JEAN-PHILIPPE SHIELDS

Élaboration du modèle conceptuel flexible et extensible d'une architecture logicielle orientée-objet permettant la parallélisation et la distribution d'une architecture de simulation séquentielle

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en Génie Électrique
pour l'obtention du grade de Maître ès sciences (M. Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

Mai 2007

©Jean-Philippe Shields, 2007

Résumé

La parallélisation est une solution possible pour améliorer un simulateur séquentiel s'il devient trop lent dû à une surcharge de calculs et qu'on ne désire pas concevoir à neuf un nouveau simulateur parallèle.

Ce mémoire présente la conception UML™d'une architecture pour un simulateur parallèle flexible et extensible capable de gérer différents environnements de déploiement par une configuration au temps d'exécution. Ce projet a vu le jour dans le but d'améliorer les performances de l'architecture KARMA, une architecture séquentielle pour la simulation d'engagements d'armes, pour réussir à atteindre le temps-réel dans une simulation haute-fidélité avec matériel dans la boucle. L'approche retenue propose une architecture non commerciale et développable à faibles coûts. L'implémentation et les tests préliminaires ont été basés sur un logiciel source libre et portable sur plusieurs plates-formes nommé ACE.

Table des matières

Résumé ii

Table des matières iii

Liste des tableaux vii

Liste des figures viii

1 Introduction 1

1.1 Contexte 1

1.2 Problème 2

1.3 Stratégie 2

2 Revue de littérature 4

2.1 Concepts de simulation discrète 4

2.1.1 Référence : Parallel and Distributed Simulation Systems 4

2.1.2 Référence : Simulation Model Design and Execution - Building Digital Worlds 5

2.2 Discrete Event System Specifications (DEVS) 5

2.2.1 Référence : Theory of Modeling and Simulation (2e Édition) . . 6

2.2.2 Référence : The RTDEVS/CORBA Environment for Simulation Based Design of Distributed Real-Time Systems 6

2.2.3 Référence : Implementation of the DEVS Formalism over the HLA-RTI : Problems and Solutions 7

2.2.4 Référence : Parallel Discrete Event Simulation with Application to Continuous Systems 8

2.3 Simulation discrète orientée-objet temps-réel 8

2.3.1 Référence : Platform-Independance and Scheduling in a Multi-threaded Real-Time Simulation 9

2.3.2 Référence : The Use of Multiple Threads in An Object-Oriented Real-Time Simulation 10

2.4 Parallélisation 10

2.4.1 Référence : Experiences Parallelizing a Commercial Network Simulator . . . 11
2.4.2 Référence : Parallel Execution of a Sequential Network Simulator 12
2.4.3 Référence : Automatic Parallelization of Discrete Event Simulation Program . . . 13
2.4.4 Référence : Parallelizing a Sequential Logic Simulator using an Optimistic Framework based on a Global Parallel Heap Event Queue : An Experience and Performance Report. . . . 14
2.4.5 Référence : Case Study : Parallelizing a Sequential Simulation Model . . . 16

3 Évaluation des outils 18
3.1 Environnements de simulation . . . 18
3.1.1 KARMA . . . 19
3.1.2 FDK . . . 20
3.1.3 HLA . . . 20
3.1.4 SimPack . . . 21
3.1.5 Matlab™, RTW™et Simulink™ . . . 22
3.1.6 LaSRS++ . . . 22
3.1.7 Environnements de simulation commerciaux . . . 23
3.1.8 Autres langages utilisés . . . 23
3.2 Autres architectures logicielles utiles . . . 24
3.2.1 ACE . . . 24
3.2.2 TAO . . . 25
3.2.3 MPI . . . 25

4 Fondements de la simulation discrète 27
4.1 Représentation du temps . . . 28
4.1.1 Temps physique . . . 28
4.1.2 Temps simulation . . . 28
4.1.3 Temps horloge . . . 29
4.2 L'avancement du temps . . . 29
4.2.1 Exécution en temps-réel . . . 30
4.2.2 Exécution en temps-réel proportionnel . . . 31
4.2.3 Exécution aussi vite que possible . . . 31
4.3 L'exécution et l'évolution dans le temps . . . 32
4.3.1 Exécution par pas de temps . . . 32
4.3.2 Exécution par événements . . . 33
4.4 La simulation parallèle et distribuée . . . 35

5 Architecture de simulation de KARMA 37

5.1 Liens entre les modèles dans KARMA 37
5.2 Échange des variables d'états 39
5.3 Exécution séquentielle des modèles 39

6 Processus de parallélisation 41
6.1 Simulation séquentielle 41
6.2 Simulation parallèle 42

7 Les processus logiques 44
7.1 Architectures 44
7.2 Définition 45
7.3 Fils 46
7.4 Design conceptuel 47
7.4.1 ACE_Task 47
7.4.2 ACE_TSS 48
7.4.3 Protocole DEVS 51
7.4.4 Adaptation des Modèles KARMA au Formalisme DEVS 52
7.4.5 Simulateur DEVS conservateur 55
7.4.6 Le parallélisme dans KARMA 61
7.5 Limites et optimisations 64

8 La communication 67
8.1 Communication interprocessus (IPC) 68
8.2 Communication locale 70
8.2.1 Architecture 71
8.2.2 Design conceptuel 72
8.3 Communication distribuée 75
8.3.1 Architecture 77
8.3.2 Design conceptuel 78
8.4 Messages 83
8.5 Limites et optimisations 87

9 L'Application principale 90
9.1 L'environnement 90
9.2 Le contrôleur 92
9.3 L'ordonnanceur 95
9.4 La gestion du temps 98
9.5 La gestion des exceptions 102
9.6 Design flexible 102
9.7 Limites et optimisations 104

10 Application du simulateur parallèle à l'architecture *KARMA* 107

10.1 Avancement des travaux 107
10.1.1 Simulateur parallèle 108
10.1.2 Simulateur distribué 109
10.2 Problèmes rencontrés 110
10.3 Les changements nécessaires à l'architecture de KARMA 112
10.4 Environnements d'analyses et scénarios utilisés 114

11 Conclusion 116
11.1 Résultats obtenus 116
11.2 Travaux futurs et suivi 119

Appendices

A Détails des scénarios utilisés 121

B Details sur la non-causalité de l'architecture KARMA 123

Bibliographie 125

Liste des tableaux

7.1 Exécution séquentielle des modèles pour le scénario spécifié 63

10.1 Problème de la causalité des données 111
10.2 Solution à la causalité des données 112

11.1 Temps d'exécution comparatifs pour le modèle AircraftLauncher entre les deux simulateurs pour un scénario de 2 s 117
11.2 Temps d'exécution comparatifs venant du scénario exécuté sur l'ordinateur à deux processeurs pour une simulation de 2 s 118

B.1 Problème de divergence numérique des données avec le missile 124

Liste des figures

2.1 Architecture du simulateur parallèle *OPNET* 11
2.2 Relation entre le cadre d'applications commun et des langages de simulation séquentielle [TF93] 13
2.3 Adresse mémoire : la mémoire conventionelle vs. la mémoire espace-temps [TF93] 14
2.4 L'interface du cadre d'applications optimiste basé sur la pile parallèle [SKN00] 15
2.5 Topologie simplifié originale (a) et améliorée (b) du simulateur Rumor [BBM99] 17

