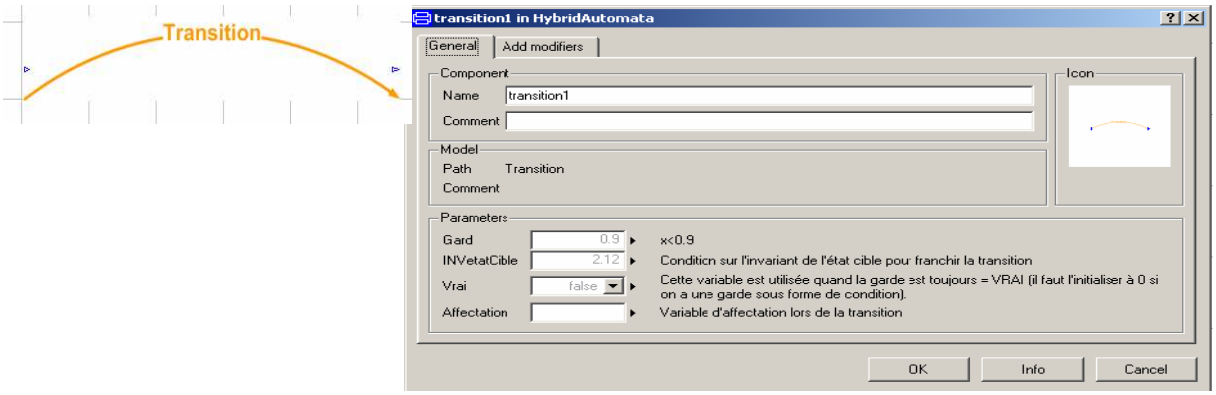


Figure 5.15 Le modèle *Transition* et son interface graphique sous Dymola

5.4 Conclusion

Nous avons présenté dans ce chapitre deux approches pour la conception de notre application. Nous avons fait le choix de la deuxième approche qui consiste à modéliser le comportement du système via des automates hybrides couplés qui représentent la partie commande et la partie opérative. L'aspect géométrique 3D est traité sous OpenMASK à travers l'animation des objets graphiques.

Pour mettre en œuvre cette approche, nous avons proposé une méthode générique pour implémenter les automates hybrides avec le langage Modelica. Nous avons proposé aussi une méthode qui permet de coupler deux automates hybrides avec Modelica. Cette méthode est basée sur la notion intéressante de Modelica des connecteurs. Nous avons par la suite proposé une bibliothèque Modelica qu'on a nommé HybridAutomataLib. Cette dernière permet de modéliser graphiquement un automate hybride comportant une équation différentielle de second ordre sous Dymola. Le code généré à partir de cette modélisation graphique peut être utilisé directement sous OpenModelica bien que ce dernier n'implémente pas complètement les spécifications du langage Modelica. Nous avons indiqué aussi les modifications à réaliser au niveau des modèles *Transition* et *state* pour les adapter dans les cas des automates hybrides avec plusieurs équations différentielles par situation.

6. Chapitre 6 : Intégration des services de HLA dans les simulateurs envisagés

6.1 Introduction

Dans le chapitre précédent, nous avons proposé une approche pour l'implémentation des automates hybrides avec Modelica. Une fois la partie contrôle et la partie opérative décrites par des automates hybrides couplés, on peut utiliser notre approche pour simuler le comportement du système. Une partie graphique peut être ajoutée à notre modélisation. L'utilisation d'un modèle 3D est nécessaire et son animation est assurée par le simulateur OpenMASK. La simulation du système doit donc supporter d'une part l'évolution dynamique représentée par des automates hybrides portant en particulier des équations électro-mécaniques et d'autre part sa représentation graphique 3D.

Nous rappelons que nous avons choisi dans le chapitre précédent (paragraphe 5.2) d'une part d'affecter à OpenMASK la représentation graphique 3D avec la possibilité ultérieure d'ajouter des interactions de type haptique supportées par cet environnement et d'autre part d'affecter à Openmodelica le comportement représenté par des automates hybrides.

Nous détaillons dans ce chapitre comment établir une communication entre les deux simulateurs en se basant sur l'architecture HLA. Pour cela, nous proposons une conception détaillée de la fédération représentant la simulation distribuée et des fédérés qui la composent. Nous nous basons sur les modèles de conception fournis par l'architecture HLA tels que le FOM (Federation Object Model) et le SOM (Simulation Object Model).

Afin de pouvoir faire communiquer les deux simulateurs OpenMASK et OpenModelica en se basant sur HLA nous devons les rendre compatibles à HLA. Quelques méthodes générales pour l'intégration des services HLA dans des simulateurs ont été présentées. Nous avons choisi parmi ces méthodes générales la solution la plus adéquate à chacun des deux simulateurs. Chaque solution est détaillée

en précisant les différents services HLA utilisés.

6.2 Conception de la fédération

Le FEDEP (Federation Development and Execution Process) est un outil méthodologique de conception générique d'une fédération HLA **[IEEE1516.3, 2003]**. Il ne peut être considéré comme solution universelle pour tous les développeurs d'HLA. Il est intéressant de l'utiliser dans des applications importantes, pouvant contenir plus de 100 fédérés **[Torpey, 2001]**, et complexes où la structuration prend une part importante. Le recours à cette méthode ne s'impose pas sur de petites applications. L'architecture HLA est composée par plusieurs règles de simulation de base définissant un fédéré, une fédération et leurs interactions à travers le RTI. Certaines règles sont moins restrictives que d'autres. Selon les règles 1 et 6, nous devons avoir un OMT (Object Model Template) pour décrire les objets et leurs interactions. La spécification d'un format standard pour l'OMT est une partie du standard HLA.

La simulation d'un système mécatronique, en général, consiste à simuler son comportement et l'animation 3D de ses composants mécaniques. Ainsi, nous pouvons identifier deux fédérés : un fédéré de comportement et un fédéré graphique pour l'animation des objets 3D. Ces deux fédérés forment une fédération dont le but est de simuler le comportement d'un système mécatronique et avoir une animation 3D de ses composants mécaniques. Les composants électroniques, quand à eux, peuvent être insérés dans la scène virtuelle comme étant des objets visuels statiques comme étant des parties du décor. Deux approches de conception sont présentées dans les paragraphes suivants :

6.2.1 Première approche

- Fédéré graphique : OpenMASK

Chaque composant mécanique du système mécatronique et qui intervient dans la simulation est décrit par un objet HLA au sein du fédéré OpenMASK. Chaque composant dynamique dans le système mécatronique a ses propres variables d'état. En effet, l'état dynamique d'un système est un état instantané et il est déterminé par les valeurs de toutes les variables d'état à cet instant. Nous définissons pour chaque objet HLA un ensemble de variables d'état lui correspondant.

- Fédéré de comportement : OpenModelica

Un système mécatronique est un système dynamique hybride. A chaque instant de la simulation, le système est déterminé par les valeurs de toutes ses variables d'état. Un système mécatronique est composé par plusieurs sous systèmes mécatroniques qui sont eux-mêmes composés par des composants élémentaires de différents domaines. Il est possible de décrire chaque composant du système par un Objet HLA. Vu que nous nous intéressons seulement aux variables d'état dynamiques, nous décrivons chaque composant mécanique par un objet HLA. Les attributs de ces objets sont les variables d'état leur correspondant.

Le fédéré OpenModelica doit publier ses attributs d'objets HLA avant le début de la simulation. Le fédéré OpenMASK, quand à lui, va souscrire aux différents attributs du fédéré OpenModelica. Au cours de la simulation, le fédéré OpenModelica met à jour les attributs de ses objets HLA en appelant le service HLA de publication. Ainsi, la RTI s'occupe de refléter les attributs mis à jour vers le fédéré OpenMASK.

6.2.2 Deuxième approche

Comme avec la première approche, on s'intéresse aux variables d'état concernant les composants mécaniques du système mécatronique. Cette deuxième approche consiste en l'utilisation des classes d'interaction au lieu des classes d'objets

HLA. En effet, on peut déclarer une ou plusieurs classes d'interactions avec différents paramètres dans le fédéré OpenModelica. Ainsi, pour une classe d'interaction on peut associer un ensemble de variables d'état des différents composants mécaniques en tant que paramètres de la classe d'interaction. Le fédéré OpenModelica doit publier les paramètres de la classe d'interaction. Le fédéré OpenMASK, quand à lui, doit s'inscrire à la classe d'interaction précédemment décrite ainsi qu'à ses paramètres. Au cours de la simulation, le fédéré OpenModelica envoie des interactions en utilisant le service HLA (Send Interaction). L'interaction est reçue par OpenMASK ainsi que ses paramètres. Les variables d'état représentés par les paramètres de l'interaction sont extraites et utilisées pour l'animation 3D.

6.2.3 Choix de l'approche

Nous choisissons d'utiliser la deuxième approche. En effet, on peut utiliser une seule classe d'interaction avec plusieurs paramètres pour représenter les variables d'état dynamique du système, au lieu d'utiliser plusieurs classes d'objet HLA en adoptant la première approche. Ceci rend la tâche de conception du fichier FED et l'implémentation des différents services moins ardus.

Les données communes des SOMs des différents fédérés permettent de produire le FOM de la fédération. Le FOM contiendra en conséquence les données échangées (les variables d'état dynamique du système) entre les deux fédérés. Ces informations seront partagées sous forme de classes d'interaction. Les paramètres des interactions entre les deux fédérés sont définis « TimeStamped Order » pour respecter le principe de causalité. Ces interactions sont donc émises avec une date associée au temps logique local du fédéré émetteur (OpenModelica) et sont stockées dans un échéancier du LRC avant d'être délivrées au souscripteur (OpenMASK) lorsque ce dernier sera temporellement en mesure de traiter ce message.

6.3 Méthodes générales d'intégration

Dans cette partie, nous rappelons les quatre méthodes générales d'intégration des services de HLA dans les simulateurs.

HLA a été développé essentiellement pour les simulations distribuées militaires. Il existe quelques applications visant à intégrer HLA dans des simulateurs industriels qui vont être présentées dans ce qui suit. Les difficultés apparaissent dès qu'on veut rendre les simulateurs industriels ou académiques compatibles avec HLA. En effet, les simulateurs militaires ont été développés comme des outils spécifiques où les fonctions HLA ont pu être implémentées dès le début du développement. Par contre, pour les simulateurs non militaires, les fonctions HLA n'ont pas été prévues. Quatre solutions générales peuvent être identifiées pour intégrer les outils de simulation dans l'architecture HLA. **[Straßburger, 1998]**

- Re-implémentation du simulateur avec les extensions HLA : Cette solution est la plus évidente si le code source est accessible ainsi qu'une bonne documentation. L'outil OpenMASK n'est malheureusement pas encore suffisamment documenté pour appliquer cette méthode. Quant à l'outil OpenModelica, malgré la disponibilité de son code source et de sa documentation, une telle technique reste très laborieuse si on n'est pas directement impliqué dans le développement de l'outil au sein de l'association Modelica.

- L'utilisation d'une interface de programmation externe : Cette solution convient aux outils qui offrent une architecture ouverte et extensible. L'outil doit fournir une interface de librairie (DLL pour Windows) avec la possibilité d'appeler des fonctions arbitraires ou des méthodes dans ces librairies. Nos outils ne disposent pas de cette interface.

- Extension du code intermédiaire: Plusieurs outils de simulation traduisent les descriptions des modèles écrites dans un langage propre à l'outil en un autre

langage de programmation (C++). Ce code intermédiaire est compilé en un fichier exécutable. Il est possible de modifier ce code pour réaliser les extensions HLA. Dans notre cas, une simulation sous OpenMASK est décrite par des classes C++ qui sont compilées en un fichier exécutable. Cette méthode nous a donc paru la plus appropriée pour intégrer les services de HLA dans OpenMASK. En revanche, cette méthode n'est pas appropriée pour OpenModelica qui fournit un code intermédiaire en C.

- Couplage avec une passerelle: La dernière solution pour les outils qui ne peuvent pas être connectés au RTI par l'une des méthodes précédentes est le développement d'une passerelle. Cette dernière doit communiquer avec l'outil de simulation à travers des fichiers, des pipes etc. Par élimination des précédentes méthodes, on a choisi cette méthode pour OpenModelica. Nous ne pouvons pas utiliser l'API Corba d'OpenModelica. En effet, plusieurs fonctions ne sont pas encore implémentées, par exemple des fonctions pour figer la simulation ou la relancer.

6.4 Intégration des services de HLA dans OpenModelica

OpenModelica est un outil à code source ouvert. Il fournit une API Corba qui permet d'interagir avec d'autres programmes ou applications. Nous ne pouvons pas utiliser cette API puisque plusieurs fonctions ne sont pas encore implémentées tels que les fonctions pour lire les valeurs de la simulation pendant l'exécution. Nous ne pouvons pas appeler les fonctions du RTI Ambassadeur directement à partir de OpenModelica puisqu'il se base sur le langage Modelica. OpenModelica génère du code C pour la simulation qui est exécutée. Nous ne pouvons pas intégrer les fonctions du RTI ambassadeur dans les fichiers C de la simulation. En effet, le code généré n'est pas lisible et est compliqué. Mélanger les deux langages C et C++ peut générer plusieurs erreurs d'incompatibilité.

Compte tenu de ces contraintes, nous proposons une passerelle pour communiquer avec le simulateur à travers les sockets **[Hadj-Amor (a), 2008]**. Cette

passerelle est implémentée en C++, ainsi nous avons pu intégrer les services de HLA pour interagir avec le RTI.

Il est possible d'appeler des fonctions implémentées en C ou FORTRAN et définies en dehors du langage Modelica. Le mot clé *external* est utilisé dans le corps de la fonction externe. L'utilisation des fonctions externes permet d'utiliser des fonctions ou des bibliothèques implémentés en C. Nous pouvons appeler ces fonctions externes à l'intérieur de OpenModelica en leur passant un ensemble d'arguments. Un utilisateur peut utiliser les fonctions externes de la même manière qu'il utilise une fonction non externe.

```
function log
input Real x;
output Real y;
external "C";
end log;
```

Figure 6.1 Exemple d'une fonction externe.

Nous allons nous baser sur cette propriété intéressante de Modelica pour échanger des données avec une interface externe à la simulation (une passerelle) en utilisant les *sockets*.

6.4.1 Les Sockets

Les *sockets* représentent une interface de programmation pour les communications entre processus (IPC : Inter Process Communication). Elles ont été introduites pour la première fois dans la version 4.3BSD (1986) de l'UNIX de l'université de **Berkeley**. Avant leur introduction, le seul mécanisme standard qui permettait à deux processus de communiquer se faisait par l'intermédiaire des pipes.

Une socket est un identifiant unique représentant une adresse sur le réseau. Des processus peuvent s'y connecter pour y envoyer des données ou pour en recevoir. Les processus devront adopter un protocole de communication afin d'assurer un échange de données cohérent. L'adresse du socket est spécifiée par le nom de l'hôte sur lequel on la crée et le numéro de port.

Les sockets sont généralement implémentés en langage C, et utilisent des fonctions et des structures disponibles dans la librairie <sys/socket.h>. L'un des avantages de l'interface socket est d'offrir une sémantique similaire à celle de l'accès aux fichiers (Utilisation de *read*, *write* et *close* une fois la connexion est établie). Deux protocoles peuvent être utilisés avec les sockets : Le protocole TCP (Transport Control Protocol) qui est un protocole dit « connecté » et le protocole UDP (User Datagram Protocol) qui est un protocole dit « non connecté ». Le protocole TCP est un protocole fiable, orienté connexion, qui permet l'acheminement sans erreur d'une machine à une autre. Le protocole TCP s'occupe également du contrôle de flux de la connexion. Le protocole UDP est en revanche un protocole plus simple que TCP : il est non fiable et sans connexion. Son utilisation présuppose que l'on n'a pas besoin ni du contrôle de flux, ni de la conservation de l'ordre de remise des paquets. Il est clair que le protocole TCP est plus fiable et semble le mieux adapté à notre application. Nous nous sommes appuyés sur le protocole TCP-IP qui se chargera d'assurer le transport des données (paquets) entre les processus serveur (la passerelle en C++) et client (la simulation sous OpenModelica). Dans ce contexte, le serveur est le processus qui écoute toute nouvelle connexion de client et effectue la récupération des données. Le client est donc le processus qui va tenter de se connecter au serveur et de lui envoyer des données. Il faut préciser que les processus communicant à travers les sockets ne sont pas nécessairement sur la même machine. Le serveur qu'on va développer doit être à l'écoute de messages éventuels. Toutefois, l'écoute se fait différemment selon que la socket est en mode connectée (TCP) ou non (UDP). Vu notre choix précédent, la communication entre le client et le serveur est une communication en mode ***connecté***.

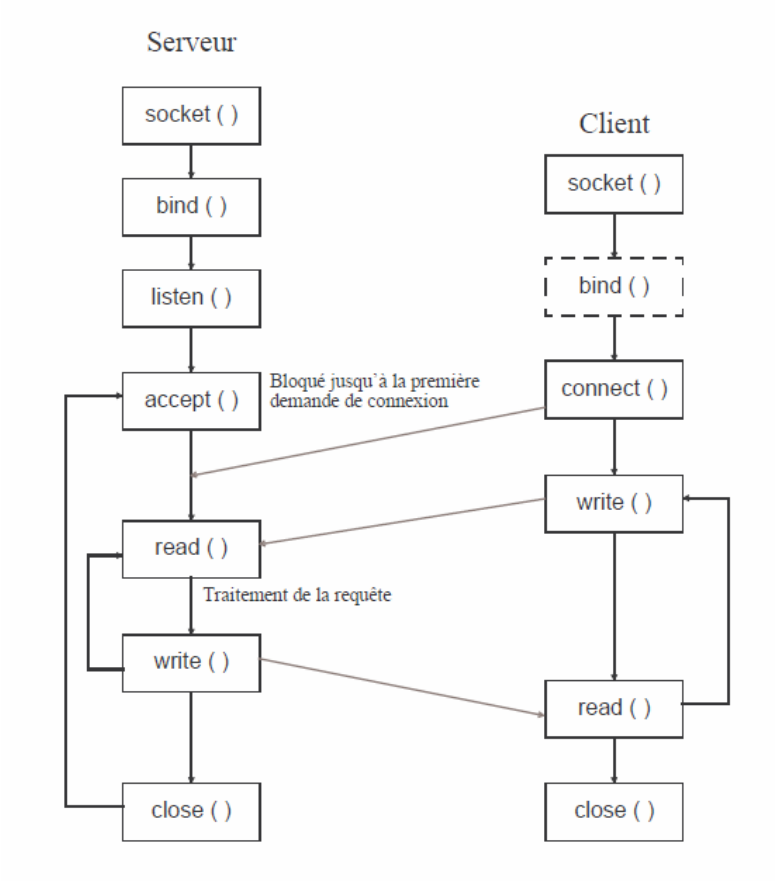


Figure 6.2 Communication entre client et serveur en mode connecté

Ainsi en mode connecté, la fonction *listen()* permet de placer la socket en mode passif (à l'écoute des messages). En cas de message entrant, la connexion peut être acceptée grâce à la fonction *accept()*. Lorsque la connexion a été acceptée, le serveur reçoit les données grâce à la fonction *recv()*. La fin de la connexion se fait grâce à la fonction *close()*.