3.1 Le processus de modélisation et de simulation (*Modeling and Simulation* ou M&S) KARMA [HGJ+04] 19
3.2 Modules du FDK qui forment le RTI 20
3.3 Types de différents modèles possibles [Fis95] 21
3.4 Modules de LaSRS++ [Mad99] 22
3.5 Les composants clés dans ACE et leurs relations hiérarchiques 24
3.6 La configuration logicielle de TAO 25

4.1 Exécution en temps-réel 30
4.2 Exécution par pas de temps 32
4.3 Algorithme d'une simulation à pas de temps précis 33
4.4 Exécution par événements 34
4.5 Algorithme d'une simulation à pas de temps variable 34

5.1 Composition d'un modèle de missile (Baseentity) à l'aide de Parts [BL03] 38
5.2 Diagramme de classe : `Root`, `Entity`, `Baseentity` et `Parts` 38

6.1 Schéma présentant l'exécution actuelle d'une simulation séquentielle dans l'environnement KARMA 42
6.2 Schéma présentant l'exécution souhaitée d'une simulation parallèle dans l'environnement KARMA 43

7.1 Architectures des ordinateurs [Fis95] 45

7.2 Structure de la classe `ACE_Task` contenant un ou plusieurs fils et le lien de communication `ACE_Message_Queue` [Syy] 48
7.3 Structure des participants dans le patron *Thread-Specific Storage* [SHP97] 49
7.4 Diagramme de classe UML™d'une partie du boîtier `LogicalProcess` . 50
7.5 Définition d'un DEVS classique 51
7.6 UML™de l'adaptateur DEVS pour les modèles KARMA basé sur le simulateur classique DEVS 52
7.7 Résumé du protocole du simulateur DEVS [ZPK00] 53
7.8 Algorithme du simulateur classique DEVS [ZPK00] 54
7.9 Structure du simulateur parallèle DEVS conservatif [ZPK00] 56
7.10 Algorithme du simulateur conservatif DEVS [ZPK00] 58
7.11 Diagramme de classe du simulateur conservatif DEVS et du simulateur classique DEVS 61
7.12 Scénario d'engagement entre 2 avions 62
7.13 Exemple de l'exploitation du parallélisme sur plus d'un processeur pour le scénario spécifique de l'engagement des 2 avions. 64

8.1 Synchronisation de deux fils à l'aide d'un mécanisme IPC semblable au principe du producteur-consommateur 69
8.2 Structure de la communication avec les files de messages [Mar99] 71
8.3 Structure de la communication du simulateur local parallèle 72
8.4 Représentation graphique de l'architecture de communication réseau Acceptor-Connector 78
8.5 Structure UML™de la partie Acceptor de la communication réseau . . 80
8.6 Diagramme de séquence de l'`Acceptor` 82
8.7 Structure UML™de la partie `Connector` de la communication réseau . 82
8.8 Structure UML™de la spécialisation de l'objet `ACE_Message_Block` et l'inclusion d'une structure complexe par le pointeur `AbstractData` . . . 83
8.9 Design UML™de la classe `Messages` 86

9.1 Diagramme de classe UML™de l'environnement montrant son lien avec le `Controller` et KARMA 92
9.2 Diagramme de classe UML™du contrôleur et de l'ordonnanceur 93
9.3 Fontions de la classe `SimulationController` 94
9.4 Diagramme de classe UML™du `Scheduler` et de son lien avec le boîtier `TimeManagement` 96
9.5 Fonctions de la classe `SimulationController` 97
9.6 Diagramme de classe UML™du boîtier `TimeManagement` 98
9.7 Fontions de l'interface du `TimeManagement` 99
9.8 Diagramme de classe UML™de la classe `ThreadRecord` 99
9.9 Diagramme de classe UML™de la classe `SequentialAlgo` 100

9.10 Diagramme d'activité de l'algorithme implémentée dans la classe `SequentialAlgo` 101
9.11 Diagramme de classe du boîtier `Exceptions` 102
9.12 Diagramme de classe du patron de conception *Factory Method* 104

Chapitre 1

Introduction

Ce projet de maîtrise a été exécuté en collaboration avec le centre de Recherche et Développement pour la Défense Canada (RDDC) situé à Valcartier. Cette collaboration avait pour but, à travers un projet de recherche, d'élaborer le modèle conceptuel d'une architecture logicielle orientée-objet de simulation discrète événementielle permettant l'exécution de simulations en boucle fermée avec le simulateur SEMAC (Simulateur d'engagement Missile Avion Contre-mesure).

1.1 Contexte

À la base de ce projet de maîtrise, se situent deux autres outils de simulations d'envergures développés auparavant au RDDC.

Le premier est nommé SEMAC et est un banc d'évaluation dynamique pour l'étude de l'interaction entre des cibles simulées et un auto-directeur à guidage infrarouge. Ce simulateur matériel utilise deux sources corps noir combinées optiquement pour simuler les signatures infrarouges et le mouvement relatif de deux objets. Par exemple, ces signatures pourraient soit venir d'un avion ou de leurres si le contexte en est un d'engagement d'armes. Ces signatures sont alors projetées devant un autodirecteur à infrarouge.

Le deuxième est nommé KARMA et est une architecture logicielle qui permet d'exécuter des simulations constructives et virtuelles. KARMA simule des engagements de missiles guidés et intègre la connaissance sur des systèmes d'armes de précision et de

guerre électro-optique. KARMA est aussi un processus et une architecture de modélisation et simulation (M&S) flexibles et extensibles. [HGJ+04] et [HGL+02].

Le projet de maîtrise s'intègre dans le développement d'un nouveau contrôleur de SEMAC qui fermera la boucle entre SEMAC et KARMA pour éviter le dédoublement dans le développement scientifique en permettant la réutilisation des modèles développés dans l'architecture de KARMA sur le simulateur matériel SEMAC.

Le but principal du projet était de produire le prototype d'un environnement de simulation discrète événementielle basé sur l'architecture et les modèles de l'outil de simulation KARMA. Le design souhaité devait intégrer un formalisme de simulation discrète et ajouter la capacité de traiter en parallèle plusieurs modèles dont l'exécution pouvait être concurrente afin d'améliorer les temps de calcul des simulation numériques pour respecter les contraintes temporelles imposées par le banc d'évaluation dynamique SEMAC en boucle fermée. Les tâches principales dans ce projet étaient de défricher le domaine scientifique par une revue de littérature, de procéder à une évaluation et à l'élaboration de concepts et de réaliser un modèle conceptuel. Évidemment, le design conceptuel se devait d'être flexible et portable sur les systèmes d'exploitation Windows et QNX. Windows est la plateforme sur laquelle les développeurs travaillent et QNX est le système d'exploitation utilisé pour contrôler SEMAC.

1.2 Problème

Pour pouvoir fermer la boucle et respecter les contraintes temporelles de SEMAC, les simulations effectuées par l'outil de simulation KARMA devaient subir certains changements : soit retirer certaines composantes dans la simulation, comme supprimer les collisions, soit réduire la fidélité numérique des modèles, comme de remplacer un aérodynamisme à six degrés de liberté par un aérodynamisme à trois degrés de liberté.