Les différentes fonctions des sockets sont présentées dans les parties suivantes lors de la présentation des fonctionnalités de cette passerelle.

6.4.2 Communication entre OpenModelica et sa passerelle

La passerelle est implémentée comme un serveur TCP. La simulation fait appel aux fonctions externes en C pour se connecter à la passerelle et échanger des données avec elle. Une fois la connexion est établie, la simulation envoie des informations à la

passerelle (serveur). Cette dernière peut aussi envoyer des informations de retour à la simulation.

L'utilisation des sockets sous Modelica consiste en un fichier écrit en C et son fichier d'entête (.h). Nous avons créé cinq fonctions de base qui seront utilisées pendant le processus de communication. Ces fonctions déclarées dans le fichier d'entête et définies dans le fichier C :

- Fonction *createSocket(IPAddress, port)*: Avant de pouvoir échanger les données de la simulation, il faut établir la connexion entre le client et le serveur.
- Fonction *sendMessage (message)* pour envoyer un message où message est un string.
- Fonction *receiveMessage ()* pour recevoir un message où message est un string.
- Fonction *recMsgBlock()* pour figer la simulation sous OpenModelica.
- Fonction *clean()* pour fermer la connexion socket après la fin de la simulation.

Pour utiliser ces fonctions sous Modelica, nous utilisons des fonctions « enveloppantes ». Ainsi, nous avons créé une bibliothèque Modelica qu'on a appelé *CommunicationLib*. Cette bibliothèque contient une classe *socket* ainsi que les fonctions « enveloppantes » utilisant les fonctions externes définies en C.

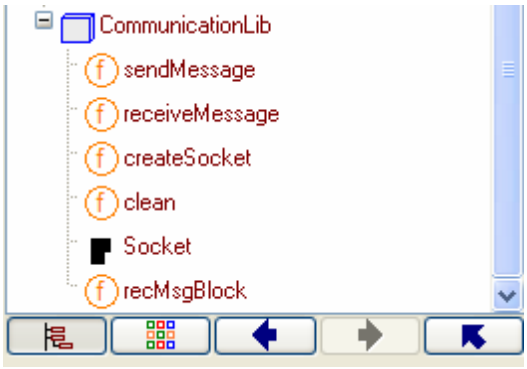


Figure 6.3 Vue de CommunicationLib dans l'explorateur de Dymola

Chaque fonction enveloppante correspond à une fonction externe. Dans le corps de la fonction enveloppante, nous précisons le chemin vers le fichier d'entête contenant la définition de la fonction externe ainsi que le chemin vers la librairie générée à partir du fichier C.

Nous allons présenter dans cette partie l'ensemble des fonctions enveloppantes utilisées au niveau de OpenModelica pour la communication avec la passerelle.

6.4.2.1 Les fonctions enveloppantes

La fonction createSocket() :

La fonction *createSocket()* contient deux paramètres: Le premier paramètre est l'adresse ip du serveur qui est une chaîne de caractères. Le deuxième paramètre est de type réel représentant le numéro du port. Cette fonction peut renvoyer ou non un résultat. Son rôle est de créer le socket sur l'ip donnée avec un numéro de port précisé. Nous avons choisi de retourner une valeur de type entier pour vérifier si la socket a bien été créée ou non. Nous précisons ensuite dans la déclaration de la fonction enveloppante le chemin précis vers la bibliothèque contenant l'implémentation de la fonction externe ainsi que le chemin vers le fichier d'entête.

```
function createSocket
  input String ip;
  input Real port;
  output Real z;
  external "C" annotation(Library="socket.a",Include="#include \"socketMe.h\"");
end createSocket;
```

Figure 6.4 Fonction enveloppante *createSocket()*

La fonction externe équivalente à la fonction enveloppante createSocket() doit avoir le même nom. Elle est implémentée en C. La création d'une socket se fait grâce à la fonction *socket()*. La fonction *connect()* permet d'établir une connexion avec le serveur.

int socket (int *famille*, int *type*, int *protocole*) ;

- L'entier *famille* désigne la famille de protocoles qu'on veut utiliser. (*AF_INET* pour TCP/IP utilisant une adresse Internet sur 4 octets : l'adresse IP ainsi qu'un numéro de port afin de pouvoir avoir plusieurs sockets sur une même machine, *AF_UNIX* pour les communications UNIX en local sur une même machine).
- L'entier *type* désigne le type de la socket. Dans le cas d'une communication en mode connecté, l'argument *type* doit prendre la valeur SOCK_STREAM (communication par flot de données).
- L'entier *protocol* indique le protocole à utiliser sur la socket. Normalement il n'y a qu'un seul protocole par type de socket pour une famille donnée, auquel cas l'argument *protocol* peut être nul.

La fonction *socket()* retourne un descripteur référençant la socket créée en cas de réussite. En cas d'échec la valeur -1 est renvoyée.

int connect(int *sockfd*, struct sockaddr * *servaddr*, int * addrlen) ;

- Le paramètre *sockfd* est une socket.
- *servaddr* représente l'adresse de l'hôte à contacter. Le client ne nécessite pas de faire appel à la fonction bind() pour établir une connexion.
- *Addrlen* représente la taille de l'hôte à contacter. Ce paramètre peut être remplacé par l'expression suivante utilisant la fonction sizeof() : *sizeof (servaddr).*

La fonction *connect()* renvoie 0 si elle réussit, ou -1 si elle échoue.

La combinaison de ces fonctions nous permet de créer une fonction *createSocket()* en language C. Cette fonction peut être appelée à partir d'un modèle sous Modelica en utilisant une fonction enveloppante ayant le même nom et les mêmes paramètres.

```
//create a socket to IP-Address and port
double createSocket(const char* IP_ADDRESS, double port)
{

   sockfd=socket(AF_INET,SOCK_STREAM,0);

   bzero(&servaddr,sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr=inet_addr(IP_ADDRESS);
   servaddr.sin_port=htons(port);

   connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
   return 1;
}
```

Figure 6.5 Fonction externe *createSocket()*[19]

La fonction sendMessage() :

Pour pouvoir utiliser la fonction *sendMessage()* nous devons changer la valeur à envoyer en une chaîne de caractères (string). Ceci est fait en faisant appel à la fonction *String()* de Modelica. Chaque valeur des variables de la simulation est concaténée avec le nom de son attribut. Le tout est envoyé en faisant appel à la fonction *sendMessage()*. Nous choisissons un pas de temps fixe pour envoyer et recevoir des données.

```
function sendMessage
  input String texte;
  output Real z;
  external "C" annotation(Library="socket.a",Include="#include \"socketMe.h\"");
end sendMessage;
```

Figure 6.6 Fonction enveloppante *sendMessage()*

L'envoi de messages se fait à l'aide de la fonction *send()*. Elle est utilisée seulement en mode connecté. On utilise la fonction *sendto()* pour le mode non connecté. (info)

int send(int *s***, const void ****msg***, size_t** *len***, int** *flags***);**

[19] La version 1.4.4 de OpenModelica n'implémente pas les spécifications complètes du langage Modelica. Dans cette version, nous sommes obligés d'utiliser une valeur renvoyée de la fonction C externe de type *double* et par conséquent une valeur de retour dans la fonction enveloppante Modelica de type *Real*.

- *s* représente le socket précédemment ouvert.
- *msg* représente un tampon mémoire contenant les octets à envoyer au client.
- Le champ *flag* contient des indicateurs servant à typer les données. Par exemple, on a le flag MSG_OOB qui indique que les données sont des données urgentes. Si ce champ n'est pas utilisé on peut utiliser la fonction *write* au lieu de *send*.

La fonction externe équivalente à la fonction enveloppante *sendMessage()* doit avoir le même nom et les mêmes paramètres et elle est implémentée en C. Elle est décrite dans la figure suivante.

```
//send a string through socket
double sendMessage(const char* buf)
{

        send(sockfd,buf,strlen(buf),0);
        return 1;

}
```

Figure 6.7 Fonction externe sendMessage()

La fonction receiveMessage() :

```
function receiveMessage
  "receive a message through non blocking socket"
output Real z;
external "C" annotation(Library="socket.a",Include="#include \"socketMe.h\"");
end receiveMessage;
```

Figure 6.8 Fonction enveloppante *receiveMessage()*

La fonction *receiveMessage()* est utilisée pour pouvoir récupérer des données envoyées par la passerelle (le serveur). Nous traitons les données reçues dans la fonction externe et non dans la fonction enveloppante. En effet, vu que dans notre application la simulation sous OpenModelica ne reçoit pas des données de la part de OpenMASK, il n'est pas nécessaire de créer alors une fonction *receiveMessage()* qui retourne une valeur. Les seules données qui peuvent êtres reçues par la passerelle sont

des messages d'activation ou de désactivation de la simulation. Ses messages servent pour établir une synchronisation entre OpenModelica et la passerelle d'un côté et entre la passerelle et le RTI de l'autre côté. Cette partie concernant la synchronisation entre les différents simulateurs est plus détaillée dans le paragraphe 7.2.

La simulation sous OpenModelica peut recevoir deux sortes de messages : message demandant à la simulation de s'arrêter et message demandant à la simulation de reprendre son exécution. L'envoi des messages est implémenté dans la passerelle que nous allons présenter par la suite. Le message de demande d'arrêt contient la chaîne de caractère « stop ». Celui de demande de la reprise de la simulation contient la chaîne de caractère « wake ». La fonction *receiveMessage()* utilise la fonction système *recv()* sur une socket non bloquante. Les fonctions *recv()* et *fcntl()* sont décrites dans le paragraphe présentant la fonction *recMsgBlock()*.

La fonction *receiveMessage()* est implémentée comme suit :

```
//receive a string through non blocking socket
double receiveMessage(const char* nein)
{

        x=0;
        fcntl(sockfd, F_SETFL, O_NONBLOCK);

         if (recv(sockfd,receiveBuffer,100,0)>0 && receiveBuffer[0] == 's')
            {
            x= x+1;
            }
         if (recv(sockfd,receiveBuffer,100,0)>0 && receiveBuffer[0] == 'w')
            {
            x= x-2;
            }
         return x;
}
```

Figure 6.9 Fonction externe *receiveMessage()*

La fonction *receiveMessage()* renvoie 1 si elle reçoit la chaîne de caractère « stop » et renvoie -2 si elle reçoit la chaîne de caractère « wake ». La valeur de retour est utilisée dans la simulation sous OpenModelica pour figer la simulation ou la relancer.

La fonction recMsgBlock() :

La fonction recMsgBlock() est une fonction intéressante et qui joue un rôle primordial dans notre application. Cette fonction permet de figer la simulation sous OpenModelica pendant un intervalle de temps. En effet, la fonction recMsgBlock() attend la réception d'un message contenant une chaîne de caractère bien précise. Cette attente est bloquante et la simulation est ainsi figée. A la réception du message attendu, la simulation reprend de nouveau.

```
function recMsgBlock "receive a message through a blocking socket"
output Real y;
external "C" annotation(Library="socket.a",Include="#include \"socketMe.h\"");
end recMsgBlock;
```

Figure 6.10 Fonction enveloppante *recMsgBlock()*

L'idée pour développer cette fonction se base sur la manipulation des descripteurs de socket en utilisant la fonction *fcntl()*. En utilisant cette dernière, nous modifions les attributs de notre socket déjà déclarée précédemment. Nous pouvons ainsi changer l'attribut O_NONBLOCK par ~O_NONBLOCK. Ce changement est effectué en utilisant la même fonction *fntl()* et en utilisant le paramètre F_SETFL qui permet de fixer de nouveaux attributs pour un descripteur de fichier. Une fois que la socket est devenue bloquante, nous appelons la fonction recv() qui permet de lire dans une socket en mode connecté (TCP).

int recv(int *socket*, char * *buffer*, int *len*, int *flags*)

- *socket* représente la socket précédemment ouverte.
- *buffer* représente un tampon qui recevra les octets en provenance du client.
- *len* représente le nombre d'octets à lire.
- *flags* correspond au type de lecture à adapter. Il est constitué par un OU binaire entre une et plusieurs des valeurs suivantes :

MSG_OOB : permet la lecture des données hors-bande qui ne seraient autrement pas placées dans le flux de données normales.
MSG_PEEK : permet de lire les données en attente dans la file sans les enlever de cette file.
MSG_WAITALL : demande que l'opération de lecture soit bloquée jusqu'à ce que la requête complète soit satisfaite.
MSG_NOSIGNAL : désactive l'émission de **SIGPIPE (définition de SIGPIPE en bas de page)** sur les sockets connectées dont le correspondant disparaît.
MSG_TRUNC : Renvoie la longueur réelle du paquet, même s'il était plus long que le buffer transmis.
MSG_ERRQUEUE : Cet attribut demande que les erreurs soient reçues depuis la file d'erreur de la socket.
0 : Le *flag* 0 indique une lecture normale.

La fonction *recv()* renvoie le nombre d'octets reçus si elle réussie ou -1 si elle échoue.

Pour implémenter la fonction recMsgBlock() nous suivons les étapes suivantes :

- On modifie les attributs de la socket déclaré précédemment en utilisant la fonction *fcntl()* pour la rendre bloquante.
- On appelle la fonction recv() sur notre socket bloquante.
- A la réception d'un message, on le compare à un paramètre (chaîne de caractère) déjà prédéfini pour savoir si le message a pour but de débloquer la simulation ou pas. Si la comparaison est égale alors la fonction recMsgBlock() renvoie 1, sinon elle renvoie 0.

La fonction *recMsgBlock()* renvoie une valeur double (soit 0 soit 1). Elle prend comme paramètre une variable de type chaîne de caractère muette. Cette variable n'intervient pas dans la fonction *recMsgBlock()*. Cette dernière est implémentée comme suit :

```
//receive a string through blocking socket
double recMsgBlock(const char* neine)
{
        x=0;
        int flags;


        flags = fcntl (sockfd, F_GETFL);
        fcntl (sockfd, F_SETFL, flags & ~O_NONBLOCK);
         if (recv(sockfd,receiveBuffer,100,0)>0 && receiveBuffer[0] == 'w')
            {
            x= x+1;
            }
         return x;
}
```

Figure 6.11 La fonction externe *recMsgBlock()*

Dans le modèle Modelica, l'appel de la fonction recMsgBlock() peut être utilisé dans une boucle *when* pour maintenir le blocage de la simulation ou la débloquer. Nous montrerons par la suite une combinaison entre la fonction receiveMessage() et recMsgBlock() pour figer la simulation et la débloquer suivant les messages reçus de la passerelle (serveur TCP).

La fonction clean( ) :

La fonction *clean()* est une fonction enveloppante en Modelica. Elle fait appel à une fonction externe développée en C. Le rôle de cette fonction est de fermer la socket. Elle est appelée à la fin de la simulation sous OpenModelica pour interrompre la connexion établie entre la passerelle et OpenModelica et fermer la socket.

```
function clean "close the socket"
output Real y;
external "C" annotation(Library="socket.a",Include="#include \"socketMe.h\"");
end close;
```

Figure 6.12 Fonction enveloppante *clean()*

La fonction externe *clean()* utilise la fonction close() pour fermer la socket et interrompre la connexion. En effet, la fonction *close()* permet la fermeture d'une soket.

int close(int *sockfd*);

La fonction *close()* ferme le descripteur du socket *sockfd* et libère ses ressources s'il n'est plus partagé. Quand il reste des données dans un SOCK_STREAM/AF_INET, le système essaie d'acheminer ces données. Dans ce cas, la primitive de fermeture peut être rendu bloquante. La fonction *close()* renvoie 0 s'il réussit, ou -1

L'implémentation de la fonction externe est comme suit :

```
//Close the socket
double clean()
{

        close(sockfd);
        return 1;
}
```

Figure 6.13 Fonction externe *clean()*

Nous choisissons une valeur de retour égale à 1 pour la fonction *clean().*

6.4.2.2 Les modèles Modelica d'intégration

Le modèle (classe) Socket

La classe ou modèle *Socket* est paramétrable. On peut modifier l'adresse ip ou le port. Dans notre, cas nous utilisons l'adresse 127.0.0.1 (*localhost*) puisque OpenModelica et sa passerelle s'exécutent sur la même machine. Nous choisissons ensuite un port quelconque. L'adresse ip est un paramètre de type chaîne de caractère (String) alors que le port est un paramètre de type réel (Real). Le modèle *Socket* est utilisé par les simulations interagissant avec des programmes externes via une socket. Au début de l'exécution de la simulation, la socket doit être créée et connectée. Ainsi, nous choisissons d'appeler la fonction *createSocket()* dès que la simulation se lance. Pour détecter le début de la simulation, nous utilisons la fonction Modelica *initial()* dans une boucle *when*.

Une instance de ce modèle peut être utilisée dans d'autres modèles Modelica. Au lancement de la simulation, une socket est créée et une connexion est établie sur les paramètres *ip* et *port*. Le modèle Socket est présenté dans la figure suivante.

```
model Socket
Real a;
parameter String ip = "127.0.0.1";
parameter Real port=12345;

equation
when (initial()) then
a=createSocket(ip,port);
end when;

end Socket;
```

Figure 6.14 Modèle Socket

Le modèle RemoteController

Le modèle *RemoteController* est le point de connexion entre OpenModelica et la passerelle. En effet, ce module peut être utilisé pour connecter OpenModelica à des programmes externes qui peuvent communiquer via sockets. Le modèle *RemoteController* est une combinaison des fonctions externes et des modèles présentés précédemment.