1.3 Stratégie

Pour réussir à minimiser le problème des contraintes temporelles, il faut chercher à diminuer le temps d'exécution actuel des simulations. Avec un temps d'exécution diminué, les scénarios d'engagement qui le permettront pourront soit posséder une composition plus complexe et s'exécuter dans la même période de temps qu'auparavant ou soit s'exécuter plus rapidement avec la même composition. Pour réaliser cet objectif,

la stratégie choisie est d'augmenter la puissance de calcul du logiciel grâce à l'ajout de nouvelles capacités permettant l'exécution de simulations parallèles et distribuées dans l'architecture de l'outil de simulation KARMA. L'ajout de ces capacités doit donc permettre aux scénarios possédant des capacités d'exécution concurrente, ceux dont plus d'un modèle peut être simulé parallèlement, de réduire leur temps total de simulation. Cette stratégie doit cependant respecter une contrainte : celle d'affecter le moins possible l'architecture de KARMA pour que les modifications restent le plus transparentes possible pour les utilisateurs et les développeurs. Cette importante contrainte place le projet de recherche dans une catégorie peu explorée jusqu'à maintenant dans la communauté scientifique : la parallélisation. Elle se définit par l'action de paralléliser un outil existant de simulation séquentielle. Certaines recherches ont déjà été accomplies dans le domaine et seront présentées dans la revue de littérature.

Chapitre 2

Revue de littérature

Le démarrage de ce projet demandait certaines connaissances qui ont motivé la poursuite d'une revue détaillée de la littérature scientifique portant sur l'application des concepts de l'orienté-objet au contrôle en temps-réel et à la simulation de type matériel dans la boucle (*Hardware-In-The-Loop* ou HWIL). En fait, il s'agit d'un tour d'horizon des recherches, des travaux et des publications déjà accomplies dans le domaine afin de mieux comprendre les concepts en jeu et d'être en mesure de bénéficier des retombées de travaux similaires.

2.1 Concepts de simulation discrète

Dans cette section se retrouvent plusieurs références touchant le coeur du projet : la simulation discrète. Ces références sont des livres qui présentent les grands fondements de la simulation discrète et ils sont à caractère académique.

2.1.1 Référence : Parallel and Distributed Simulation Systems

Ce livre est un rassemblement de tous les concepts et principes du domaine de la simulation parallèle, mais surtout distribuée et non pas une revue des résultats de ses recherches. Le livre traite en détails des algorithmes disponibles et optimisations possibles pour l'exécution d'une simulation : exécution synchrone conservative et asynchrone optimiste (*time warp*). De plus, le livre analyse en profondeur la gestion du temps, l'ordenancement des événements et les problèmes de distributions tels la latence

réseau, les protocoles de communication et la distribution des données.

Ce livre renferme les principes et règles de base, mais n'entre pas dans les détails au sujet de l'implémentation nécessaire pour réaliser ces concepts, seuls les grandes lignes et idées sont enseignées. Son contenu permet donc d'analyser l'environnement dans lequel une simulation se déroule et donne des solutions possibles pour résoudre certains problèmes, mais l'implémentation peut soulever des difficultés face auxquelles le lecteur doit alors se tourner vers une autre source plus détaillée d'information spécifique dans la littérature scientifique.

2.1.2 Référence : Simulation Model Design and Execution - Building Digital Worlds

Paul A. Fishwick, professeur de modélisation et de simulation à l'Université de Floride, a écrit cet ouvrage [Fis95] portant sur les fondements de la simulation discrète. Fishwick est aussi à l'origine des efforts qui ont conduit au simulateur numérique SimPack discuté à la section 3.1.1. Son livre s'étend sur des sujets d'ordre général comme les techniques de modélisation et les algorithmes de simulation. Cet ouvrage se prête bien à une initiation dans le domaine de la simulation, mais devient une référence dans le domaine de la modélisation. La modélisation discrète y est présentée sous toutes ses formes. Les divers modèles présentés sont illustrés à la figure 3.3. Ce livre offre aussi un bref aperçu des possibilités de la simulation parallèle et distribuée, mais toujours sous un angle de modélisation.

2.2 Discrete Event System Specifications (DEVS)

Cette section propose l'analyse de certains ouvrages portant sur une branche spécifique de la simulation discrète soit celle concernant la théorie des DEVS ou son application. La théorie des DEVS propose des formalismes de modélisation et des algorithmes de simulation correspondant à plusieurs situations de simulation : séquentielle, parallèle, distribuée et temps-réel. Ces formalismes permettent d'établir un meilleur standard de modélisation en offrant des détails techniques sur les interactions entre les différents modèles dans plusieurs situations.

2.2.1 Référence : Theory of Modeling and Simulation (2e Édition)

Ce livre présente la théorie sur les événements et la simulation discrets et, en particulier, trois formalismes de modélisation : les *Discrete Event System Specification* (DEVS), les *Discrete Time System Specification* (DTSS) et les *Differential Equation System Specification* (DESS). La simulation de ces formalismes et les protocoles d'échange de données et d'information sont présentés avec détails dans plusieurs exemples et tous les algorithmes nécessaires au bon fonctionnement des modèles sont expliqués. La simulation parallèle et distribuée des DEVS est décrite et certains dérivés de ses formalismes comme les DEVS temps-réel (*Real-Time DEVS*, RT-DEVS) sont aussi présentés.

Cette référence comble un vide important sur les principes de la simulation discrète en développant des formalismes qui détaillent les interactions des différents modèles à travers plusieurs algorithmes convenant à diverses situations et environnements. Ce formalisme serait parfait pour imposer une interface et un mode d'utilisation pour tous les modèles développés au RDDC Valcartier. De plus, ces modèles pourraient alors être réutilisable et interchangeables selon un formalisme connu et bien établi mondialement.

2.2.2 Référence : The RTDEVS/CORBA Environment for Simulation Based Design of Distributed Real-Time Systems

Cette référence [YXBP03] décrit comment les modèles d'un système distribué temps-réel peuvent être simulés et testés avec RTDEVS/CORBA avant d'être exécutés dans l'environnement temps-réel original. Cette référence démontre qu'un système de simulation basé sur le formalisme RTDEVS peut respecter les exigeantes contraintes requises par un système temps-réel.

Le système présenté dans la référence est implémenté sur le logiciel des couches intermédiaires (*middleware*) TAO, discuté à la section 3.2.2, et donc sur le standard CORBA [OMG95]. Les auteurs prétendent que ce logiciel leur permet de respecter plusieurs contraintes d'un système temps-réel comme la synchronisation du temps dans un environnement distribué, la livraison de données à l'intérieur d'un intervalle de temps précis et, en général, de garantir une bonne qualité de services (*Quality of Service* ou QoS). Cette référence démontre clairement qu'un système distribué temps-réel peut être construit à partir du formalisme DEVS, sous son extension temps-réel, sur le standard CORBA.

Cependant, les auteurs spécifient que le modeleur doit s'assurer que ses modèles peuvent exécuter leurs activités dans un temps définit qu'il doit connaître et donner en paramètre au modèle. Le système ne règle pas le problème de la gestion dynamique des tâches, mais s'assure que les temps morts, qui sont indispensables, sont bien gérés pour que le temps de simulation soit exactement égal au temps de l'ordinateur et que tous les modèles soient bien synchronisés.

Cette référence pourrait donc servir de modèle pour l'implémentation de la partie distribuée du simulateur avec KARMA. Comme il est basé sur le formalisme DEVS, ce système permetterait la réutilisation de tous les modèles KARMA s'ils sont adaptés DEVS et leur utilisation dans un contexte de distribution. Cependant, comme le mentionne les auteurs, dans tous les cas, la simulation globale sera toujours limitée par le rapport des activités à accomplir versus la puissance de calcul fournie.