Le modèle *RemoteController* peut avoir un ou plusieurs connecteurs d'entrée pour le relier à d'autres modèles Modelica. Le nombre de ces connecteurs dépend du nombre des variables à communiquer au programme externe via les sockets. Nous n'avons pas besoin de connecteurs de sortie pour relier le *RemoteController* à d'autres modèles Modelica sauf dans le cas où on a besoin de communiquer des données externes reçues par les sockets à un modèle interne sous OpenModelica. Dans le modèle *RemoteController*, on crée une instance du modèle Socket avec les paramètres IPAdress et Port voulus. Le fait de créer cette instance permet de créer la socket et d'établir une première connexion entre OpenModelica et la passerelle. Nous

choisissons un pas de temps fixe et de durée minime pour envoyer les données de la simulation via la socket. L'échantillonnage et l'envoi sont effectués à l'aide de la fonction *sample()* de Modelica dans le bloc *equation*.

```
Boolean SampleEvent;
Real a(start=0);

equation
SampleEvent = sample(0,0.01);
when SampleEvent then
a= sendMessage(String(din));
end when;
```

Figure 6.15 Echantillonnage et envoie de données

La fonction externe *sendMessage()* est appelée à chaque pas de temps en lui passant comme paramètre un message. Ce message contient une valeur d'une variable d'état de la simulation convertie en chaîne de caractère. La conversion se fait à l'aide de la fonction Modelica *String()*. Quand à la fonction *receiveMessage()* elle est appelée dans le bloc *equation* et en dehors de la boucle d'échantillonnage. En effet, la simulation doit réagir à n'importe quelle interaction avec un programme externe et non à chaque pas de temps fixe. Ceci implique que la fonction *receiveMessage()* est à l'écoute des messages de la passerelle à chaque instant de l'exécution de la simulation. A la réception d'un message d'arrêt la fonction *receiveMessage()* renvoie 1. A la réception d'un message de reprise d'exécution elle renvoie -2. Dans tous les autres cas, elle renvoie 0. Suivant la valeur renvoyée de cette fonction, nous implémentons une boucle *when* dans le bloc *algorithm* qui va permettre de figer la simulation. Dans cette boucle nous combinons les deux fonctions *receiveMessage()* et *recMsgBlock()*.

```
Real value;

Real blockactiv(start=0);

equation

value = receiveMessage(String(din));


algorithm

when value >=1 then

blockactiv:= recMsgBlock(String(din));

end when;
```

Figure 6.16 Boucle de blocage et de déblocage de OpenModelica

Si la valeur *Value* est égale à 1 (réception d'une demande de blocage), la boucle *when* est alors active. La fonction recMsgBlock() est bloquante. La simulation est donc figée jusqu'à réception d'un message de reprise de la simulation. (Voir fonction *recMsgBlock()*). Une fois le message est reçu, la boucle *when* n'est plus active. Après la reprise de la simulation, la valeur *Value* demeure égale à 0 tant qu'aucun message de la passerelle n'est reçu. Ainsi la boucle *when* dans le bloc *algorithm* reste non active.

Ainsi, nous avons un modèle qui sert d'outil de connexion aux programmes externes à OpenModelica. Ce modèle peut être connecté à n'importe quel autre modèle Modelica. En effet, une simulation sous OpenModelica utilisant les composants de base de la bibliothèque Modelica peut utiliser le modèle *RemoteController* pour interagir avec un programme externe.

6.4.2.3 La passerelle

La passerelle est un programme implémenté en C++ qui permet la communication entre la simulation sous OpenModelica et le RTI. Nous avons choisi de l'implémenter en C++ puisque l'implémentation de la RTI utilisée est en C++, qui est le CERTI.

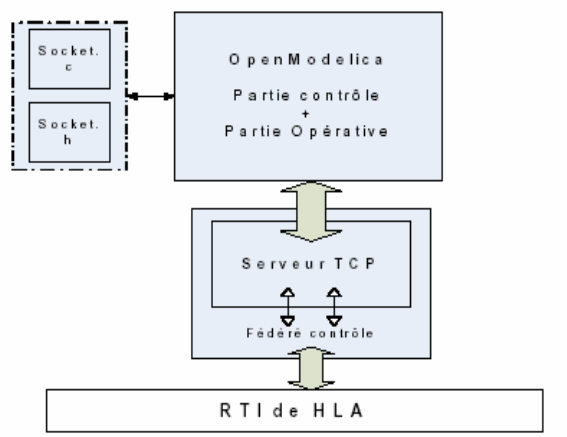


Figure 6.17 Serveur TCP / OpenModelica

La passerelle est divisée en deux parties majeures: une partie qui implémente le serveur TCP et une autre partie qui implémente la boucle de la simulation. En effet, de point de vue HLA, la passerelle joue le rôle d'un fédéré.

Implémentation du serveur TCP

Nous suivons l'organigramme présenté dans la figure 6.2 pour établir le serveur. Comme pour le cas du client implémenté sous OpenModelica, une socket est créée grâce à la fonction *socket()*. Après la création de la socket, il s'agit de la lier à un point de communication définie par une adresse et un port. Ceci est le rôle de la fonction *bind()*.

int bind(int *sockfd***, struct sockaddr ****my_addr***, socklen_t** *addrlen***);**

bind() fournit à la socket *sockfd*, l'adresse locale *my_addr*. *my_addr* est longue de *addrlen* octets. Traditionnellement cette opération est appelée "affectation d'un nom à une socket". La fonction *bind()* renvoie 0 s'il réussit ou -1 s'il échoue.
Ensuite pour mettre la socket créée en attente de connexion, nous appelons la fonction listen(). En effet, la connexion listen() n'est utilisée qu'en mode connecté et donc avec le protocole TCP.

int listen(int *socket***, int** *backlog***);**

Le paramètre *socket* représente la socket précédemment créée. Le backlog représente le nombre de connexions pouvant être mises en attente. La fonction listen() renvoie 0 si elle réussit ou -1 en cas d'échec.
Une fois que la socket est en mode d'écoute, elle doit accepter les demandes de connexion au serveur. Pour cela nous faisons appel à la fonction *accept()*.

int accept(int *socket***, struct sockaddr ****adresse***, socklen_t ****longueur***);**

Le paramètre *socket* représente la socket précédemment ouverte. Le paramètre *adresse* représente un tampon destiné à stocker l'adresse de l'appelant. Le paramètre longueur représente la taille de l'adresse de l'appelant. L'appel renvoie -1 en cas d'erreur. S'il réussit il renvoie un entier non-négatif, constituant un descripteur pour la nouvelle socket.

La réception des données est implémentée dans la boucle de la simulation, nous utilisons la fonctions *recv()* présentée précédemment.

Implémentation de la boucle de la simulation
Pour établir une communication avec le RTI, nous concevons deux parties. Une première partie qui invoquent les services HLA pour envoyer des données au RTI. Cette partie du fédéré est appelée LRC (Local RTI Component). La deuxième partie capte les évènements et les messages envoyés par le RTI et les traduit en des requêtes

TCP à destination du modèle Modelica. Cette deuxième partie du fédéré est appelée FedCode. Le LRC fait appel aux méthodes de la classe RTIAmbassador tandis que le FedCode fait appel aux méthodes de la classe FederateAmbassador.

Pour pouvoir invoquer les services de HLA, tout fédéré doit donc créer une instance de la classe RTIAmbassador et une instance de la classe FederateAmbassador. Les services de la classe RTIAmbassador constituent la librairie fournie par la distribution du RTI. Ils n'ont pas à être codés par le concepteur. En revanche, les services de la classe FederateAmbassador doivent être renseignés par le concepteur du fédéré.

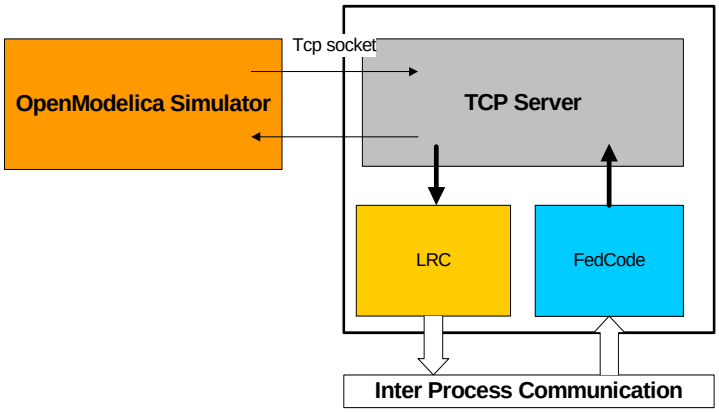


Figure 6.18 Fédéré OpenModelica

LRC

Pour créer la fédération et la rejoindre nous faisons appel aux services createFederationExecution et joinFederationExecution de la classe RTIAmbassador. Pour cela, nous créons une classe qui hérite de la classe NullFederateAmbassador. Dans cette classe, nous déclarons une instance du RTIAmbassador qu'on a nommé *rtiamb*. Nous appelons cette classe *chat.* Elle est formée par deux fichiers: un fichier C++ nommé chat.cc et un fichier d'entête nommé chat.h. Nous implémentons une fonction membre de la classe chat qu'on nomme join(). Elle permet de créer et de rejoindre la fédération en utilisant le deux services createFederationExecution et joinFederationExecution du RTIAmbassador. La fonction *join()* prend comme

paramètre le nom de la fédération et le nom du fichier FED. Elle est implémentée de la manière suivante.

```
void
chat::join(string federation_name, string fdd_name)
{
    federationName = federation_name ; // A définir dans chat.hh

    // create federation
    try {
        rtiamb.createFederationExecution("hassen",
                                         fdd_name.c_str());
        cout << "Federation execution created." << endl;
        creator = true ;
    }
    catch (RTI::FederationExecutionAlreadyExists& e) {
        cout << "Federation execution already created." << endl;
    }

    // join federation
    bool joined = false ;
    int nb = 5 ;

    while (!joined && nb > 0) {
        nb-- ;
        try {
            handle = rtiamb.joinFederationExecution(federateName.c_str(),
                                                    federation_name.c_str(),
                                                    this);
            joined = true ;
            break ;
        }
        catch (RTI::FederateAlreadyExecutionMember& e) {
            cout << "Federate " << federateName.c_str()
                        << "already exists." << endl ;

            throw ;
        }
        catch (RTI::FederationExecutionDoesNotExist& e) {
            cout << "Federate : FederationExecutionDoesNotExist." << federateName.c_str() << endl;

        }
        catch (RTI::Exception& e) {
            cout << "Federate" << federateName.c_str() << " :Join Federation Execution failed" << endl;
            throw ;
        }
    }
}
```

Figure 6.19 La fonction join()

La gestion des erreurs se fait à chaque appel de l'un des services HLA. Ainsi, quand une erreur se produit lors de l'exécution, un message indiquant l'erreur apparaît. L'appel de la fonction *join()* est effectué au sein de la passerelle et en première étape de la manière suivante:

```
// Federation and .fed names
string federation= "Passerelle";
string fedfile= "Chat.fed";
char federate_name [40];

        cout << "give a name to the federation" << endl;
        cin.getline (federate_name, sizeof (federate_name));
        chat chatting(federate_name);


// Join and create federation
        cout << "Create or join federation" << endl ;
        chatting.join(federation, fedfile);
```

Figure 6.20 Appel de la fonction *join()* depuis la passerelle

La deuxième étape effectuée dans le module RLC est de parcourir les objets et interactions HLA de la fédération et récupérer leurs identifiants. Ceci est effectué en appelant la fonction getHandle() immplémentée comme fonction membre de la classe chat. L'appel se fait de la manière suivante.

```
RTI::FederateHandle handle = chatting.getHandle();
```

Figure 6.21 Appel de la fonction *getHandle()*

La fonction *getHandle()* renvoie l'ensemble des identifiants (*handles*) des objets, attributs, interactions et paramètres HLA (HLA items). La fonction *getHandle()* est implémentée en appelant les services du RTIAmbassador *getObjectClassHandle*, *getAttributeHandle*, *getInteractionClassHandle* et *getParameterHandle*. L'implémentation de la fonction *getHandle()* varie suivant le nombre d'items HLA dans la fédération. Un exemple de l'implémentation de cette fonction est le suivant:

```
// ------------------------------------------------------------------------
/** get handles of objet/interaction classes
 */
void
chat::getHandles()
{
    cout << "Get handles..." << endl ;
    ParticipantClassID = rtiamb.getObjectClassHandle(CLA_PARTICIPANT);
    cout << "ParticipantClassID " << ParticipantClassID << endl;

    // Attributs des classes d'Objets
    AttrNameID = rtiamb.getAttributeHandle(ATT_NAME, ParticipantClassID);

    cout << "AttrNameID" << AttrNameID << endl;

    // Interactions
    CommunicationClassID = rtiamb.getInteractionClassHandle(INT_COMMUNICATION);

    ParamMESSAGEID = rtiamb.getParameterHandle(PAR_MESSAGE, CommunicationClassID);
    ParamUSERID = rtiamb.getParameterHandle(PAR_SENDER, CommunicationClassID);
   cout << "CommunicationClassID =" << CommunicationClassID <<
   "Message_ID =" << ParamMESSAGEID << "User_ID =" << ParamUSERID << endl;


}
```

Figure 6.22 Exemple d'implémentation de la fonction *getHandle()*

La tache suivante à effectuer dans la partie LRC est de s'abonner aux objets et interactions qui intéressent le fédéré et de publier les objets et les interactions propres au fédéré. Pour cela, nous avons crée une fonction *publishandSubscribe()* membre de la classe *chat*. L'appel de cette fonction se fait de la manière suivante:

```
// Publish and subscribe
        chatting.publishAndSubscribe();
```

Figure 6.23 Appel de la fonction *publishAndSubscribe()* dans la passerelle

L'implémentation de cette fonction se fait en appelant les services du RTIAmbassador pour souscrire à une classe et ses attributs subscribeObjectClassAttributes, ou pour publier les attributs d'un objet HLA en utilisant le service publishObjectClass, ou pour publier et souscrire à des interactions en utilisant les services: subscribeInteractionClass et publishInteractionClass.

La tâche avant dernière dans la partie LRC est de mettre à jour des attributs des objets HLA ou d'envoyer des interactions HLA aux autres fédérés. Nous avons choisi d'utiliser arbitrairement les interactions. Ainsi, nous avons déveoppé une fonction membre de la classe *chat* appelée sendMessage(). Cette fonction est exécutée à chaque fois que la passerelle reçoit des nouvelles valeurs de la simulation sous OpenModelica.

```
while (recv(connectSocket, line, MAX_MSG, 0) > 0) {
cout << line <<"\n";
//chatting.step(); fonction décrite dans le chapitre Gestion du temps
chatting.sendMessage(line, federate_name);
memset(line, 0x0, LINE_ARRAY_SIZE);  // set line to all zeroes
cout.flush();
}
```

Figure 6.24 Boucle de simulation de la passerelle

Il faut préciser qu'à la réception du message contenu dans la variable *line*, il faut décrypter le nom de la variable pour utiliser l'interaction qui lui est associée. Ainsi, nous devons créer pour chaque variable de simulation reçue une fonction *sendMessage()* qui fait appel à la classe d'interaction HLA associée à cette variable. Dans le cas où on a *n* variables nous devons créer *n* classes d'interaction de sorte que chaque variable a une classe d'interaction qui lui est associée.
La fonction *sendMessage()* est implémentée en faisant appel au service HLA du RTIAmbassador sendInteraction.

Enfin, la dernière étape dans la partie LRC concerne la fin de la simulation. On fait appel aux services HLA *resignFederationExecution()* pour quitter la fédération et *destroyFederationExecution()* pour la détruire.

D'autres fonctions sont implémentées dans la partie LRC. Elles seront présentées dans le chapitre Gestion du temps et contraintes temporelles. En effet, ces fonctions permettent la synchronisation du fédéré avec les autres fédérés et permettent aussi d'adapter une stratégie d'avancement du temps du fédéré.

FedCode

La partie du FedCode consiste à récupérer les messages envoyés par le RTI vers la passerelle. Certains de ces messages doivent être transformés par des messages vers la simulation sous OpenModelica via la socket. Nous avons présenté dans la partie LRC la classe chat qui hérite de la classe FederateAmbassador. Les méthodes de la classe FederateAmbassador sont les services invoqués par la RTI pour diffuser des informations issues de la fédération vers le fédéré. Ces derniers services sont communément appelés « callbacks ». L'appel des callbacks par la RTI sert à transmettre des données au fédéré « La passerelle ». Ces données peuvent êtres des mises à jour d'attributs, des interactions envoyées par d'autres fédérés, une demande de synchronisation, une demande d'avancement dans le temps...etc.

Dans notre application, nous nous limitons à envoyer les valeurs de la simulation vers le fédéré OpenMASK sans attendre un retour de données de ce dernier. Dans ce cas, un seul callback a été implémenté. Il s'agit du callback announceSynchronizationPoint() qui annonce au fédéré un point de synchronisation commun à tous les fédérés dans la fédération. Ce callback est présenté dans le chapitre 7.

6.4.3 Description de la dynamique

Une socket serveur est créée et, elle est à l’écoute des demandes de connexion. Si une demande de connexion arrive, une connexion est établie et le serveur peut recevoir les données de la simulation sous OpenModelica. Chaque donnée reçue est décryptée, le nom de la variable est extrait ainsi que sa valeur. Ensuite les valeurs de la simulation sont envoyées à travers les services HLA au RTI.

6.5 Intégration des services de HLA dans OpenMASK

Pour intégrer les services HLA dans le simulateur OpenMASK nous avons opté pour la première solution présentée précédemment (paragraphe 6.3) dans les méthodes générales d'intégration de HLA dans les simulateurs **[Hadj-Amor, 2006]**. Cette méthode semble la plus évidente puisque le code de OpenMASK est accessible. Une application sous OpenMASK est un ensemble de modules C++ qui interagissent. Ces modules sont compilés avant d'être exécutés. L'idée est d'intégrer les services de HLA dans un ou plusieurs de ces modules avant de les compiler. Etant donné que le bus HLA utilisé est le CERTI, qui est une bibliothèque codée en C++, nous n'avons pas un problème de comptabilité.

Une simulation sous OpenMASK est organisée dans un arbre de simulation composé d'éléments modulaires : les modules de simulation. Ces modules sont reliés entre eux par un bus qui permet à ces derniers d'échanger des messages et/ou des signaux. Un module OpenMASK a une architecture prédéfinie. Il s'agit d'une classe C++ qui peut contenir l'une des méthodes génériques suivantes : Init(), compute(), processEvent().