2.2.3 Référence : Implementation of the DEVS Formalism over the HLA-RTI : Problems and Solutions

Cette référence [BPGH+99] décrit la technique employée pour développer le principe DEVS/HLA, un environnement de simulation compatible à HLA, présenté à la section 3.1.3, et utilisant le formalisme haut niveau des DEVS. La référence concentre ses indications sur l'implémentation de l'environnement de simulation en C++ et les problèmes survenus avec cette approche.

La référence explique comment les auteurs ont relié les entrées et les sorties des deux logiciels par une interface modulaire et comment, en se basant sur les informations des modèles DEVS, l'environnement DEVS/HLA se charge des déclarations, de la création des fédérations, de leurs commmunications et de la gestion du temps. Les auteurs ont cependant conclus qu'il n'est pas possible d'implémenter directement le protocol DEVS à partir du HLA et que pour réussir, il fallait implémenter un coordonnateur formel entre les différents fédérés servant d'intermédiaire.

DEVS/HLA est un simulateur distribué, comme pourrait l'être KARMA, utilisant des modèles respectant un formalisme DEVS et exécutés dans un environnement utilisant le standard HLA. Présentement, les modèles numériques développés pour l'outil de simulation KARMA peuvent être utilisés dans une simulation HLA grâce l'outil de simulation commercial développé par CAE nommé STRIVE. Cependant, les simulation numériques produites par l'environnement de simulation de KARMA ne sont pas compatibles au standard HLA. Il est donc possibe d'imaginer, dans l'avenir, que l'outil

de simulation KARMA soit capable de produire des simulations distribuées compatible HLA si le formalisme DEVS est introduit pour modéliser les interactions de l'architecture.

2.2.4 Référence : Parallel Discrete Event Simulation with Application to Continuous Systems

Cette thèse de doctorat de J. Nutaro [JJ03], un étudiant de Zeigler, discute des simulations discrètes parallèles et discrètes distribuées appliquées aux systèmes continus.

Cette référence décrit principalement un environnement logiciel de simulation distribuée utilisant des algorithmes optimistes. Bien que la majorité de la thèse développe des sujets concernant les DEVS, leur utilité est peu pertinente au projet du simulateur parallèle KARMA puisque la simulation est conservatrice. Une section plus particulière, concernant les horloges logiques et les algorithmes de temps, peut cependant se révéler très instructive.

Cette section discute en détail des techniques pour déterminer le temps logique atteint par chaque processeur dans son exécution, ce qui permet de trouver le temps minimum global dans l'exécution de la simulation. Certaines techniques discutées pour déterminer le temps virtuel global de la simulation (GVT) ne sont utiles que dans un contexte de distribution optimiste et servent à repérer et à réparer les incohérences temporelles. Les algorithmes et techniques décrites et analysées dans cette thèse seraient aussi utiles dans la conception de la section distribuée du simulateur KARMA.

2.3 Simulation discrète orientée-objet temps-réel

Cette section présente deux articles sur un environnement de simulation construit par la National Aeronautics and Space Administration (NASA). Cet environnement de simulation est relativement semblable à KARMA. Ces articles ciblent donc très bien le sujet du projet de maîtrise.

2.3.1 Référence : Platform-Independance and Scheduling in a Multi-threaded Real-Time Simulation

Cette référence [SRK01] décrit le design d'un cadre d'applications (*framework*) logiciel nommé LaSRS++, discuté à la section 3.1.6, construit par la NASA pour simuler numériquement en temps-réel le comportement de modèles dans le domaine de l'avionique. Le design proposé dans la référence permet un usage portable et réutilisable de fils (*threads*) et offre aussi un mécanisme qui automatise l'équilibrage de charges des modèles de la simulation.

Cette référence, et plusieurs autres ouvrages sur portant le même environnement de simulaton, est très intéressante car elle décrit un environnement de simulation relativement comparable à l'architecture de KARMA. Cet environnement, LaSRS++, simule majoritairement des modèles d'avions et leur comportement dans différentes situations et fonctionne sur un ordinateur multi-processeur symétrique (*Symetric Multi-Processor* ou SMP).

La référence présente plusieurs techniques d'optimisations des calculs aidant à l'optention de simulations respectant le temps-réel. La planification de l'exécution des fils et la répartition des charges sont accomplies par un algorithme pré-simulation. Cette phase analyse le temps de calcul de chaque entité dans la simulation et retient une estimation de la durée maximale d'exécution de toutes les tâches. Par la suite, l'algorithme calcule la répartition des tâches qui permet de respecter les contraintes de temps-réel et, si cela est impossible, en informe l'utilisateur.

Les systèmes d'exploitation choisis pour exécuter la simulation doivent répondre à certains besoins comme pouvoir générer des fils à portée globale (*system-scope threads*) à l'intérieur d'un processus bloqué en mémoire RAM, pouvoir exécuter les fils sur des processeurs spécifiques et pouvoir empêcher la préemption des fils qui s'exécutent. Cependant, le système d'exploitation que les développeurs de KARMA utilisent, Windows, ne satisfait pas tous ces besoins. L'implémentation du design proposé dans l'article devient alors très complexe sur le système d'exploitation Windows. De plus, dans la référence, seuls la portabilité des fils et leur assignement à un processeur sont démontrés par l'usage des patrons de conception *Bridge* et *Factory* [GHJV95]. Qu'en est-il de la mémoire partagée et des mécanismes de communication inter-processus (IPC) qui ne sont pas nécessairement portables sur tous les systèmes d'exploitation ? La référence n'en fait pas mention.

2.3.2 Référence : The Use of Multiple Threads in An Object-Oriented Real-Time Simulation

Cet article [Gey99], écrit par l'Institut Américain d'Aéronautique et d'Astronautique (AIAA), présente l'environnement de simulation logiciel LaSRS++. Cette référence analyse beaucoup plus les besoins qui ont mené à l'utilisation de fils et de leur implémentation dans l'environnement de simulation orienté-objet que des optimisations possibles pour atteindre un but précis comme le temps-réel.

Cette référence explique le cheminement fait par les auteurs entre l'analyse de leurs besoins pour une plus grande puissance de calcul et la solution finale implémentant l'utilisation des fils pour offrir un parallélisme à leur environnement de simulation. Les auteurs discutent des problèmes rencontrés comme l'échange d'information en mémoire partagée, la synchronisation des différents fils et la synchronisation de la simulation avec une horloge externe. Ils présentent aussi les solutions qu'il ont employées lors du design final de l'environnement. De plus, ils proposent plusieurs manières de garder un bon synchronisme avec tous les fils dans le processus principal à l'aide de barrières, mutex et sémaphores. L'environnement de simulation comprend même un fil supplémentaire réservé pour l'interface graphique et utilisé avec une moindre priorité pour interagir avec la simulation. Bien que l'article soit une présentation sommaire de la réalisation d'un environnement supportant les traitements multifils (*multithread*) de simulation et présente peu de détails techniques, il fournit des notions intéressantes sur l'analyse des besoins et l'évaluation des concepts possibles. Cependant dans le projet de parralélisation du simulateur d'engagement KARMA, le design de bas en haut d'un environnement de simulation n'est pas dans les options envisageables, puisque cette option s'avérerait une tâche trop colossale. Le but est d'adaptater un système parallèle à une architecture déjà existante. Même si le cheminement peut paraître à prime à bord semblable, le design final sera probablement très différent en raison des contraintes très différentes.