Notre approche pour intégrer les services de HLA dans OpenMASK est de créer un module global qu'on a appelé HLAC (HLA Communication), qui jouera le rôle de communicateur avec le bus HLA. En même temps, ce module échangera les données reçues avec les autres modules à travers le bus de OpenMASK.

Module HLAC

Le module HLAC est un module de simulation OpenMASK composé de deux parties. La première partie sert à détecter les évènements ou à capter les attributs des objets à partir du modèle de simulation et les traduire en des appels aux méthodes appropriés du RTIAmbassador. Ce code est appelé LRC (Local RTI Component). La deuxième partie capte les messages et les mises à jour reçues de la part du RTI en utilisant les fonctions de callback dans la classe FederateAmbassador. Cette partie est

appelée Fed-code (Federate Code). Dans ce qui suit nous allons présenter comment implémenter ces deux parties du fédérés.

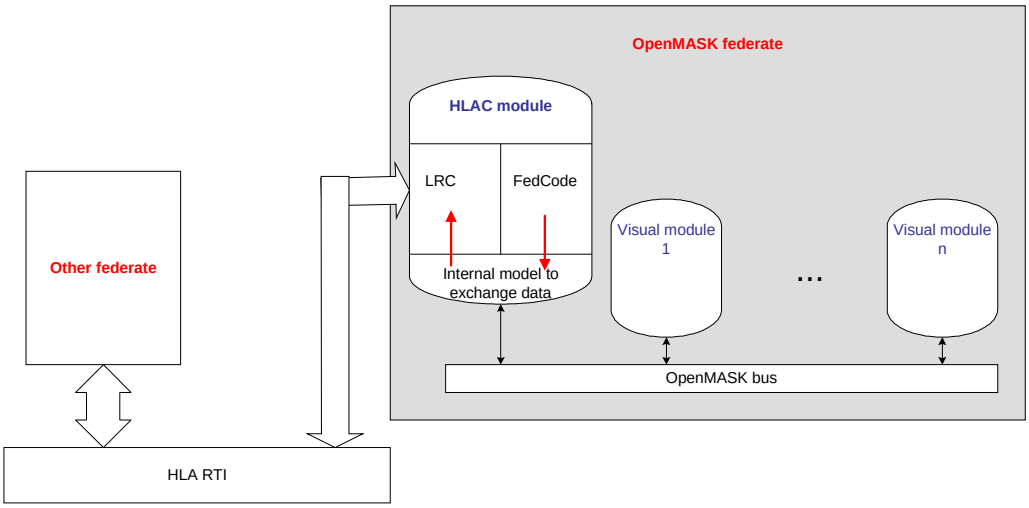


Figure 6.25 Module HLAC

LRC

Le module HLAC comporte les méthodes génériques suivantes : *init()*, *compute()*, *loadParameters()* ainsi que le constructeur et le destructeur de l'objet HLAC. La méthode *init()* est appelée une seule fois à l'initialisation de l'objet HLAC ainsi que le constructeur. Cette méthode sert à initialiser les attributs de l'objet de simulation. Nous utilisons cette méthode pour invoquer les services HLA pour rejoindre une fédération déjà créée par un autre fédéré, parcourir les objets et interactions HLA de la fédération et récupérer leurs identifiants et enfin, s'abonner aux objets et interactions qui intéressent le fédéré et publier les objets et les interactions propres au fédéré.

Pour rejoindre la fédération nous faisons appel au service permettant la participation d'un fédéré à une fédération joinFederationExecution(). Nous fournissons le nom de la fédération à rejoindre, le nom du fédéré et le pointeur vers la classe implémentant les callbacks du fédéré. Pour acquérir les identifiants attribués par le RTI aux objets, attributs, interactions et paramètres des interactions, nous faisons appel aux services : *getObjectClassHandle()*, *getInteractionClassHandle()*,

getAttributeHandle() et *getParameterHandle()*. Une fois les identifiants acquis, nous faisons appel aux services HLA permettant de s'abonner à un ou plusieurs objets ou interactions.

Les actions effectuées par un fédéré uniquement une seule fois sont intégrées dans la méthode *init()*. Pour les actions dynamiques, elles sont implémentées dans la méthode générale *compute()*. En effet, cette dernière est exécutée à une fréquence qu'on attribue à l'objet au début de la simulation. Au cours de la simulation, le fédéré envoie des informations vers le RTI concernant la mise à jour de ses attributs ou des interactions destinées à d'autres fédérés. Une détection d'une collision au cours de la simulation ou une interaction avec l'utilisateur déclenche un événement. Le fédéré peut ainsi, soit appeler les services d'envoi d'interactions à d'autres fédérés ou mettre à jour les attributs d'un ou plusieurs de ses objets HLA. Ainsi, nous faisons appel aux services *sendInteraction()* ou *updateAttributeValues()*.

Une fois arrivée à la fin de la simulation, on fait appel au service *resignFederationExecution()* pour quitter la fédération.

FedCode

La partie du FedCode consiste à récupérer les appels envoyés par le RTI et à extraire les données nécessaires pour le fédéré. Une classe qui hérite de l'interface FederateAmbassador doit être implémentée par le fédéré et doit définir les fonctions de callbacks. Les méthodes de la classe FederateAmbassador sont les services invoqués par la RTI pour diffuser des informations issues de la fédération vers le fédéré. Ces derniers services sont communément appelés « callbacks ». L'appel des callbacks par la RTI peut alors répercuter des mises à jour d'attributs souscrits, accorder une demande d'avance dans le temps, recevoir une interaction avec ses paramètres.
Ces fonctions de callback sont implémentées en dehors des méthodes générales du module HLAC. En effet, ces callbacks sont des méthodes de la classe HLAC puisque cette dernière hérite de la classe FederateAmbassador.

Dans la figure suivante nous présentons une fonction de callback permettant de recevoir une interaction. Elle est implémentée en dehors des méthodes générales du module HLAC (en dehors de *init()*, *compute()*...etc. Cette fonction est générale et peut être utilisée pour recevoir d'autres interactions. Il faut changer bien sûr le nom de l'interaction à recevoir dans le corps de cette fonction.

```
//////////////////////////////// HLA Callbacks//////////////////////////////////////////////
/** Callback : receive interaction
 */
void
ScooterOSO::receiveInteraction(RTI::InteractionClassHandle theInteraction,
                        const RTI::ParameterHandleValuePairSet& theParameters,
                        const RTI::FedTime& /*theTime*/,
                        const char */*theTag*/,
                        RTI::EventRetractionHandle /*theHandle*/)
    throw (RTI::InteractionClassNotKnown,
           RTI::InteractionParameterNotKnown,
           RTI::InvalidFederationTime,
           RTI::FederateInternalError)
{
    char *parmValue ;
    RTI::ULong valueLength ;
    char *parmValue1 ;
    RTI::ULong valueLength1 ;

    if (theInteraction == CommunicationClassID) {

                    for (int i = 0; i < theParameters.size(); ++i){
                        if (theParameters.getHandle(i) == ParamMESSAGEID)
                        {

                        valueLength = theParameters.getValueLength(i);
                        parmValue = new char[valueLength] ;
                        theParameters.getValue(i, parmValue, valueLength);

                        }
                        else

                        if (theParameters.getHandle(i) == ParamUSERID) {
                        valueLength1 = theParameters.getValueLength(i);
                        parmValue1 = new char[valueLength1] ;
                        theParameters.getValue(i, parmValue1, valueLength1);}

                       }
                                              }
    else cout << "Problème lors de la réception de l'intéraction!" << endl;

}
```

Figure 6.26 Fonction de callback générale *receiveInteraction()*

L'appel des fonctions de callbacks ne se fait que quand on rend la main au RTI. Les valeurs des attributs reçues à travers les callbacks sont utilisées pour animer l'objet 3D sous OpenMASK. En effet, l'objet visuel est représenté lui même par un module de simulation et reçoit par envoi de messages ou de signaux les nouvelles positions de l'objet pour l'animer.

Etapes de communication avec le RTI

- Participation à la fédération
- Initialisation
- Déclaration des intentions de publication et de souscription
- Synchronisation
- Boucle de simulation
- Fin de la simulation

Participation à la fédération

Il est nécessaire de créer des instances de classes RTIAmbassador et FederateAmbassador. Le fédéré peut créer et rejoindre la fédération. Ces fonctions de création et de participation sont implémentées dans la méthode générale *init()*.

Initialisation

Le fédéré demande au RTI la liste des *handles* de tous les « éléments HLA ». Cette action est implémentée aussi dans le bloc *init()*.

Déclaration des intentions de publication et de souscription

Le fédéré déclare ensuite ses intentions de publication et de souscription. Etant donné que cette action est exécutée une seule fois lors de la simulation on a implémenté ces déclarations dans le bloc *init()*.

Synchronisation

Nous présentons deux méthodes de synchronisation pour l'exécution du fédéré OpenMASK. Ces deux méthodes sont décrites dans le chapitre 7.

Boucle de simulation

La boucle de simulation se situe dans la méthode générale compute(). Elle a la structure suivante :

Tant que non fin-de-la-simulation Faire

Avancer dans le temps

Envoi des mises à jours ou interactions pour les entités simulées suivant le modèle de comportement.

Fin Tant que

Fin de la simulation

La fin de la simulation est déclenchée par l'interaction de l'utilisateur avec la simulation en appuyant sur une touche particulière du clavier. Une fois la boucle de simulation achevée, le fédéré supprime les objets enregistrés, désactive sa politique de gestion de temps et quitte la fédération.

6.6 Description de la dynamique globale des deux simulateurs

On commence la simulation par le lancement du fédéré graphique implémenté sous OpenMASK. Il crée la fédération et la rejoint. OpenMASK commence à afficher les objets graphiques dans leurs positions initiales. On lance ensuite le fédéré de contrôle. Les deux fédérés commencent par publier leurs attributs et souscrire aux attributs sollicités par chacun. Après la phase de publication et de souscription des attributs, le fédéré créateur demande au RTI de synchroniser l'ensemble des deux fédérés. Un point de synchronisation est annoncé aux deux fédérés. Au terme d'une initialisation, le RTI annonce, en même temps, aux deux fédérés la synchronisation de la fédération. La passerelle envoie une requête TCP socket demandant à OpenModelica de lancer la simulation. Ce dernier calcule les nouvelles valeurs des positions des différents objets et les envoie par des requêtes TCP socket à chaque pas de simulation. On a choisi un pas de simulation fixe pour OpenModelica pour permettre l'envoi et/ou la réception des requêtes TCP. La passerelle reçoit les messages TCP et affecte les nouvelles valeurs des positions à ses attributs. Le RTI détecte la mise à jour des attributs du fédéré contrôle ; il envoie ainsi un message au fédéré graphique contenant les nouvelles valeurs mises à jour. Le fédéré sous

OpenMASK affiche les nouvelles valeurs des positions des différents objets géométriques jusqu'à la réception de nouvelles valeurs de positions.

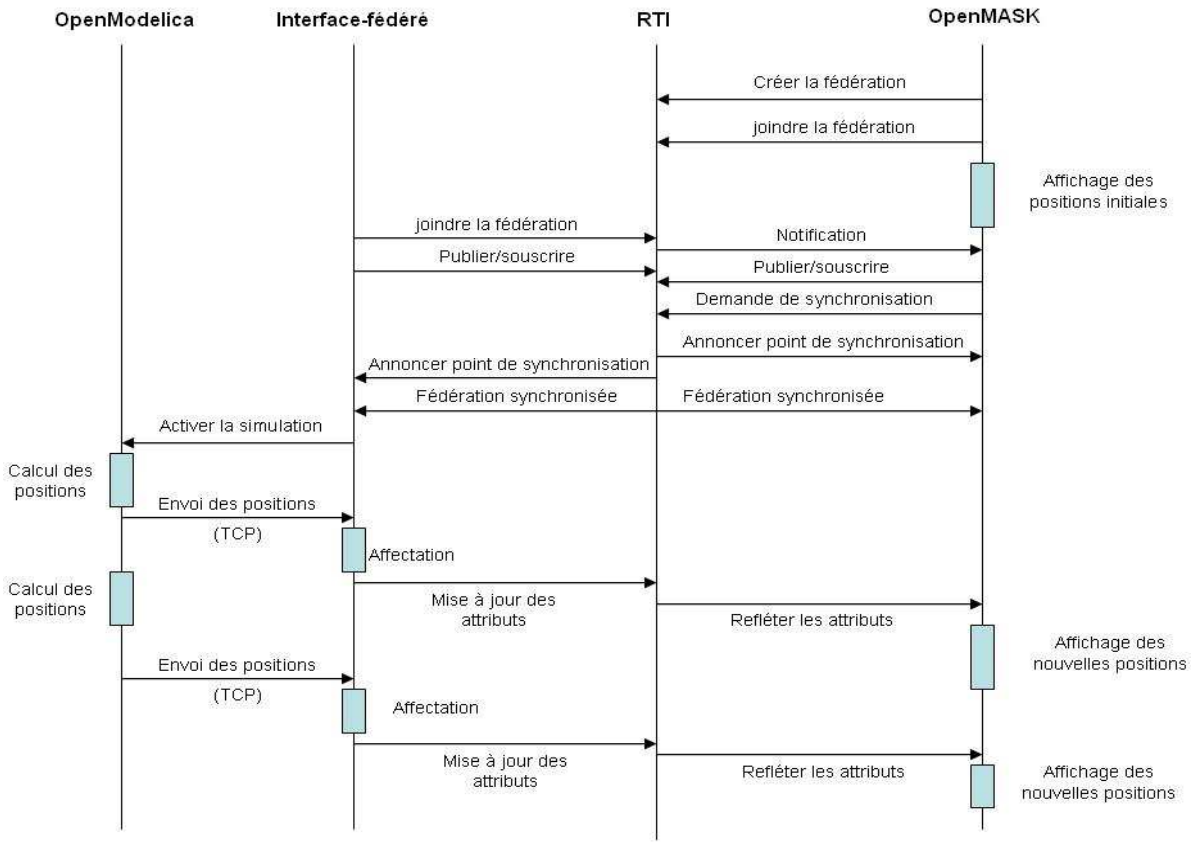


Figure 6.27 Diagramme de séquence des objets lors de la simulation

6.7 Conclusion

Nous avons présenté dans ce chapitre une méthode pour communiquer les deux simulateurs OpenMASK et OpenModelica. Pour chaque simulateur nous avons choisi l'une des méthodes générales d'intégration. Chaque méthode utilisée pour l'un des simulateurs est détaillée en précisant les services HLA utilisés et la manière de les appeler. Une description de la dynamique générale est présentée à la fin sans faire intervenir le temps. En effet, jusqu'à maintenant notre fédération n'a aucune stratégie de temps ainsi que les fédérés (les deux simulateurs). Cela sera le sujet du chapitre 7.

7. Chapitre 7: Gestion du temps et contraintes temporelles

7.1 Introduction

Nous proposons dans ce chapitre une contribution à la prise en compte et la gestion du temps dans les simulations basées sur HLA, plus précisément les simulations faisant intervenir OpenMASK et OpenModelica comme le cas dans notre approche. Notre objectif est d'aboutir à une simulation distribuée temps réel d'un système mécatronique **[Hadj-Amor (b), 2008]**.

7.2 Synchronisation des simulateurs au début de la simulation

Nous proposons deux méthodes possibles afin de synchroniser le début d'exécution des deux simulateurs.

Première méthode : cette méthode est la plus simple et consiste à lancer le fédéré OpenMASK en premier lieu. Ce dernier doit afficher les positions initiales des objets 3D et attendre que le fédéré OpenModelica rejoigne la fédération. Ainsi, le fédéré OpenMASK doit être le fédéré créateur de la fédération. Un mécanisme d'attente doit être alors implémenté. Le principe de base de cette méthode est le suivant :

- OpenMASK est le fédéré créateur de la fédération.
- Il est déclaré comme fédéré contraint par l'avancement du fédéré OpenModelica.
- Le fédéré OpenMASK effectue une demande au près du RTI pour avancer son temps logique en utilisant le service *timeAdvanceRequest()*.
- Le fédéré OpenMASK reste en attente du callback *timeAdvanceGrant()* pour qu'il puisse continuer son exécution. Entre temps, le fédéré OpenMASK envoie des requêtes au RTI en appelant la méthode *tick()* du RTIAmbassador. La fonction *tick()* fait du « polling réseau » ; elle interroge le RTI en vue d'un éventuel message, dans notre cas, le fédéré OpenMASK est en attente du message callback

timeAdvanceGrant(). La fonction *tick()* est une fonction bloquante, ainsi le fédéré OpenMASK est figé jusqu'à réception du callback *timeAdvanceGrant()*.

- Après que le fédéré OpenModelica rejoigne la fédération, il est déclaré fédéré régulateur et demande d'avancer son temps logique. Le RTI permet à OpenModelica d'avancer son temps et c'est seulement à ce moment que le fédéré OpenMASK reçoit le callback *timeAdvanceGrant()*.

Dans la figure suivante, nous montrons la boucle du *tick()* et elle est implémentée dans la méthode *compute()* du module HLAC.

```
void HLAC::compute()
{


if (step_valide)  {
                rtiamb.queryFederateTime(localTime);
                //je demande d'avancer le temps.
                RTIfedTime time_aux(localTime.getTime()+TIME_STEP.getTime());
                rtiamb.timeAdvanceRequest(time_aux);
                granted = false;
                }



   while (!granted) {
        try {
            rtiamb.tick();
            step_valide = false;
        }
        catch (RTI::Exception& e) {
            cout << "******** Exception ticking the RTI " << endl;
            throw ;
        }
    }

if (granted) {step_valide = true;}

//if granted on affiche les nouvelles positions sinon tick(à) est appelé!



if (granted) {


OMK::ExtensibleSimulatedObject::compute();
```

Figure 7.1 Synchronisation au niveau du module HLAC

La fonction *compute()* est exécutée par le contrôleur de OpenMASK à une fréquence qu'on définit dans le fichier de configuration. Ainsi, a chaque exécution de cette fonction, on vérifie si le fédéré a reçu le callback *timeAdvanceGrant()* ou pas. Si le

RTI rend la main à OpenMASK, il affiche les valeurs de simulations reçues par l'interaction et demande de nouveau à avancer son temps logique en appelant le service *timeAdvanceRequest()*.