2.4 Parallélisation

Cette section traite des références scientifiques sur des expériences semblables au projet effectué sur l'outil de simulation KARMA, soit partir d'un simulateur séquentiel et de lui ajouter des capacités supplémentaires pour lui permettre une exécution parallèle ou distribuée. Les efforts déployés dans ce domaine ont pour but de permettre des parallélisations transparentes à l'utilisateur et aussi d'éviter les changements majeurs dans l'architecture originale des simulateurs séquentiels. Cette approche, évite au

modeleurs d'avoir à maîtriser de nouvelles techniques pour être compatibles avec les changements apportés dans l'engin de simulation.

2.4.1 Référence : Experiences Parallelizing a Commercial Network Simulator

Cette référence [WFR01] décrit la méthodologie employée pour étendre les capacités d'un simulateur de communication réseau séquentiel, nommé OPNET [Tec], en simulateur parallèle. Le modèle de l'architecture HLA a été utilisé, ce qui permet de faciliter l'interopérabilité et la réutilisation des modèles de simulation pour offrir les services de communication et de synchronization entre tous les fédérés (modèles de simulation). Les communications entre les modèles situés sur des processeurs différents utilise un serveur mandataire (*proxy*) qui implemente les interactions de manière transparente. L'implémentation de ce projet exploite le *Federated simulations Development Kit* (FDK) développé à Georgia Tech et, plus spécifiquement, le *Run-time Infrastructure* (RTI) pour les services de gestion d'événements et de gestion du temps. L'architecture du simulateur est montrée à la figure 2.1.

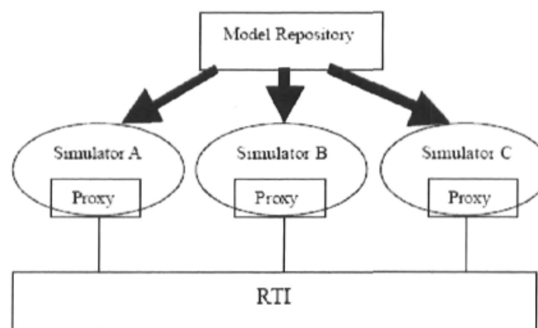

FIG. 2.1 – Architecture du simulateur parallèle *OPNET*

L'article mentionne une faiblesse dans l'implémentation qui se situe au niveau de la valeur de l'anticipation (*lookahead*) qui, dans plusieurs cas, se trouve à prendre la valeur zéro. L'anticipation est défini comme étant la prochaine valeur de temps possible à laquelle le modèle pourra produire un événement externe. Cette valeur empêche donc la simulation d'avancer rapidement dans le temps puisque les incréments de temps deviennent minuscules. Pour cette raison, tous les protocoles disponibles dans le simulateur réseau n'ont pas été parallélisés, seulement ceux qui comportaient moins de problèmes à adapter au FDK et RTI.

Dans cet article, le simulateur de départ contenait des fonctionnalités pour la gestion des événements, c'est-à-dire que les communications de base et les échanges d'information reposaient déjà sur des principes correspondant à la simulation discrète par événements. Cette particularité rendait, dans ce cas, la tâche beaucoup moins compliquée que dans l'architecture KARMA puisque l'implémentation des algorithmes de haut niveau pouvait alors se faire sans trop de difficulté. Cependant, l'idée d'utiliser un serveur mandataire pour interfacer le RTI au simulateur séquentiel est un point qui sera maintenu dans le simulateur parallèle car il est évident que l'interface des modèles numériques de KARMA, discutés à la section 5.1, devra être adaptée aux algorithmes et protocoles parallèles et distribués.

2.4.2 Référence : Parallel Execution of a Sequential Network Simulator

Cet article [JD00] présente aussi l'extension des capacités d'un simulateur de réseau séquentiel, nommé ns, en simulateur parallèle. Cet article est très semblable au précédent (section 2.4.1) à l'exception que les interactions entre les modèles sur des processeurs différents ont été implémentées à partir d'une variante du protocole des *null messages* qui se classe dans la section des algorithmes de synchronisation conservative. Ces messages *null* ou "vide" ne correspondent pas à des événements de la simulation, mais servent à propager une valeur minimum sur le temps d'exécution du prochain événement en utilisant le principe de l'anticipation[1]. L'implémentation du système de communication parallèle est basée sur le standard d'échange de messages *Message Passing Interface* (MPI) [Sta] avec la distribution MPICH [Imp].

À cause de l'utilisation de MPI et de la parallélisaton des modèles sur plusieurs processeurs, les chercheurs ont dû apporter des changements à la syntaxe ns pour refléter le partionnement et l'emplacement des modèles sur les différents processeurs utilisés. Cette dernière option semble cependant devoir être entrée manuellement par l'utilisateur. Les résultats obtenus sont dépendants de certains facteurs inhérents à la simulation parallèle et distribuée : la valeur de l'anticipation des modèles et le nombre de connexions entre les différents modèles ou processeurs. De plus, la technique des *null messages* peut provoquer un goulot d'étranglement au niveau des liens de communication si trop de messages sont produits.

Cet article montre combien les résultats obtenus sont dépendants de l'anticipation et du nombre d'événements générés par les modèles. Plus il y a d'événements et plus

[1]Pour plus de détails voir [CM79] et [Bry77]

l'anticipation est grande, plus les processeurs ont des chances de trouver des événements à exécuter en parallèle, jusqu'à une saturation, et donc d'améliorer le temps total d'exécution d'une simulation. Il est à noter que dans une simulation d'engagement avec KARMA peu de modèles sont instanciés en même temps et les simulation sont généralement de courte durée, ce qui pourrait être une limitation au gain en vitesse d'exécution pour l'implantation parallèle.

2.4.3 Référence : Automatic Parallelization of Discrete Event Simulation Program

Cette référence [TF93] présente un outil développé dans le but de convertir une simulation séquentielle en simulation parallèle à partir du code original du simulateur séquentiel. Les auteurs ont développé une méthode qui automatise la conversion du code de simulation par l'entremise d'un compilateur spécial. Une étude de cas, à partir du langage SIMSCRIPT II.5, est présentée. Leur approche consiste à utiliser un cadre d'applications commun sur lequel peut être appliqué une variété de programmes de simulation discrète séquentielle comme présenté à la figure 2.2.

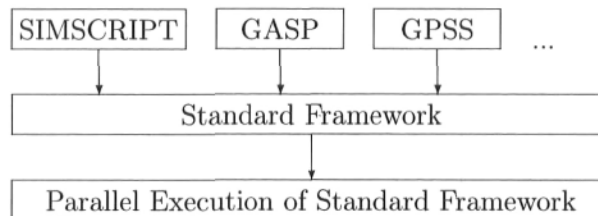

FIG. 2.2 – Relation entre le cadre d'applications commun et des langages de simulation séquentielle [TF93]

La technique utilisée par les auteurs implique un langage spécifique qui oblige les modeleurs à apprendre de nouveaux outils et de nouvelles techniques de travail. Cette obligation peut donc être contraignante pour les modeleurs et l'utilisation de cette approche n'est donc pas souhaitable. Dans le cas présent, comme KARMA est déjà implémenté dans un langage de haut niveau, le C++, non supporté par la technique décrite ci-haut, son exploitation est pratiquement impossible.