Deuxième méthode : cette méthode consiste à lancer l'exécution des deux fédérés sans ordre défini. Elle est basée sur les services de gestion du temps HLA. Le fédéré créateur demande au RTI un point de synchronisation pour les deux fédérés en appelant la fonction *registerFederationSynchronizationPoint()*. Le RTI leur délivre alors un point de synchronisation en utilisant le callback *announceSynchronizationPoint()*. Les deux fédérés doivent utiliser la fonction *tick()* et de se mettre en mode d'attente jusqu'à réception du callback *federationSynchronized()* pour se lancer.

Cette méthode ne pose pas de problèmes d'implémentation au niveau de OpenMASK mais elle est plus difficile à utiliser avec OpenModelica. En effet, la passerelle de OpenModelica doit traduire tous les appels de callbacks du RTI à des appels vers la simulation sous OpenModelica. Pour cela, nous avons procédé de la manière suivante :

- Dans la passerelle, on déclare le fédéré OpenModelica créateur.
- On envoie un message à la simulation sous OpenModelica pour la figer. Pour cela, on envoie la chaîne de caractère « stop ». Nous rappelons que la fonction *receiveMsg()* décrite dans la figure 6.8 et 6.9, est toujours à l'écoute de messages provenant de la passerelle. A la réception de la chaîne de caractère « stop », un évènement est déclenché dans la simulation sous OpenModelica et la fonction *recMsgBlock()* est alors activé. (figure 6.10 et 6.11). Ainsi, la simulation sous OpenModelica est figée jusqu'à réception de la chaîne de caractère « wake ».
- Ensuite, la passerelle demande au RTI un point de synchronisation et se met en attente jusqu'à réception du callback *announceSynchronizationPoint()*. Une fois les deux fédérés synchronisés, la simulation de toute la fédération peut être lancée en appuyant sur une touche au clavier ou en mettant un *timer* au niveau du fédéré créateur.

- Une fois que l'utilisateur appuie sur une touche pour lancer la simulation des deux fédérés synchronisés, la passerelle doit en même temps donner l'ordre à la simulation sous OpenModelica de reprendre son exécution. Ceci s'effectue en envoyant un message contenant la chaîne de caractère « wake » qui sera capté par la fonction *recMsgBlock()* et la simulation ne sera plus figée.

La procédure d'activation et de désactivation de l'exécution de la simulation sous OpenModelica est présentée dans la figure commentée suivante :

```
// Envoyer une demande de blocage à OpenModelica
if (connectSocket > 0) {send(connectSocket,buf,strlen(buf)+1,0);
// Synchronization
cout << "Debut de la synchronization... " << endl;

chatting.pause(); //registerFederationSynchronizationPoint


// Déclarer le fédéré comme régulateur
chatting.setTimeRegulating();


// Synchronisation des fédérés jusqu'à réception du callback federationSynchronized
chatting.synchronize();
// A la fin de la synchronisation, on envoie une demande de déblocage à OpenModelica.
if (!chatting.paused) {send(connectSocket,buf1,strlen(buf1)+1,0);
```

Figure 7.2 Implémentation de la procédure d'activation/désactivation d'OpenModelica

Nous avons testé les deux méthodes. La première méthode est plus simple à implémenter et à mettre en œuvre. En effet, il n'est pas nécessaire d'implémenter les services de gestion du temps. De plus, il n'y a pas de messages à traduire de la passerelle vers le simulateur OpenModelica. Par contre, cette méthode n'est applicable que dans le cas ou le fédéré OpenMAsk est contraint par l'avancement du fédéré OpenModelica. Dans un cas plus général, il est préférable d'utiliser la deuxième méthode qui se base sur les services de gestion du temps.

7.3 Stratégie et degré d'implication des fédérés

Le temps dans un fédéré peut être représenté par des points sur l'axe du temps de la fédération. Ainsi, chaque fédéré peut avancer sur cet axe pendant l'exécution. Le service de gestion du temps fournit des mécanismes pour contrôler l'avancement des fédérés dans le but de garantir un ordre causal entre les différents évènements émis. La perception du temps courant peut être différente selon le fédéré mais l'avancement du temps est coordonné par la fédération.

Nous rappelons ici les différentes sortes de fédérés. Il existe quatre types de fédérés: fédéré régulateur, fédéré contraint, fédéré régulateur et contraint ainsi que fédéré ni contraint ni régulateur. Un fédéré qui déclenche un évènement ou envoie un message avec une estampille temporelle à d'autres fédérés est appelé fédéré régulateur. Par contre, un fédéré contraint est un fédéré qui reçoit des évènements ou des messages estampillés de la part d'autres fédérés. Ainsi, le temps d'avance d'un fédéré contraint dépend du celui du fédéré régulateur. En d'autres termes, l'avancement du fédéré contraint dans le temps est contraint par l'avancement dans le temps du fédéré régulateur.

L'approche la plus simple pour gérer le temps dans une simulation est d'avancer le temps par des pas fixes. Chacun des deux fédérés OpenMASK et OpenModelica peut être contraint, régulateur ou les deux. A chaque pas de temps fixe, chacun des deux fédérés peut envoyer des interactions ou recevoir des données. Chaque fédéré doit être impliqué dans la gestion du temps et chacun doit choisir son degré d'implication. Le fédéré OpenModelica est un fédéré de comportement. Les attributs des objets (object attributes) de ce dernier sont utilisés par le fédéré graphique OpenMASK pour animer le modèle 3D. En d'autres termes, le fédéré OpenModelica doit contrôler le fédéré OpenMASK pour fournir une simulation fluide. Pour cette raison, nous choisissons le fédéré OpenModelica comme fédéré régulateur et le fédéré OpenMASK comme fédéré contraint. L'avancement du temps logique de ce dernier est contraint par le fédéré régulateur.

Le fédéré OpenModelica, qui est régulateur et non contraint, synchronise le reste de la fédération mais il n'est pas contraint par les autres fédérés de la fédération. Le fédéré OpenMASK, quand à lui, est contraint mais non régulateur. Il permet au reste de la federation de réguler son temps logique mais il ne peut pas avoir une incidence sur les autres fédérés de la fédération. Ceci est recommandé surtout pour les fédérés passifs ou les fédérés graphiques. Ceci a été évoqué par la communauté CERTI ainsi que par Frederick Kuhl **[Kuhl, 1999]**.

Le service de gestion du temps HLA coordonne l'avancement du temps logique de tous les fédérés dans la fédération. Le RTI prévient les fédérés contraints de s'exécuter sans respecter les fédérés régulateurs. Puisque le fédéré OpenModelica est régulateur, nous avons choisi de le rendre temps réel. Dans ce cas, l'avancement du temps logique du fédéré OpenMASK sera contraint par celui du fédéré OpenModelica. Ainsi, la fédération va synchroniser son temps avec celui du temps d'horloge (*wallclock time*). Dans la partie qui suit, nous présentons un modèle Modelica pour synchroniser une simulation sous OpenModelica avec le temps réel **[Hadj-Amor (b), 2008]**.

7.4 Module temps réel pour OpenModelica

La progression du temps simulé durant l'exécution d'une simulation peut avoir ou pas une relation avec la progression du temps d'horloge. Dans les simulations analytiques où homme et matériel n'interagissent pas avec la simulation, la progression du temps simulé est souvent non synchronisée avec celui du temps d'horloge. Ces simulations sont mentionnées comme des simulations *as-fast-as-possible*. Ceci est le cas d'OpenModelica.

Nous rappelons que la fonction de transformation pour transformer le temps d'horloge (wallclock time) en temps simulé peut être utilisée pour les simulations temps réel ou temps réel échelonné.

$$T_s = f(T_w) = T_{start} + K * (T_w - T_{wStart})$$

Où :

T_w Une valeur du temps d'horloge.

T_{start} Temps simulé au début de la simulation.

T_{wStart} Temps d'horloge au début de la simulation.

K est un facteur.

Nous avons enrichi OpenModelica par un mécanisme pour synchroniser son exécution avec le temps d'horloge (wallclock time). Ce mécanisme de synchronisation doit introduire un mécanisme d'attente pour prévenir la simulation de s'exécuter plus rapidement que le temps d'horloge (temps réel). Nous utilisons ce mécanisme dans le module **RealTime** pour OpenModelica.

RealTime est un modèle Modelica que nous avons développé pour OpenModelica et Dymola en collaboration avec le chercheur Florian Wagner de l'université de Kaiserslautern en Allemagne. D'autres modèles Modelica peuvent instancier le modèle **RealTime**. Ces derniers seront synchronisés alors avec le temps réel. Le modèle **RealTime** est très utile dans les simulations HIL (Hardware-In-the-Loop) et pour l'utilisation des interfaces Homme/Machine avec le simulateur OpenModelica. L'idée de base est de tester à des pas de temps fixes et minimes si le temps réel est supérieur au temps simulé. Dans ce cas, le simulateur OpenModelica est figé jusqu'à ce que le temps réel devienne égal au temps simulé.

Parfois, la simulation se fige quelques instants à cause par exemple d'un grand nombre d'évènements. Le temps simulé "perdu" est alors reconquis le plus rapidement possible en simulant le plus rapidement possible. Ce phénomène peut induire à des incohérences d'exécution avec un programme externe ou un utilisateur en interaction

avec la simulation parce que les données vont être échangées le plus rapidement possible dans la phase de reconquête. En limitant la vitesse de simulation, ce comportement peut être évité. L'algorithme utilisé pour ce module est présenté dans la figure suivante:

While (simulation in progress)
    while simulation time > real time or simulationSpeed > maxSimulationSpeed
    wait a small time step
    end while
    compute state of the system at the end of this time step
end while

Figure 7.3 Algorithme du modèle *RealTime*

Afin d'implémenter le mécanisme d'attente, nous avons développé trois fonctions sous OpenModelica : une fonction *initialisationTime()*, une fonction *getTime()* et une fonction *sleepMicro()*. La première fonction sert à initialiser l'horloge du système. Elle est appelée au début de l'exécution de OpenModelica en utilisant la fonction *initial()* décrite dans le paragraphe 2.4.2.6. La deuxième fonction sert à lire le temps d'horloge et la troisième fonction sert à figer l'exécution de OpenModelica. Pour implémenter ces fonctions nous avons fait recours aux fonctions externes en C. Nous avons alors développé trois fonctions en C qui sont appelées au cours de la simulation par les fonctions enveloppantes sous OpenModelica. Nous avons choisi d'utiliser les mêmes noms pour les fonctions enveloppantes en Modelica et pour les fonctions externes qui leurs corresponde.

Pour lire le temps simulé au cours de l'exécution de la simulation nous avons utilisé directement la fonction *time()*de Modelica. Nous présentons les fonctions externes dans ce qui suit.

La fonction *initialisationTime()* :

Nous utilisons la fonction *gettimeofday()* pour initialiser l'horloge du système. Elle fournit une résolution temporelle très fine de l'ordre de la microseconde. La date de début de l'exécution est stockée dans la structure *timestart.*

```
static struct timeval timestart, timeend;

static struct timezone tz;


void initialisationTime()
{
gettimeofday(&timestart, &tz);
}
```

Figure 7.4 La fonction externe *initialisationTime()*

La fonction *getTime()* :

Cette fonction renvoie la date de l'horloge du système. Elle est présentée dans la figure suivante :

```
double getTime()
{
gettimeofday(&timeend,&tz);

double t1, t2;


        t1 =  (double)timestart.tv_sec + (double)timestart.tv_usec/(1000*1000);

        t2 =  (double)timeend.tv_sec + (double)timeend.tv_usec/(1000*1000);

        return t2-t1;
}
```

Figure 7.5 La fonction externe *getTime()*

La fonction *sleepMicro()* :

Cette fonction permet de figer l'exécution de OpenModelica pendant 0.001 secondes. Pour cela, nous avons utilisé la fonction système *usleep()* de Linux. Elle suspend l'exécution du programme appelant durant *x* microsecondes.

```
double sleepMicro(double x)
{

        usleep (x*1000000);
        return 1;
}
```

Figure 7.6 La fonction externe *sleepMicro()*

Sous OpenModelica, nous effectuons cet appel pour suspendre son exécution pendant 0.001 secondes dans la boucle de synchronisation présentée dans la figure 7.3.

Nous avons implémenté ce modèle sous OpenModelica. Il suffit de créer une instance de ce modèle dans un autre modèle Modelica pour que ce dernier soit synchronisé avec le temps réel. Ce modèle peut être utilisé sous Dymola en modifiant les fonctions externes pour leur rendre compatibles à la plateforme windows. Pour une modélisation graphique, il suffit de glisser ce dernier dans un autre modèle pour qu'il s'exécute en synchronisation avec le temps réel.

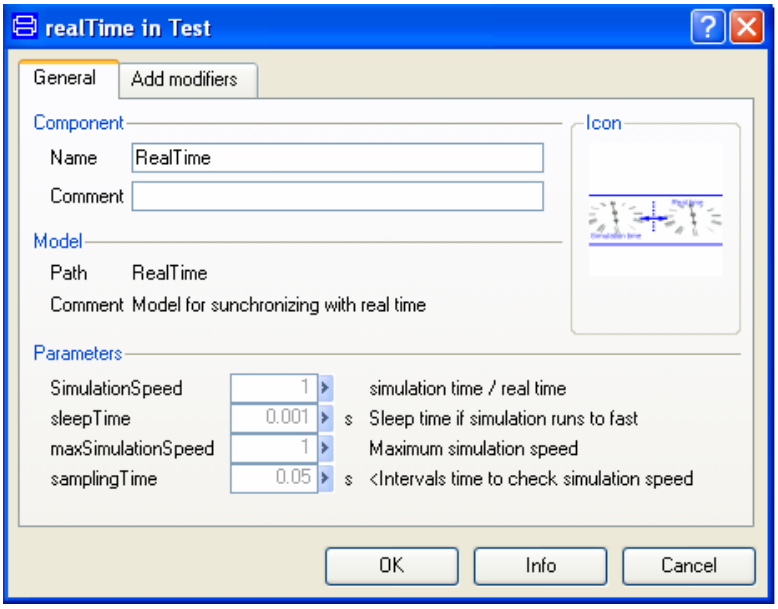


Figure 7.7 Modèle RealTime sous Dymola

Le module *RealTime* a été testé. Nous présentons les résultats dans le chapitre 8.

7.5 Évolution du temps durant la simulation

Le module **RemoteController**, décrit précédemment dans le chapitre 6, envoie les valeurs de la simulation à chaque pas de temps fixe à la passerelle. Le pas de temps (*time step*) est un paramètre dans le module RemoteController qui peut être modifié.

La passerelle joue le rôle du fédéré OpenModelica. Il effectue le cycle habituel **TIME ADVANCE REQUEST** nommé *TAR* et **TIME ADVANCE GRANT**, nommé *Grant*. De l'autre coté, le fédéré OpenMASK effectue le même cycle pour avancer son temps logique. Le fédéré OpenMASK est contraint par l'avancement du temps du fédéré OpenModelica. Ceci veut dire que le fédéré OpenMASK est figé quand le fédéré OpenModelica est en train de mettre à jour ses attributs ou en attente pour synchroniser avec le temps réel. Plus explicitement, quand le fédéré OpenModelica est en attente pour se synchroniser avec le temps réel, il n'envoie pas des valeurs de simulation à la passerelle. Ceci est logique vu que OpenModelica est figé à cet instant

là. La passerelle, qui joue le rôle du fédéré OpenModelica, effectue une demande pour avancer son temps logique seulement si elle reçoit de nouvelles valeurs de la simulation OpenModelica à travers les sockets. Comme la passerelle est régulateur et le fédéré OpenMASK est contraint, alors ce dernier ne peut pas avancer son temps logique et il est figé quand OpenModelica est en mode d'attente pour se synchroniser avec le temps réel.

7.6 Conclusion

Nous avons présenté dans cette partie une stratégie de gestion du temps pour notre application. Cette stratégie se base sur le service HLA de gestion du temps en définissant pour chaque fédérateur son rôle et son degré d'implication dans la gestion du temps.

Nous avons ensuite proposé une approche pour exécuter la fédération en temps réel. Pour cela, un modèle Modelica a été développé. Ce dernier permet à une simulation sous OpenModelica à s'exécuter en temps réel. Notre approche a été testée. Les résultats sont présentés dans le chapitre Application.

8. Chapitre 8 : Application

Nous appliquons dans ce chapitre le processus complet de notre méthode de simulation distribuée temps réel d'un système mécatronique pour le cas d'un système de guidage à vis à billes. Un prototype virtuel de ce système est obtenu à la fin. On met l'accent dans notre exemple surtout sur trois points :

- Modélisation de la partie contrôle et la partie opérative et leur couplage.
- Conception de la fédération.
- Implémentation de la stratégie de gestion du temps.

Le système de guidage linéaire est contrôlé par une commande. Ce dernier alimente un moteur DC par une tension selon la position *x* de la masse en déplacement. La spécification de la commande est comme suit :

- Faire 10 fois :
  - Démarrer et déplacer la masse à partir de la position gauche jusqu'à la droite.
  - Démarrer et déplacer la masse à partir de la position droite jusqu'à la gauche.
- Attendre 3 secondes.

Le modèle physique du système de guidage linéaire, représenté dans la figure suivante, est composé d'un moteur DC (Direct Current) à courant continu, une tige filetée et une masse guidée en translation.

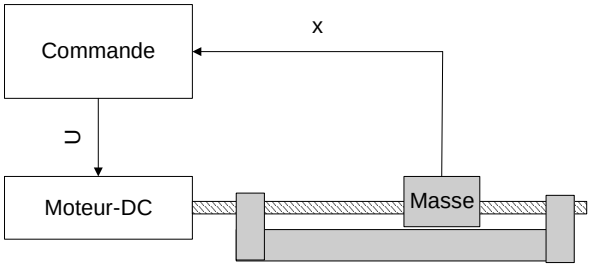


Figure 8.1 Système de guidage

La modélisation de ce système mécatronique nous permet de distinguer 3 parties à modéliser : la partie commande, la partie opérative qui représente le modèle de comportement, et les modèles graphiques (fichiers 3D).

Comme nous l'avons présenté précédemment, la nouvelle version de OpenMASK se base sur le nouveau moteur de rendu Ogre (paragraphe 2.5.1). Ainsi pour animer un modèle 3D, nous devons utiliser le format hiérarchique d'Ogre (*.scene*). Les modèles graphiques de notre exemple ont été réalisés grâce à un outil de CAO (3ds MAX). Une fois les fichiers 3D créés, nous utilisons le plugin OgreMax **[OgreMax]** pour les convertir en format *.scene* qui est supporté par OpenMASK V4.

Les équations dynamiques du système sont : **[Aublin, 1992]**

$(J + mp^2)\ddot{\theta} + b\dot{\theta} = ki$ et $x = p\theta$ **(1)**

$L\frac{di}{dt} + Ri = V - k\dot{\theta}$ **(2)**

Où :

J : moment d'inertie
θ : angle de rotation
m : masse
p : pas de vis
b : coefficient de frottement
x : déplacement
L : inductance
R : résistance
V : tension
i : intensité
k : force éléctromotrice constante

Le comportement global est représenté par les deux automates hybrides suivants : Un automate hybride pour la partie commande et un autre pour la partie opérative.

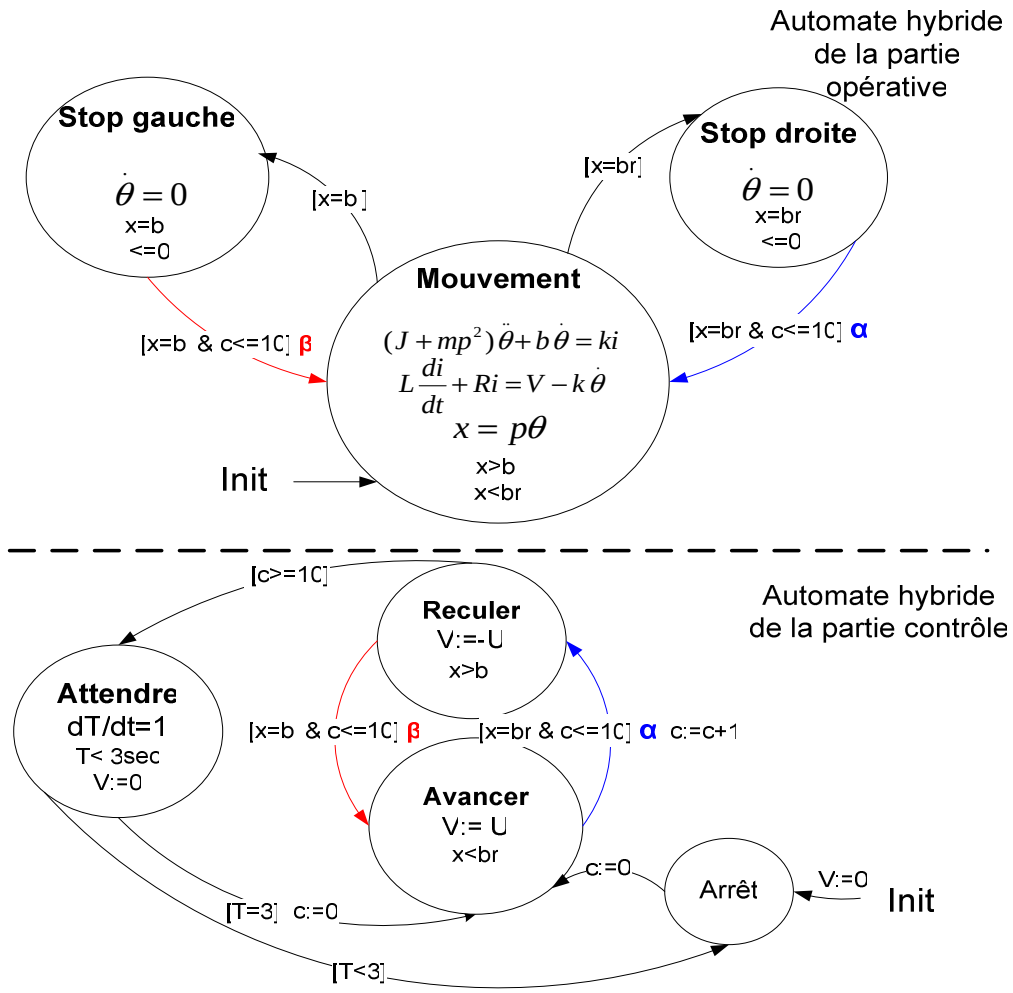


Figure 8.2 Automates hybrides de la partie commande et de la partie opérative

On peut distinguer 3 situations pour l'automate de la partie opérative (comportement) : Une situation de mouvement, une situation d'arrêt à droite et une situation d'arrêt à gauche. Chaque situation est composée d'une équation différentielle et un invariant. Les variables d'état sont la variable x de déplacement et la variable **θ** qui détermine l'angle de rotation de la tige. Les évènements étiquetés a et b permettent le couplage de la partie commande à la partie opérative.

Pour l'automate de commande, on peut distinguer 4 états : état d'arrêt, état d'attente, état d'avancement et état de recul. Dans les deux états *Avancer* et *Reculer* les équations consistent à inverser le voltage pour passer d'un état à un autre. Après 10 activations des états *Avancer*/*Reculer* l'état *attendre* est activé. L'équation de l'état *Attendret* est : V :=0. Cet état est alors actif pendant 3 secondes.

Nous avons implémenté sous Modelica les automates hybrides des deux parties contrôle et opérative en utilisant notre approche. Les objets mobiles dans notre système mécatronique sont la tige filetée (rotation) et la masse (déplacement). Chacun des deux objets a une variable d'état propre à lui. La tige a comme variable $\boldsymbol{\theta}$, quand à la masse, elle a comme variable x. De l'autre coté, dans les automates hybrides couplés, les variables x et $\boldsymbol{\theta}$ représentent les variables d'état de l'automate hybride de la partie opérative. Les variables $\boldsymbol{x}$ et $\boldsymbol{\theta}$ sont communiquées à la passerelle pour pouvoir les communiquer à OpenMASK. Ceci est le rôle du modèle *RemoteController* de passer ces variables à la passerelle. Nous devons déclarer une interaction HLA dans le fédéré représentant OpenModelica (la passerelle). Cette interaction aura comme paramètres deux variables $\boldsymbol{x}$ et $\boldsymbol{\theta}$. La variable réelle, i, n'est pas communiquée à OpenMASK car elle n'est pas nécessaire à l'animation graphique 3D, bien qu'elle soit nécessaire dans l'automate hybride.

Dans notre modèle général qui contient toutes les instances connectées, nous créons une instance de la classe *RealTime* pour synchroniser la simulation sous OpenModelica avec le temps réel. En effectuant une première expérimentation en utilisant seulement le simulateur OpenModelica nous obtenons les résultats suivants :

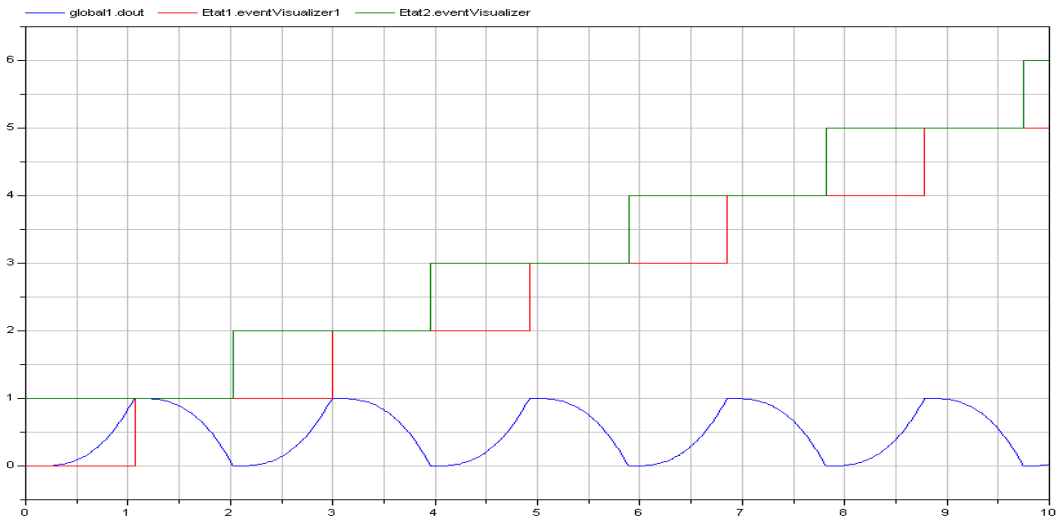


Figure 8.3 Résultats de la simulation sous OpenModelica

La variable de déplacement x est représentée par la courbe en bleu. Les évènements d'activation de l'état *Avancer* sont représentés en vert et les évènements d'activation des de l'état *Reculer* sont en rouge.

Nous remarquons aussi que la différence entre le temps simulé sous OpenModelica et le temps d'horloge est de l'ordre de 10^{-2} secondes. Ce qui montre le bon fonctionnement du module *RealTime* sous OpenModelica.

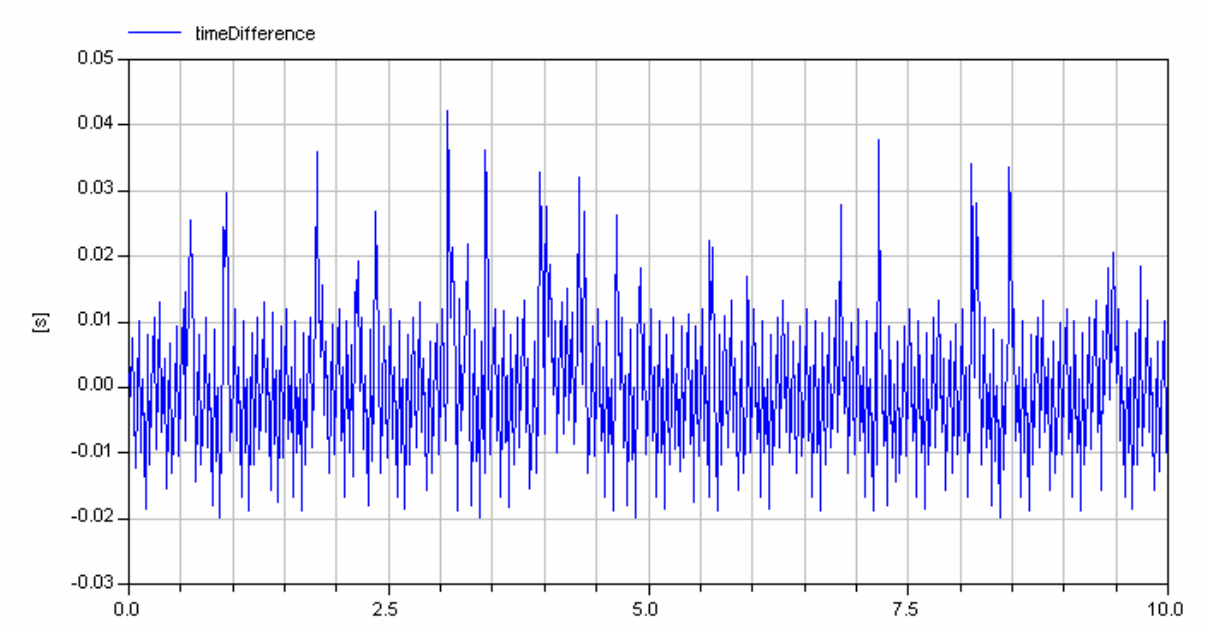


Figure 8.4 Différence entre le temps simulé et le temps d'horloge

Dans un premier terminal Linux, nous exécutons le RTI. Dans un deuxième terminal Linux, nous exécutons la passerelle. Dans un troisième terminal Linux, nous exécutons OpenMASK. Une fois le fédéré OpenMASK lancé, il affiche les objets 3D avec leurs positions initiales. Finalement, nous exécutons dans un autre terminal Linux OpenModelica pour simuler les automates hybrides. Le code des modèles de OpenModelica et OpenMASK sont disponibles au laboratoire LISMMA et occupent environt 20 pages.

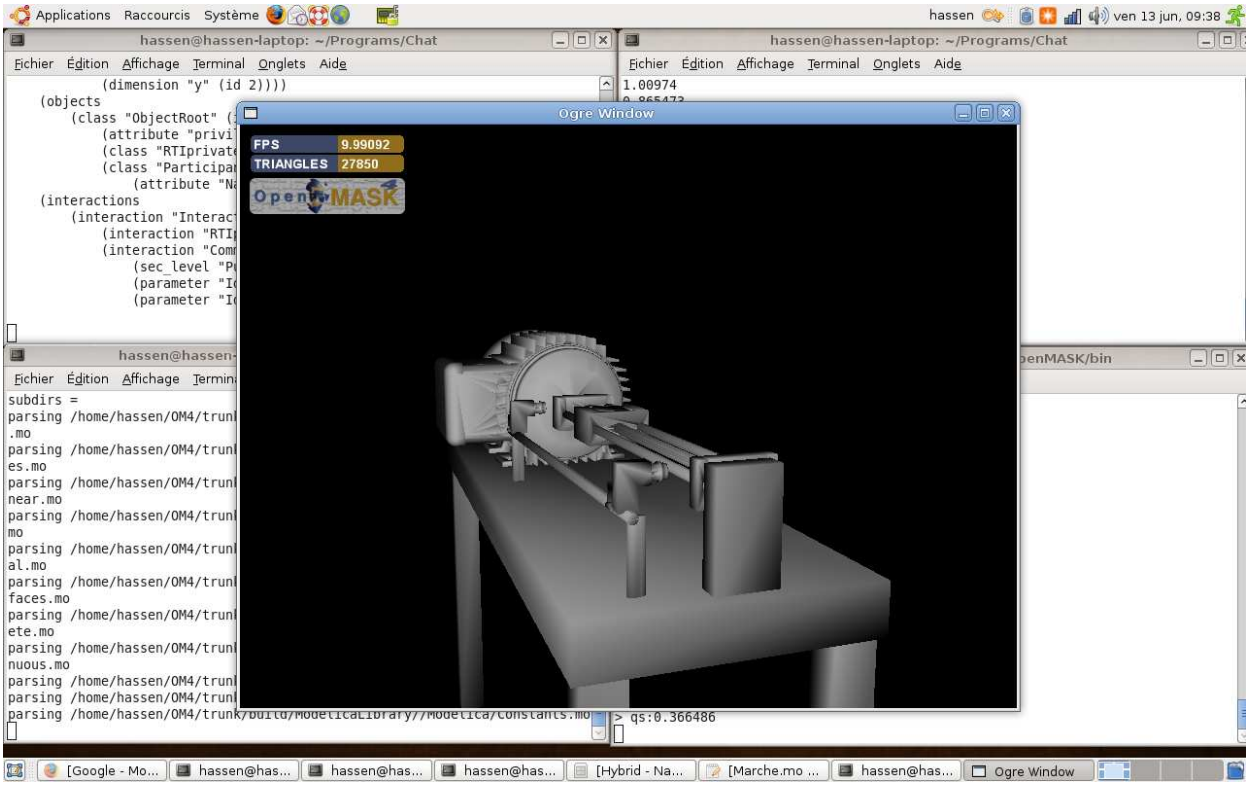


Figure 8.5 Capture d'écran de l'animation du système

Conclusion générale et perspectives

Dans cette thèse nous avons proposé une méthodologie pour la modélisation et la simulation en temps réel des systèmes mécatroniques. L'évolution rapide des marchés concurrents exige la diminution du temps de développement d'un produit en conservant la qualité et la performance du système. Il est donc nécessaire d'augmenter l'efficacité du processus de conception. Pour répondre à cette nécessité, en complément des outils d'analyse, la simulation, et spécialement le prototypage virtuel, est devenu l'une des clés technologiques. Nous nous sommes basés sur la norme HLA IEEE1516 afin d'établir une communication entre différents simulateurs de différents domaines technologiques, plus spécifiquement, entre les simulateurs OpenMASK et OpenModelica. Utiliser un standard de communication était pour nous une priorité car les utilisateurs ont besoin d'outils fiables et unifiés. La coopération entre plusieurs simulateurs de différents domaines pour aboutir à un prototype virtuel performant d'un système mécatronique était également nécessaire. Pour répondre à ce besoin, nous avons montré comment utiliser le standard HLA afin d'établir une communication entre deux simulateurs à code ouvert. Nous avons modélisé le comportement du système en se basant sur le langage Modelica que nous l'avons complété par un formalisme qui nous semblait indispensable.

Pour la modélisation du système mécatronique, nous nous sommes basés sur notre approche qui permet de distinguer la partie commande de la partie opérative tout en étudiant les liens d'échange entre ces deux parties. Nous avons utilisé le formalisme des automates hybride afin de modéliser le comportement de ce système dynamique hybride. Nous avons alors présenté une méthode générale pour implémenter les automates hybrides avec le langage Modelica. Par la suite, nous avons proposé une méthode afin d'implémenter le couplage entre les automates hybrides de la partie commande et la partie opérative. Afin de faciliter l'utilisation de cette méthode, nous avons proposé une librairie Modelica qu'on a nommé HybridAutomataLib afin de

modéliser graphiquement le comportement du système sans avoir recours à l'écriture de code.

Du point de vue du standard HLA, nous avons présenté une approche pour rendre les deux simulateurs compatibles à HLA. Pour chaque simulateur nous avons choisi une méthode d'intégration. Pour OpenModelica, nous avons créé une passerelle, basée sur les sockets, qui lui permet de communiquer avec le bus HLA. Cette dernière permet à OpenModelica de communiquer avec n'importe quel autre simulateur compatible à HLA. De même pour OpenMASK, nous avons implémenté une méthode qui permet à une simulation 3D de communiquer avec des autres simulateurs HLA et plus précisément avec le simulateur OpenModelica.

Afin de synchroniser les deux simulateurs, nous avons établi une stratégie d'avancement du temps de la simulation globale. Dans cette stratégie, l'avancement dans le temps du simulateur graphique OpenMASK est régulé par l'avancement du simulateur OpenModelica. Rendre l'exécution de la simulation globale une exécution temps réel était l'un de nos défis. Nous nous sommes basés sur notre stratégie de synchronisation qui fait appel aux services de gestion du temps de la norme HLA pour proposer une méthode qui permet à la simulation de s'exécuter en temps réel. Un module de synchronisation avec le temps d'horloge pour OpenModelica a été développé. On l'a nommé *RealTime*. Nous avons regroupé conjointement la stratégie d'avancement du temps et le module *RealTime* pour aboutir à une exécution de la simulation globale synchronisée avec le temps réel.

Enfin, nous avons choisi un système d'une masse pilotée en position pour l'application. Nous avons utilisé les différentes approches et méthodes pour la simulation de ce système. Cela nous a permit de démontrer les possibilités de notre méthodologie globale en termes de faisabilité.