Cette approche révèle cependant une option intéressante qu'il est judicieux de souligner. Premièrement, pour réussir à paralléliser correctement les programmes, le compilateur utilise une abstraction appelée la mémoire espace-temps (*space-time memory*) présenté à la figure 2.3.

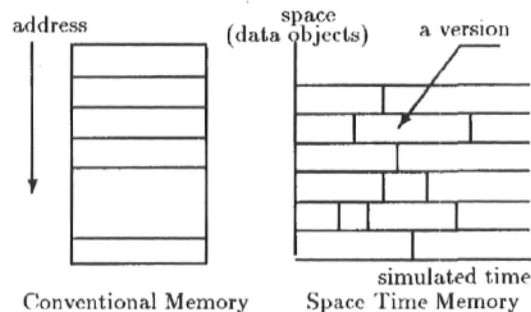

FIG. 2.3 – Adresse mémoire : la mémoire conventionelle vs. la mémoire espace-temps [TF93]

Cette mémoire assure la cohérence et la persistence des variables d'états du début à la fin d'une simulation pour que l'environnement optimiste puisse y référer en cas d'inconsistance temporelle. Cette technique revient à garder en mémoire partagée toutes les valeurs ou tous les changements survenus aux différentes variables d'état puisqu'une simulation optimiste peut exécuter différents événements possédant différentes estampilles temporelle (*timestamp*) au même instant. Cette structure de données est supportée par certains mots clés, appelées primitives, dans le langage utilisé comme MakeObj (size), ReadObj (var_id), WriteObj (var_id, value) et EraseObj (var_id).

Bien que cet article traite principalement de la parallélisation pour une simulation optimiste et non une simulation conservatrice comme il est prévu d'instaurer dans l'environnement de simulation KARMA, le fait de devoir relier les variables d'états à une valeur spécifique du temps de la simulation sera inévitable pour l'implantation d'un environnement de simulation parallèle ou distribuée dans lesquelles plus d'un événements peuvent être exécutés en même temps.

2.4.4 Référence : Parallelizing a Sequential Logic Simulator using an Optimistic Framework based on a Global Parallel Heap Event Queue : An Experience and Performance Report.

Cette référence [SKN00] présente un autre cas de parallélisation d'un simulateur séquentiel mais en utilisant cette fois un cadre d'applications optimiste basé une liste d'événements globale implémentée par une pile parallèle (*parallel heap*). L'article spécifie que le code original et les structures de données de base du simulateur séquentiel demeurent intacts, ce qui implique un bon développement d'adapteur pour modéliser les nouveaux comportements à partir des anciens. L'article propose aussi une méthode

avec des étapes bien définies qui peut être utilisée pour paralléliser d'autres simulateurs d'événements discrets.

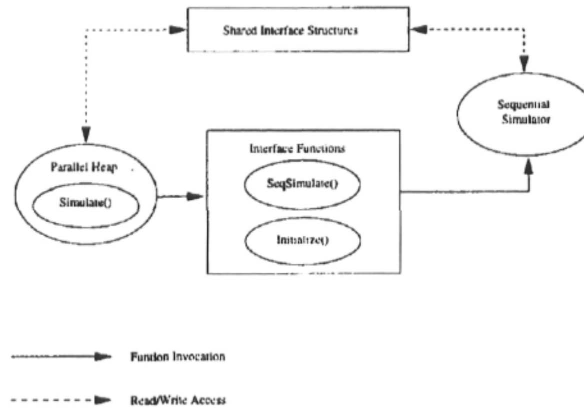

FIG. 2.4 – L'interface du cadre d'applications optimiste basé sur la pile parallèle [SKN00]

Le cadre d'applications optimiste agit comme un client qui utilise les services originaux du simulateur séquentiel pour exécuter la simulation en parallèle. Comme le montre la figure 2.4, des fonctions interfacent les liens entre les deux simulateurs (*Interface Functions*). Par la suite, sur le lien de retour, les données et variables d'états, se transmettent par l'entremise d'une pile parallèle (*shared Interface Structures*). L'approche de parallélisation mentionnée dans l'article peut être décomposée en 4 points principaux.

- Identification des composants de base du simulateur séquentiel.
- Création de nouvelles structures servant d'adapteurs autour des structures originales implémentant la gestion des événements et la simulation séquentielle.
- Modifications à l'algorithme d'exécution des événements pour qu'il soient redirigés dans la liste des événements parallèles.
- Intégration du code du simulateur séquentiel dans le cadre d'applications optimiste.

L'approche proposée d'utiliser un client qui exploite les services du simulateur devrait être exploitée dans le développement de l'environnement parallèle pour KARMA, car un des objectifs est de modifier le moins possible le code original du simulateur séquentiel ou de rendre les changements le plus transparent. Cette approche permettrait aux modèles, les clients, d'utiliser une structure de donnée globale dans l'environnement parallèle comme la liste des événements au lieu d'utiliser l'environnement de simulation pour inspecter chaque modèle comme présentement implémenté dans KARMA.

Cela permettrait un découplement entre les modèles et l'engin de simulation. Cependant, la référence explique très vaguement l'approche de parallélisation employée par les chercheurs et est donc difficilement reproduisible. De plus l'utilisation de la structure de donnée parallèle semble difficilement applicable à l'environnement de KARMA puisqu'elle est adaptée, dans la référence, pour un cadre d'applications utilisant un algorithme optimiste. Dans le prototype développé pour l'outil se simulation KARMA, il serait préférable d'utiliser une struture de donnée séquentielle afin de simplifier le développement.

2.4.5 Référence : Case Study : Parallelizing a Sequential Simulation Model

Cet article [BBM99] décrit une étude de cas sur la parallélisation d'un modèle séquentiel d'un système de fichiers par copie. Ce modèle a été développé pour être simulé dans un environnement de simulation nommé PARSEC. Bien que cet environnement supporte autant les algorithmes d'exécution séquentiels que parallèles, les auteurs soutiennent que la convertion d'un modèle d'un type à l'autre n'est pas chose courante et triviale si l'on veut obtenir de bonnes performances. Les changements les plus importants et cruciaux dans le modèle sont la simplification de la topologie de communication, le réajustement des spécifications de l'anticipation et des modifications au niveau du modèle pour obtenir plus de performance.

Le modèle de système de fichier par copie, nommé Rumor, a été parallélisé en utilisant le protocole des *null messages*, un algorithme d'exécution conservateur. L'emploi de ce protocole a forcé les auteurs à revoir la topologie des communications car une forte densité de liens de communication aura tôt fait de limiter les performances à cause de la saturation des communication dû aux trop grand nombre de *null messages* envoyés pour l'évaluation du plus petit temps d'entrée de chaque modèle. Une autre contrainte rencontrée par les auteurs est la précision de l'anticipation. Dans une simulation parallèle, il est générallement reconnu que les performances augmentent avec une valeur plus grande de l'anticipation. Ce dernier point est probablement le plus intéressant de l'article, car il démontre comment les auteurs ont procédés pour créer cette abstraction, l'anticipation, et comment ils ont changé l'architecture pour en tirer le plus de performance.

La figure 2.5 montre comment la structure de réseau a été découpée en deux couches à la figure 2.5 (b) pour mieux spécifier l'anticipation par rapport aux envois et aux réceptions de messages qui étaient préalablement fusionnés comme le montre la figure 2.5

(a). La bidirectionnalité des liens de communication dans la première version contraignait les deux fonctions ; envoi et réception, à adopter la même valeur de l'anticipation qui représente, en quelque sorte, la latence du réseau. Dans la deuxième version, seule la fonction envoi possède une valeur de l'anticipation, ce qui a grandement amélioré les résultats [BBM99].