Perspectives

Pour rendre cet environnement de simulation multidisciplinaire interactif, nous proposons de gérer les interactions au niveau du simulateur OpenMASK tels que la gestion des collisions, les interactions avec des interfaces haptiques…etc. Dans ce travail de recherche, nous n'avons pas abordé cet aspect. Une restructuration de la stratégie d'avancement du temps de la simulation globale est obligatoire. Nous proposons de définir chaque simulateur comme fédéré contraint et régulateur en même temps et ainsi OpenMASK ne sera plus un fédéré passif car il doit interagir avec l'utilisateur. Nous n'avons pas testé cette proposition qui sera l'un de nos futurs axes de recherche.

Nous n'avons pas distribué la simulation géographiquement vu que les simulateurs s'exécutaient en local. La norme HLA, plus précisément l'implémentation CERTI, nous permet d'effectuer cette distribution.

L'environnement proposé est basé sur des outils à code ouvert. Ces outils sont parfois difficiles à utiliser tel que OpenMASK. Il serait utile de réduire cette difficulté en proposant un outil pour la génération automatique du module HLAC et de la passerelle de OpenModelica en parcourant le fichier XML ou FED de la fédération. Ceci augmentera les capacités graphiques de l'environnement proposé.

Un autre aspect important pouvant améliorer cette solution sera de créer une base de données de modèles réutilisables de composants mécatroniques basée sur le formalisme des automates hybrides.

Une connexion avec un environnement de conception d'architecture de systèmes basé sur UML/SysML et la norme IEEE15288 est également envisagée **[Turki, 2008]**.

Bibliographie

[Adelantado, 2001(2)] M. Adelantado, M. Guez, B. Lesot, *"A scalable Modeling and Simulation Infrastructure for Fast Time Airport Management"* AIAA Modeling and Simulation Technologies Conference – 6-9 August 2001, Montreal (Quebec).

[Adelantado, 2001] M. Adelantado , P. Siron *"Multiresolution Modeling and Simulation of an Air-Ground Combat Application"*, 2001 Spring Simulation Interoperability Workshop, Orlando, March 25-30, 2001.

[Alla, 2001] R. David and H. Alla, *"On Hybrid Petri Nets"*, J. of Discrete Event Dynamic Systems: Theory and Applications, vol.11, **pp**. 9-40, 2001.

[Alur, 2000] R. Alur, R. Grosu, Y. Hur, V. Kumar, I. Lee, "*Modular Specification of Hybrid Systems in CHARON*." Proceedings of the 3rd International Workshop on Hybrid Systems: Computation and Control, Pittsburgh, PA, March 23-25, 2000.

[Andersson, 1994] M. Andersson. *"Object-Oriented Modeling and Simulation of Hybrid Systems"*. Thèse, Department of Automatic Control, Lund Inst. of Technology, Sweden, Décembre, 1994.

[Andreu, 1996] D. Andreu , *« Commande et supervision des procédés discontinus : une approche hybride »*. Thèse de doctorat, Université Paul Sabatier, Toulouse (France) : 173 **pp**., 1996.

[Aublin, 1992] M. Aublin et al "*Systèmes mécaniques, théorie et dimensionnement"*. Paris: Dunod, 1992.

[Ayani, 1992] R. Ayani, H. Rajaei, *"Parallel simulation using conservative time windows"***,** Proceedings of the 24th conference on Winter simulation, Virginia, USA, 1992, **pp** 709-717.

[Bieber, 1998] P. Bieber, J. Cazin, P. Siron, G. Zanon *"Security Extensions to ONERA HLA RTI Prototype"* Fall

Simulation Interoperability Workshop, Orlando, September 13-18, 1998.

[Bierbaum (a), 2001] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker and C. Cruz-Neira, "VR Juggler: A Platform for Virtual Reality Application Development", IEEE VR 2001, Yokohama, Japan, March 2001.

[Bierbaum (b), 2001] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker and C. Cruz-Neira, "VR Juggler: An Open Source Platform for Virtual Reality Applications", IEEE VR 2001, Yokohama, Japan, March 2001.

[Bierbaum, 1998] A. Bierbaum and C. Just, "*Software Tools for Virtual Reality Application Development*", SIGGRAPH' 98 Course 14, USA, 1998, **pp** 32-45.

[Bitam, 2003] M. Bitam, H. Alla,"*Modelling a Communication Network Under TCP/IP Protocol Using Hybrid Petri Nets*", On Analysis and Design of Hybrid Systems,**pp**.105,2003.

[Borchshev, 2002] A. Borchshev , Y. Karpov , V. Kharitonov , "*Distributed Simulation of Hybrid Systems With AnyLogic and HLA*". In Future Generation Computer System, 2002,Volume 18, Issue 6. **pp**.829-839.

[Borshchev,2000] A.V. Borshchev, Y.B. Kolesov, and Y.B. Senichenkov. "*Java engine for UML based hybrid state machines.*" In 2000 Winter Simulation Conference (WSC'00), Orlando, Florida, USA, Décembre 2000.

[Brams, 1983] G. W. Brams, *« Réseaux de Petri : théorie et pratique »*, Editions Masson, Paris, 1983.

[Bréholée, 2002] B. Bréholée, P. Siron "*CERTI: Evolutions of the ONERA RTI Prototype* " In Proceedings of the Fall 2002 Simulation Interoperability Workshop 8-13 September 2002, Orlando, FL, USA.

[Bréholée, 2005] B. Bréholée, "*Interconnexion de simulations distribuées HLA*" , Thèse soutenue le 8 mars 2005, Ecole Nationale Supérieure de l'Aéronautique et de l'Espace.

[Broenink, 1996] J.F. Broenink and P.B.T. Weustink. *"A combined system simulator for mechatronic systems."* In Proc. of Modeling and Simulation (ESM'96), pages 225–229, Budapest, Hungary, 1996. Publishing SCS Europe, Ghent.

[Broman, 2007] David B.,Thèse numéro 1337: "*Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments*", ISBN 978-91-85895-24-3, ISSN 0280-7971, Linköping Studies in Science and Technology, December 7, 2007.

[Bryant, 1977] Bryant R. E., *« Simulation of Packet Communication Architecture Computer Systems »*, Computer Science Laboratory. Massachusetts Institute of Technology, Cambridge, 1977.

[Calvin, 1995] Calvin J.O., Weatherly R., *"An introduction to the High Level Architecture (HLA) Runtime Infrastructure"*, 14. th. Workshop on the Standards for the Interoperability of Distributed Interactive Simulation, Orlando, Florida, 1995.

[Carloni, 2004] L. Carloni, M.D. Di Benedetto, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli. *"Modeling techniques, programming languages and design toolsets for hybrid systems."* Technical Report Deliverable DHS4-5-6, Project IST-2001-38314 COLUMBUS, 2004. http://www.columbus.gr/documents/public/WPHS/Columbus_DHS3_0.2_Cover.pdf

[Cébron, 2000] B. Cébron, "*Commande de systems dynamiques hybrids*", Thèse soutenue le 8 décembre 2000, Université d'Angers.

[Chandy, 1981] Chandy, K. M., J. Misra, *"Asynchronous distributed simulation via a sequence of parallel computations"*, Communications of the ACM, 1981.

[Champagnat, 1998] R. Champagnat, « *Supervision des systèmes discontinus : définitiond'un modèle hybride et pilotage en temps-réel* », Thèse de Doctorat de

l'Université Paul Sabatier de Toulouse, soutenue le 1er octobre 1998.

[Chauffaut, 2003] A. Chauffaut, T. Duval, C. Le tenier and M. Rouille, *"Tutorial for VR-OpenMASK, A Software Development Platform for Virtual Reality "*, Virtual Concept 2003, Biarritz, France, Novembre 2003, Tutorials **pp** 44-78.

[Communiqué Presse, 2005] PRESS RELEASE, Dynasim, *« DaimlerChrysler, BMW, Audi and Volkswagen choose Dymola for modeling and simulation of air conditioning systems »*, 31 Mars 2005.

[Cours Dymola] **«** *Cours « what is Dymola ? »*, http://www.ece.arizona.edu/~cellier/GettingStarted.pdf

[Miller, 1995] D. C. Miller & Jack A. Thorpe *"Simnet: The Advent Of Simulator Networking"* Proceedings of the IEEE 83(1995) Aug., No. 8, New York, U.S.

[Dymola User Manual] **«** *Dymola : User Manual »*, Dynasim.

[Febbraro, 2003] A. Di Febbraro and N.Sacco,*"Experimental Validation Of A Hybrid Petri-Net Based Model Of Urban Transportation Networks"*,C.On Analysis and Design of Hybrid Systems,**pp**.111,2003.

[Finkenzeller, 2003] D. Finkenzeller, M. Baas, S. Thuring and A. Schmidt, *"VISUM: A VR System for The Interactive And Dynamics Simulation Of Mechatronic Systems"*, Virtual Concept 2003, Biarritz, France, November 2003, **pp**155-162.

[Fritzon, 2002] P. Fritzson, P. Aronsson, P. Bunus, V. Engelson, L. Saldamli, H. Johanson, and A. Karström. *"The open source Modelica project."* In Proc. of 2nd International Modelica Conference, pages 297–306, 2002.

[Fritzon, 2003] P. Fritzon., *"Principles of Object-Oriented Modeling and Simulation with Modelica 2.1"*, IEEE press,Wiley-interscience, 2003.

[Fritzon, 2007] P. Fritzon, al, "*OpenModelica System Documentation, Preliminary Draft, 2007-06-20 for OpenModelica 1.4.3*", june 2007.

[Fritzson, 2006] Fritzson P. , Aronsson P., Pop A. *" OpenModelica Users Guide », PELAB"*, Department of Computer and Information Science, Linköping University, Suède, Septembre 2006.

[Fujimoto, 1998] Fujimoto, R.M.. *"Time Management in the High Level Architecture."* SIMULATION Special Issue on High Level Architecture, 1998, vol. 71, no. 6, 388-400.

[Fujimoto, 2000] R. M. Fujimoto, *« Parallel and Distributed Simulation Systems »*, Wiley Series on Parallel and Distributed Computing, Albert Y.Zomaya, Series Editor, 2000.

[Gueguen, 2000] H. Guéguen, M-A.Lefebvre,*"A Comparison of Mixed Specification Formalisms"*,Conference Proceedings,ADPM,18-19 September 2000,pages **pp**133-138.

[Gueguen, 2001] H. Gueguen, N. Bouteille: *"Extensions of Grafcet to structure behavioural specifications"*, Control Engineering Practice 9 /7 **pp** 743-756,2001.

[Hadj-Amor (a), 2008] H. Hadj-Amor, Soriano T., "*Integrating OpenModelica simulator with HLA"*, 7th France-Japan (5th Europe-Asia) congress on Mechatronics, Mecatronics 2008, Annecy, France, 21-23 May, 2008.

[Hadj-Amor (b), 2008] H. Hadj-Amor, T. Soriano, Under review, "*An approach for virtual prototyping of mechatronic systems based on HLA real time distributed architecture"*, Simulation Journal, soumise 19 août 2008.

[Hadj-Amor, 2006] H. Hadj-Amor, T. Soriano, "*Integrating a virtual prototyping simulator with HLA"*, 4th IEEE International Conference on Information and Communication Technology, ICICT 2006, Cairo, Egypt, 10-12 December, 2006.

[Hadj-Amor, 2007] H**.** Hadj-Amor, T. Soriano, *"Simulation distribuée Openmask et Openmodelica pour la mécatronique* ", Actes des journées nationales GDR MACS JNA-JDA, 2007.

[Harashima, 1996] F. Harashima, M. Tomizuka, T. Fukuda, "*Mechatronics – What is it, why and how? An editorial*", IEEE/ASME Transactions on Mechatronics 1, 1-4, 1996.

[Harrouet, 2000] F. Harrouet, « *oRis : s'immerger par le langage pour le prototypage d'univers virtuels à base d'entités autonomes*. », Thèse de doctorat, Université de Brest. Décembre 2000.

[Henzinger, 1995] T.A. Henzinger, P.W. Kopke, A. Puri, P. Varaiya, "*What's decidable about hybrid automata ?*",27th Annual ACM Symp. On Theory and Computing (STOCS), 1995.

[Henzinger, 1997] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. *"HyTech: A Model Checker for Hybrid Systems."*, Lecture Notes In Computer Science; Vol. 1254, Proceedings of the 9th International Conference on Computer Aided Verification, **pp** 460 – 463, 1997, ISBN:3-540-63166-6.

[Hibino, 2002] H. Hibino, Y. Fuduka, Y. Yura, K. Mitsuyuki, K. Kaneda,*"Manufacturing Adapter Of Distributed simulation Systems Using HLA"*. In Proceedings of the 2002 Winter Simulation Conference,2002, **pp**. 1099-1107.

[Hien, 2001] N. V. Hien,*" Une Méthode Industrielle de Conception de Commande par Automate Hybride Développée en Objets"*, Thèse, AIX-MARSEILLE III, 18 DEC 2001.

[Hoferet, 1995] R.C. Hofer, M.L Loper, *"DIS today [Distributed interactive simulation]"*, Proceedings of the IEEE Volume 83, Issue 8, Aug 1995 **pp** 1124 – 1137.

[IEEE 1278.1] *IEEE 1278.1 - IEEE Standard for Distributed Interactive Simulation*--Application Protocols was approved in 1995.

[IEEE 1516.1] The Institute of Electrical and Electronics Engineers, Inc., IEEE 1516.1-2000. *« IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) –Federate Interface Specification »*, 9 March 2001.

[IEEE 1516.2] The Institute of Electrical and Electronics Engineers, Inc., IEEE 1516.2-2000. *« IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) –Object Model Template (OMT) Specification »*, 9 March 2001.

[IEEE1516.3, 2003] The Institute of Electrical and Electronics Engineers, Inc., IEEE 1516.3-2003, *"Recommended Practice for High Level Architecture Federation Development and Execution Process (FEDEP)"*, 2003.

[IEEE 1516] The Institute of Electrical and Electronics Engineers, Inc., IEEE 1516-2000. *« IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) –Framework and Rules »*, 11 December 2000.

[IEEE15288] *15288 Adoption of ISO/IEC 15288:2002 Systems Engineering – System Life Cycle Processes*, IEEE Computer Society. 8 Juin 2005.

[Isermann, 2000] R. Isermann, *« Mechatronic systems : concepts and Applications »*, Transactions of the Institute of Measurement and control 22,1 (2000), pp.29-55.

[Jefferson, 1985] D. R. Jefferson, *"Virtual Time"*, ACM transactions on programming languages and systems, vol. 7, n°3, **pp**. 404-425.

[Kanai, 2003] Kanai S., Shimizu T., *"HLA/RTI-based Scalable Distributed Virtual Prototyping Environment for Embedded System Design"*. In Proceedings of Virtual Concept 2003, Biarritz, France. **pp**. 100-107.

[Khon, 2002] S. H. Khon, H. Zhon , H. S Tan, C. Tank., *"Virtual Environment for Manufacturing and Training (VEMAT)."* Distance Learning and the Internet (DLI 2002), Canberra and Sydney, Australia.

[Kuhl, 1999] Kuhl F., Weatherly R., Dahmann J. ,” *Creating Computer Simulation Systems: An Introduction to the High Level Architecture”*, ISBN 0-13-022511-8, 1999.

[Lee, 2005] Edward A. Lee and Haiyang Zheng, “*Operational Semantics of Hybrid Systems Invited paper in Hybrid Systems”*, Computation and Control: 8th International Workshop, HSCC, LNCS 3414, Zurich, Switzerland, March 9-11, 2005.

[Lightner, 2000] G. M. Lightner, Robert Lutz, and Reed Little. “*The IEEE HLA standards : The evolution from U.S. DoD HLA v1.3.”*, In Proceedings of the 2000 Fall Simulation Interoperability Workshop, 2000.

[Margery, 2002] D. Margery, B. Arnaldi, A. Chauffaut, S. Donikian and T. Duval, *“OpenMASK: {Multi-threaded | Modular} Animation and Simulation {Kernel | kit}: un bref survol”*, AFIG, Lyon, 2002, **pp** 179-188.

[Modelica Specifications, 2007] *Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling Language Specification Version 3.0*, September 5, 2007.

[Mosterman, 1998] P. Mosterman, M.Otter, H. Elmqvist. “*Modeling Petri Nets as Local Constraint Equations for Hybrid Systems Using Modelica.”*, In Proceedings of the Summer Computer Simulation Conference-98, Reno, Nevada, USA, July 19-22, 1998.

[Mosterman, 2002] P.J. Mosterman and G. Biswas. *“A Hybrid Modeling and Simulation Methodology for Dynamic Physical Systems.”* Journal SIMULATION , 78(1):5–17, 2002.

[OgreMax] www.ogremax.com

[Otter, 2005] M. Otter, K.-E. Arzen, I. Dressler, *“StateGraph-A Modelica Library for Hierarchical State Machines”*, Proceedings of the 4th International Modelica Conference, Hamburg, March 7-8, 2005, **pp**. 569-578.

[Pawletta, 2000] Pawletta S. Drewelow W. Pawletta T., *"HLA-based simulation within an interactive engineering environment"*. In Proceedings. Fourth IEEE International Workshop on Distributed Simulation and Real Time Applications (DS-RT 2000), 2000, **pp**.97-102.

[Perret, 2003] J. Perret, G. Hetreux, J.M. Le lann, "*Modélisation des systèmes dynamiques hybrides basée sur le formalisme Prédicats -Transitions- Différentiels-Objets*" , 4e Conférence Francophone de MOdélisation et SIMulation "Organisation et Conduite d'Activités dans l'Industrie et les Services" MOSIM'03 – du 23 au 25 avril 2003 - Toulouse (France).

[Pullen, 1995] Pullen J. M. ; Wood D. C. , *"Networking technology and DIS"*, Proceedings of the IEEE 83 (1995) Aug., No. 8, New York, U.S.

[Raulet, 2003] R. Valéry, *"Prototypage interactif et collaboratif. Vers une architecture de communication pour une interactivité coopérante dynamique dans les environnements virtuels distribués"*, Thèse, 2003, Université de Bretagne Occidentale France.

[Sang, 2001] C. P. Sang, K. C. Byoung, *« Virtual FMS Architecture for FMS Prototyping »*, American Institute of Physics Conference Proceedings, Volume 573, **pp** 628-637, 25 Juin 2001.