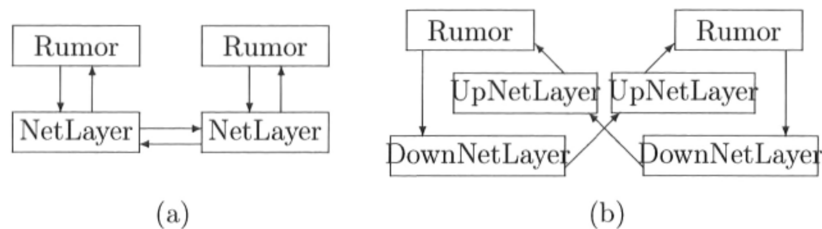

FIG. 2.5 – Topologie simplifié originale (a) et améliorée (b) du simulateur Rumor [BBM99]

Dans KARMA, il n'existe pas vraiment de topologie de communication entre les modèles puisque le point central d'échange de données est le Théâtre d'engagement. Un système de base devra donc être conçu pour permettre la distribution éventuelle sur plusieurs ordinateurs. La technique utilisée par les auteurs pour améliorer les spécifications de l'anticipation pose aussi un problème de taille puisqu'ils ont directement intégré cette varaible dans le coeur même des modèles. Dans KARMA, il est impossible et non désiré, selon les spécifications du projet, de changer les modèles utilisés. Ces modèles ont été validés par des scientifiques et aucune modification ne devrait être apportée, il faudra donc trouver une autre méthode ou un autre endroit comme un adapteur supplémentaire entourant les modèles pour inclure l'anticipation.

Chapitre 3

Évaluation des outils

Cette partie de l'analyse consistait à faire le tour d'horizon de ce qui s'est fait par le passé et de qui se fait présentement au niveau des cadres d'applications de simulation et des langages utilisés pour programmer ce type d'application. De plus, cette activité a permis de répondre à certaines questions soulevées quant à la réalisation technique de quelques objectifs comme l'indépendance envers le système d'exploitation et l'implémentation d'une architecture flexible et extensible.

3.1 Environnements de simulation

Les environnements de simulations ne sont pas rencontrés fréquemment dans le domaine public. Ceux qui sont disponibles gratuitement sont majoritairement le fruit de travaux universitaires ou de travaux gouvernementaux. Plusieurs simulateurs commerciaux sont aussi disponibles pour des sommes d'argent imposantes, ces simulateurs ne seront pas couverts dans cette section d'analyse. De plus, il est à noter que l'utilisation de ce genre d'architecture de simulation commerciale rend dépendant ses utilisateurs des compagnies propriétaires et empêche souvent les changements impromptus en plus de limiter les programmeurs aux interfaces de programmation des applications (*Application Programming Interface* ou API) déjà offertes par ces environnements.

3.1.1 KARMA

KARMA est un environnement de simulation constructif conçu par les chercheurs du RDDC Valcartier pour l'étude des engagements d'armes à l'aide de modèles de différents niveaux de fidélité. Le projet s'est amorcé à partir de cette plateforme et les premiers résultats ont aussi pu voir le jour grâce à cet outil de simulation déjà développé. KARMA a aussi servi de point de comparaison pour les travaux réalisés dans le cadre de ce mémoire.

L'architecture du simulateur d'engagement d'armes, saisie à l'aide du Unified Modeling Language (UML™) [Groc], contient les concepts majeurs pour la simulation des différents composants. Les modèles physiques, quant à eux, peuvent être développés soit en MATLAB^{TM} ou en C++ avec l'aide de la génération de code automatique. La flexibilité de l'architecture et la modularité des scénarios et des modèles simulés viennent de l'utilisation intensive de fichiers XML. Le développement de l'architecture et des modèles de KARMA sont basés sur le principe d'une architecture guidée par modèles (*Model Driven Architecture* ou MDA™) [Gro04] proposé par la communauté du génie logiciel [HGJ+04]. De ce principe, le RDDC Valcartier a développé un processus qui applique le développement guidé par modèles (*Model Driven Development* ou MDD) pour assurer l'interopérabilité, la réutilisabilité et la modularité des modèles développés par les chercheurs [HGL+02]. Le processus générique est présenté à la figure 3.1.

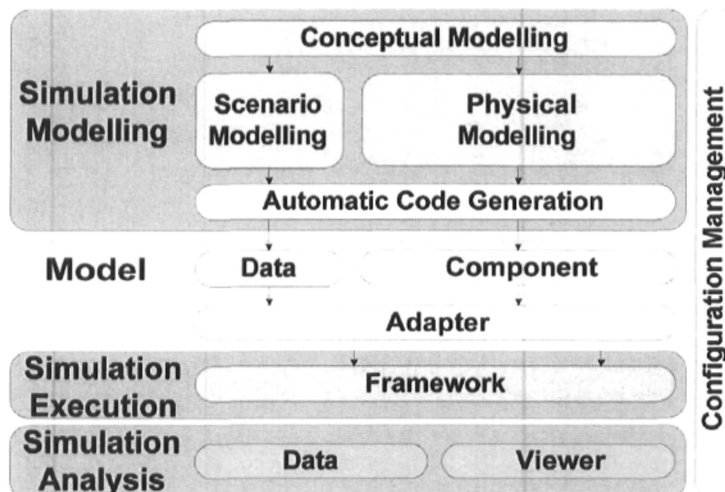

FIG. 3.1 – Le processus de modélisation et de simulation (*Modeling and Simulation* ou M&S) KARMA [HGJ+04]

3.1.2 FDK

Le *Federate Simulations Development Kit* (FDK) a été développé à l'institut de recherche de Georgia Tech aux États-Unis par le groupe *Parallel And Distributed Simulation* (PADS) sous la direction du professeur Richard Fujimoto. Cet environnement est composé de plusieurs modules indépendants conçus pour construire un RTI qui permet d'intégrer et de relier plusieurs simulations différentes. Le FDK a été développé de manière à ce que les programmeurs de RTI puissent implémenter seulement les différents modules nécessaires à leurs applications. Les développeurs de RTI ont tout à gagner à réutiliser les modules déjà développés et testés pour éviter de perdre leur temps à tout reprogrammer [Cor97].

La figure 3.2 illustre les différents modules du FDK et leurs utilisations dans une simulation. Il est a noter que le FDK fait aussi partie d'une structure de simulation plus imposante appelée HLA qui sera présentée à la section 3.1.3.