[Siron, 1998] P. Siron, ***"****Design and Implementation of a HLA RTI Prototype at ONERA."*, 1998 Fall Simulation Interoperability Workshop, Orlando, September 13-18, 1998.

[Site web 20-sim] http://www.20sim.com

[Site Web ALSP] http://alsp.ie.org/alsp/

[Site Web CERTI] http://www.cert.fr/CERTI

[Site web Charon] http://rtg.cis.upenn.edu/mobies/charon/implementation.html

[Site Web Dymola] http://www.dynasim.se/

[Site Web GENESIS] http://www.onera.fr/dprs/genesis/index.php

[Site Web Mathmodelica] http://www.mathcore.com/products/mathmodelica/

[Site Web OpenMASK] http://www.irisa.fr/bunraku/OpenMASK/

[Site Web OpenModelica] http://www.ida.liu.se/labs/pelab/modelica/OpenModelica.html

[Site Web Virtools] http://www.virtools.com/

[Site Web VRJuggler] http://www.vrjuggler.org/

[Site Web-DMSO-HLA] https://www.dmso.mil/public/transition/hla/

[Soejima, 2000] Soejima S., *"Examples of usage and the spread of Dymola within Toyota"*, Modelica Workshop Proceedings 2000, 23-24 octobre 2000, Lund, Suède.

[Stephen, 2006] S. L. Campbell, J.P. Chancelier et R. Nikoukhah, *"Modeling and Simulation in Scilab/Scicos"*, Livre, Springer, 2006.

[Straßburger, 1998] Straßburger S., Schulze T., Klein U., Henriksen J.O. *"Internet-Based Simulation Using Off-The-Shelf Simulation Tools And HLA"*. In Proceedings of 30th conference On Winter Simulation. P.,1998, **pp**. 1669-1676.

[Torpey, 2001] M. Torpey, D. Wilbert, B. Helfinstine, W. Civinskas, *"Experiences and lessons learned using RTI-NG in a large-scale, platform-level federation"*, in Proceedings of the Spring 2001 Simulation Interoperability Workshop, Orlando, FL, March 25–30, 2001, **pp**. 330–337.

[Turki, 2008] Turki S., *"Ingénierie système guidée par les modèles: Application du standard IEEE15288, de l'architecture MDA et du langage SysML à la conception des systèmes mécatroniques"*, Thèse soutenue le 02/10/2008, Université du Sud Toulon Var.

[Valentin, 1999] C. Valentin-Roubinet , "*Hybrid Systems modelling: Mixed Petri Nets.*" IEEE Conference CSCC'99, Athens, 4-8 july 1999: 3rd IMACS, pages **pp**223-228,. July 1999.

[Valentin, 2005] C. Valentin, HDR, "*Différents angles de vue sur la Modélisation, l'Analyse et la Commande de Systèmes Dynamiques Hybrides*", soutenue le 6 décembre 2005.

[Valentin, 2006] C. Valentin, M. Magos, B. Maschke. "*Hybrid port Hamiltonian systems: from parameterized incidence matrices to hybrid automata*", Nonlinear Analysis, vol. 65:6, pp. 1106-1122, 2006.

[Valentin, 2007] C. Valentin, M. Magos, B. Maschke, "*A port-Hamiltonian formulation of physical switching systems with varying constraints*", Automatica, vol. 43:7, pp.1125-1133, 2007.

[Van der Schaft, 2000] A.J. van der Schaft and J.M. Schumacher." *An Introduction to Hybrid Dynamical Systems*", volume 251 of LNCIS. Springer, London, 2000.

[Zaytoon, 2001] J. Zaytoon,(sous la direction de) « *systèmes dynamiques hybride s* »,Hermès science publications,2001.

[Zimmer, 2006] Zimmer, D., Cellier, F.E., "*The Modelica Multi-bond Graph Library.*", In Proceedings 5th Intl. Modelica Conference, Vienna, Austria (2006) Vol. 2, 559-568.

Annexes

L'approche conservative

L'approche conservative évite strictement les erreurs de causalité. Elle propose de maintenir en permanence un ordre correct des exécutions d'événement, en s'assurant qu'aucun message futur n'aura d'estampille inférieure à son temps logique. Le problème primordial que doit résoudre l'approche conservative est de déterminer quand il est « sûr » de traiter un événement. Plus précisément, si on prend le cas d'un processus qui contient un événement non traité E_{10} avec une estampille égale à T_{10} et si on suppose qu'il n'existe aucun événement qui a une estampille $< T_{10}$, et que le processus peut déterminer qu'il est impossible de recevoir un autre événement avec une estampille $< T_{10}$ alors E_{10} est dit « sûr » puisqu'on peut garantir que traiter l'événement ne va pas mener plus tard à la violation de la règle de causalité locale. Les processus qui contiennent des processus « non sûrs » doivent s'arrêter, quitte à ce que cela entraîne un blocage.

Algorithme Chandy/Misra/Bryant et Null Message

Dans cette proposition **[Bryant, 1977] [Chandy, 1981]**, la synchronisation est réalisée à l'aide des messages estampillés servant à décrire les évènements. Supposons qu'on peut spécifier statiquement les liens entre les processus logiques qui doivent communiquer mutuellement. Chaque processus logique communique avec un autre processus logique à travers un canal. L'envoi des messages doit se faire par ordre croissant d'estampille et on considère que le transport des messages maintient cet ordre. Cela implique que les messages reçus par un processus logique sont dans un ordre croissant de leurs estampilles. Cela implique aussi que l'estampille du dernier message reçu est la limite inférieure des estampilles des messages pouvant être reçus ultérieurement.

Les messages reçus dans un canal entrant peuvent être stockés dans une liste FIFO (First In First Out). Cette liste est triée par ordre croissant puisqu'on considère que le transport des messages maintient l'ordre de l'envoi. Le protocole suivant garantit que

chaque processus logique va traiter les événements dans un ordre croissant des estampilles.

```
Tant que  La simulation n'est pas terminée
             Attendre jusqu'à ce que la liste FIFO contienne au moins un message
             Retirer le message M dont l'estampille est la plus petite
             Horloge := estampille de M
             Traiter M
```

Traitement des événements dans un ordre croissant des estampilles

Puisque les messages dans chaque liste FIFO sont triés dans l'ordre de leurs estampilles, le processus logique peut garantir l'adhérence à la contrainte de causalité locale. Ainsi, un processus logique peut exécuter un événement en ayant l'assurance qu'aucun message d'estampille inférieure n'arrivera par la suite. Cependant, un problème apparaît si l'un des canaux est vide lors de la détermination du prochain message à traiter. Dans ce cas, le processus logique ne peut pas connaître la date du prochain message à recevoir par ce canal et il attend indéfiniment cette information. Ceci se traduit par un blocage (DeadLock).

Pour remédier à ce problème, la notion de *Null Message* a été introduite **[Bryant, 1977]**. Les *Null Message* sont utilisés pour la synchronisation et ne correspondent à aucune activité du système physique. Ils n'ont pas de contenu autre qu'une date. Un *Null Message* avec une estampille T_x, envoyé d'un processus logique A à un processus logique B, est une promesse de la part de A qu'aucun message ne sera envoyé à B avec une estampille inférieure à T_x. Le T_x est le minimum de toutes les estampilles des canaux entrants et l'estampille du premier événement local propre au processus logique.

Ceci est l'idée de base des algorithmes Null Message ou Chandy/Misra/Bryant. Il existe plusieurs variantes du principe *Null Message*. Une variante consiste à envoyer

un *null message* sur chaque canal sortant après l'exécution de chaque événement. Ceci garantit que les processus logiques connaissent toujours les estampilles des différents futurs messages venant des autres processus logiques. Pour utiliser cette approche l'algorithme présenté précédemment doit être remplacé par l'algorithme suivant :

```
Tant que  La simulation n'est pas terminée
             Attendre jusqu'à ce que la liste FIFO contienne au moins un message
             Retirer le message M dont l'estampille est la plus petite
             Horloge := estampille de M
             Traiter M
             Envoyer null message aux processus logiques voisins avec une
             estampille égale au minimum de toutes les estampilles des futurs
             messages (horloge + lookahed)
```

Algorithme utilisant le principe *null message*

Une autre approche basée sur la demande consiste à envoyer un message NULL sur requête d'un processus logique qui cherche à faire cesser sa situation de blocage. Cette dernière approche aide à réduire le trafic de messages NULL bien qu'un délai soit requis pour des messages supplémentaires puisque deux transmissions de messages sont demandées. Plus généralement, cet algorithme mise sur une quantité appelée *lookahead*.

Lookahead : Si un processus logique à un instant T du temps simulé est en mesure de dire qu'il ne produira pas des évènements avant l'instant T+L, alors L est appelé le lookahead.

Pour l'algorithme *null message*, l'estampille des messages NULL peut être le temps courant du processus logique plus son lookahead.
Une dernière approche consiste à définir des barrières logiques ou des fenêtres temporelles entre les processus logiques. L'algorithme Conservative Time Window en est un exemple **[Ayani, 1992]**.

Détection du blocage et récupération

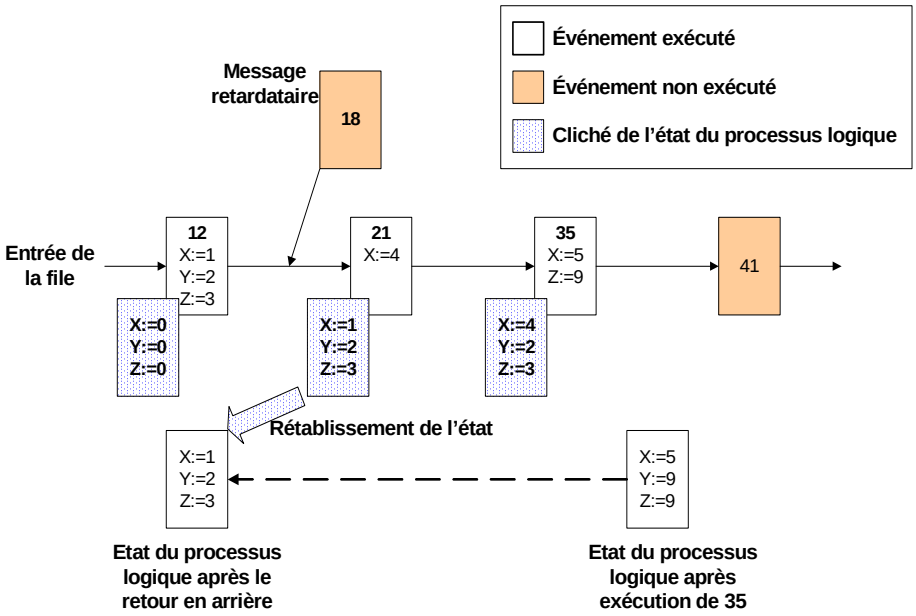
L'inconvénient majeur de l'algorithme Chandy/Misra/Bryant est le grand nombre de messages qui peuvent être générés surtout si le *lookahead* est petit. Les algorithmes se basant sur l'approche conservative ont progressé et ont atteint un état où ils peuvent être implémentés dans des simulations réelles. Néanmoins, des limites de ces algorithmes existent ; en particulier l'inconvénient majeur de l'approche conservative est qu'elle ne peut pas exploiter pleinement la concurrence dans une simulation.

L'approche optimiste

A l'opposé de l'approche conservative, l'approche optimiste permet la violation de la contrainte de causalité locale mais elle fournit un mécanisme pour se rattraper. La notion d'exécution optimiste fait référence au fait que les processus logiques exécutent les évènements de manière optimiste en assumant qu'aucune erreur de causalité ne se produira. Le mécanisme de *Time Warp (*distorsion du temps) de Jefferson était le premier et principal algorithme de synchronisation optimiste **[Jefferson, 1985]**. Dans cette approche, un processus logique exécute les évènements au fur et à mesure de leur réception, son temps logique étant alors la date d'occurrence du dernier message traité. Si à une étape postérieure une erreur de causalité est détectée, le processus de récupération est exécuté, défaisant les effets des événements qui ont été traités (*rollback*). L'événement qui a causé le rollback est appelé *straggler* (retardataire). Le retour en arrière de l'état est accompli en sauvegardant périodiquement l'état du processus. L'état est retourné en arrière à un temps de simulation inférieur ou égal au temps du retardataire. Deux techniques défaisant les modifications des variables d'état sont utilisées par les systèmes *Time Warp* :

- Copie de sauvegarde de l'état :

Un processus logique dans un système Time Warp doit effectuer une copie de sauvegarde de toutes les variables d'état modifiées. La figure suivante présente un exemple du mécanisme de sauvegarde et du processus de retour en arrière.

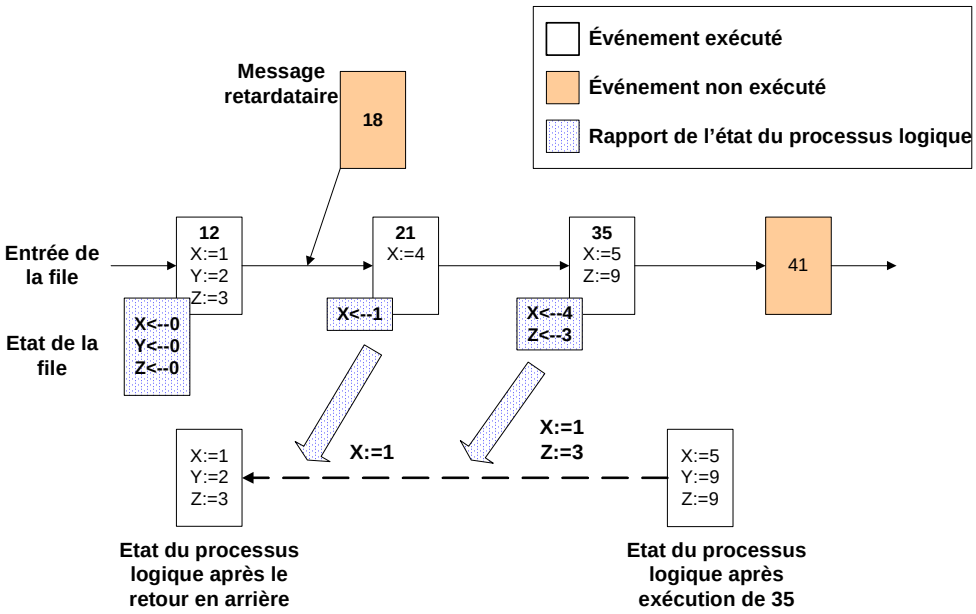


Mécanisme de sauvegarde et de retour en arrière

- Sauvegarde incrémentée de l'état :

Pour chaque exécution d 'un événement, les changements des variables d'état sont enregistrés dans un rapport (*log*). Chaque enregistrement dans le rapport comporte l'adresse de la variable modifiée et la valeur de cette dernière avant modification. Pour faire un retour à l'état avant modification en raison de l'arrivée d'un événement retardataire, le processus logique parcourt les événements à rebours, dans l'ordre décroissant des estampilles. Pour chacun de ces évènements, le processus parcourt le rapport, copie les valeurs enregistrées pour chaque variable d'état correspondante.

L'exemple suivant met en œuvre ce mécanisme.



Sauvegarde incrémentée de l'état

Il faut aussi annuler tout message incorrect envoyé. En effet, un message incorrect peut être déjà exécuté par un autre processus logique, ce qui va générer d'autres messages incorrects et ainsi les erreurs vont se propager dans toute la simulation. Une notion intéressante dans le mécanisme *Time Warp* est la notion d'*anti-messages*.

Une autre notion intéressante est le *Global Virtual Time* qui représente la date minimale d'un retour en arrière. Il est calculé à partir des temps logiques des processus et des estampilles des messages en transit. En conséquence, un processus logique qui doit sauvegarder les états successifs de son passé peut limiter ses sauvegardes grâce au GVT.

Le mécanisme *Time Warp* est le meilleur protocole *optimiste* connu. Plusieurs autres protocoles optimistes ont été développés mais le *Time Warp* reste le plus intéressant en raison de ses concepts fondamentaux qui sont utilisés dans tous les autres algorithmes optimistes. Un système *Time Warp* a quelques insuffisances qui peuvent conduire à des performances modestes pour quelques applications. Plusieurs autres algorithmes de synchronisation optimiste ont depuis été développés pour remédier à ces insuffisances.

Il est à noter qu'il existe d'autres techniques combinant les deux approches optimiste et conservative en fonction du coût d'utilisation d'une méthode ou d'une autre sur la simulation.

Résumé:
La mécatronique est l'intégration de différentes sciences et techniques de la mécanique, de l'automatique, de l'électronique et de l'informatique. L'évolution rapide des marchés concurrents exige la diminution du temps de développement d'un produit tout en augmentant la qualité et la performance du système. Il est donc nécessaire d'augmenter l'efficacité du processus de conception. Pour répondre à cette nécessité, en complément des outils d'analyse, la simulation, et spécialement le prototypage virtuel, est devenu l'une des clés technologiques. Il est difficile de trouver des outils de simulation capables d'analyser des systèmes pluridisciplinaires dépendants de différents domaines. Pourtant, un environnement qui permet une simulation intégrée multidisciplinaire de systèmes mécatroniques est nécessaire pour une évaluation fonctionnelle plus précise de la conception du produit et pour améliorer la qualité et l'efficacité de cette conception. La présente contribution décrit une méthode de conception et de simulation des systèmes mécatroniques. On identifie d'abord le modèle de comportement et le modèle géométrique 3D associé. Ensuite, le modèle de comportement est vu comme un système dynamique hybride formé de deux automates hybrides couplés (Partie Opérative, Partie Commande). Nous présentons ensuite les simulateurs OpenMASK, OpenModelica, le standard IEEE1516 HLA et les travaux reliés à cette architecture de simulation distribuée. Dans une démarche descendante, nous présentons ensuite notre approche et notre expérimentation pour intégrer les fonctionnalités de HLA dans ces simulateurs, pour distribuer les éléments de modélisation de systèmes mécatroniques de haut niveau et enfin pour compléter Modelica sur le formalisme des automates hybrides qui nous est indispensable. Nous proposons des extensions pour intégrer le temps réel en vue de simulations interactives. Nous appliquons enfin cette approche sur les simulateurs cités en utilisant le bus HLA CERTI sous un environnement Linux à partir d'un exemple représentatif d'un système mécatronique.