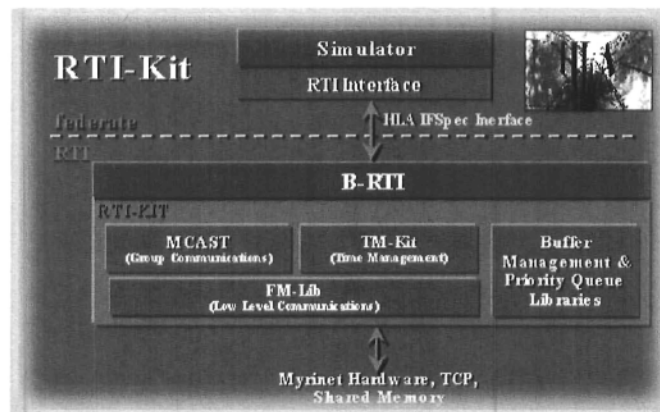

FIG. 3.2 – Modules du FDK qui forment le RTI

3.1.3 HLA

Le *High Level Architecture* (HLA) est une architecture logicielle de base conçue pour la réutilisation et l'interopérabilité des simulations. Le HLA a été développé par l'Office de Modélisation et de Simulation de la Défense américaine (DMSO) pour supporter la réutilisation et l'interopérabilité des différents types de simulations développées et maintenues par le département de la défense américaine (*Departement of Defense* ou DoD). Le HLA a été adopté comme étant l'outil pour distribuer des systèmes de simulation (Facility for Distributed Simulation Systems 1.0) par le *Object Management*

Group (OMG) en Novembre 1998. Le HLA est aussi un standard public approuvé par l'institut des Ingénieurs Électrique et Électronique (IEEE) - Standard IEEE 1516 - en Septembre 2000 [MD96].

3.1.4 SimPack

SimPack est un outil de simulation employé par Paul A. Fishwick pour accompagner son livre [Fis95] sur ls simulation discrète. Cet outil écrit majoritairement dans le langage C (une partie seulement est disponible écrite en C++ SimPack++), simule numériquement plusieurs types de modèles : conceptuels, déclaratifs, fonctionnels, spatiaux et contraints. La figure 3.3 présente tous ces types de modèles.

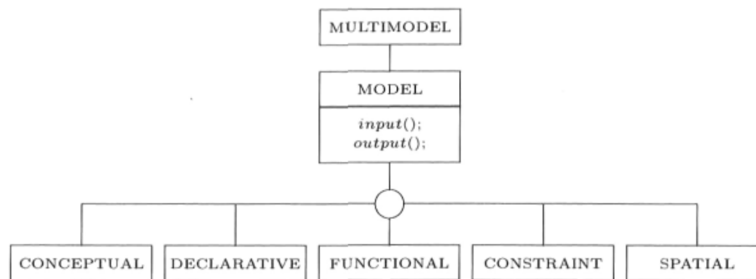

FIG. 3.3 – Types de différents modèles possibles [Fis95]

SimPack est un produit gratuit sur lequel s'appliquent certains droits d'auteur, il est distribué sous la licence GNU. SimPack est utilisé depuis plusieurs années pour enseigner la simulation numérique à l'Université de Floride au niveau du baccalauréat et des études supérieures. Le but de cet outil est de montrer quelles sont les bases de la simulation numérique, mais surtout comment modéliser les différents systèmes et leurs interactions. Présentement, SimPack est toujours utilisé, mais avec un nouvel environnement graphique développé à la même université sous le nom de MOOSE.

Comme SimPack est un simulateur séquentiel, son utilisation n'était pas requise ni souhaitée pour le projet de parallélisation. Comme SimPack est un simulateur complet, il ne prend que des formes très spécifiques de modèles, il aurait été inutile de tenter d'adopter ceux de KARMA et l'utilisation des techniques de simulation de SimPack dans KARMA n'aurait pas été une meilleure idée puisque les deux architectures sont trop différentes. Cependant, la diversité des types de modèles simulés et leur interface avec l'engin de simulation pourraient être intéressants pour l'équipe KARMA dans l'avenir si la nécessité d'incorporer d'autres types de modèles se présente [1].

[1]Présentement, KARMA ne fonctionne qu'avec des modèles de type fonctionnel

3.1.5 Matlab™, RTW™et Simulink™

Bien que Matlab™, RTW™et Simulink™soient des outils commerciaux, ils sont des outils mathématiques très utilisés pour la simulation et l'analyse numérique de systèmes. Cependant, ces outils graphiques ne sont pas très modulaires et peuvent rapidement devenir très complexes à gérer. Ils sont parfaits pour simuler, analyser et valider les modèles numériques à de petites et moyennes échelles. C'est-à-dire que la physique des modèles peut facilement être validée par ces outils, mais que les interactions globales de plusieurs modèles se simulent et se valident difficilement avec ces outils. Dans le contexte de KARMA, leur utilisation est indispensable pour développer rapidement et valider les modèles pièce par pièce. Par la suite, une bibliothèque de liens dynamiques (*Dynamic Link Library ou DLL*) est générée à l'aide de Matlab RTW™(*Real-Time Workshop*) et le TLC™(*Target Language Compiler*). Cette bibliothèque de liens dynamiques peut alors être intégrée facilement dans l'architecture de simulation et ses interactions mieux contrôlées dans le contexte d'un engagement d'armes.

3.1.6 LaSRS++

LaSRS++, qui signifie *Langley Standard Real-time Simulation in C++*, est un effort de la *NASA Langley Research Center* ou LaRC pour produire un cadre d'applications logiciel réutilisable pour toutes les simulations d'avion en temps-réel. LaSRS++ présente certaines similarités au FDK (section 3.1.2) puisqu'il définit des services réutilisables pour la construction de différentes simulations. La figure 3.4 montre les différents modules et services disponibles.

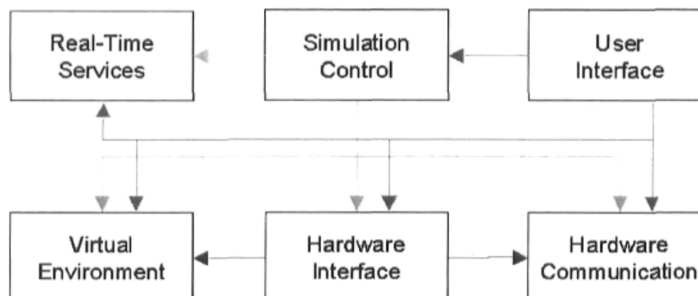

FIG. 3.4 – Modules de LaSRS++ [Mad99]

De plus, LaSRS++ est un cadre d'applications qui se spécialise dans la simulation d'avions et de véhicules. Une grande partie des articles publiée par l'AIAA (*American*

Institute of Aeronautics and Astronautics) [PoAP] explique les différentes techniques logicielles utilisées pour développer leur architecture et leurs simulations. À travers les différentes publications, il se dégage certaines constantes comme le fait que ce cadre d'applications supporte le multiprocesseurs et qu'il offre aussi la possibilité de diviser les modèles sur plus d'un processeur. LaSRS++ n'est cependant pas distribué au public et est réservé à l'usage de LaRC.

3.1.7 Environnements de simulation commerciaux

Ces autres environnements, comme mentionnés précédemment, limitent souvent les développeurs aux modèles intégrés à l'outil, mais possèdent souvent une interface graphique utilisateur (*Graphical User Interface* ou GUI) beaucoup plus développée et intuitive en plus d'avoir un visualisateur tridimensionnel (3D) impressionnant et certaines options d'extension supplémentaires. RDDC Valcartier possède, en plus d'un visualisateur maison nommé SIMPLAY, des licences d'exploitation pour Strive™de CAE. Ce produit est souvent utilisé pour accomplir les démonstrations publiques, mais aussi pour effectuer des simulations HLA. Pour ce faire, les modèles KARMA possèdent un adaptateur qui permet à Strive™de les exécuter. Une multitude d'autres environnements de développement commerciaux comme SIMSCRIPT II.5™ sont aussi disponibles.

3.1.8 Autres langages utilisés

Encore aujourd'hui certains outils de simulation commerciaux et publics ont comme base un langage propre à leur architecture de simulation (qui peut être caché par une interface graphique). Plusieurs de ces langages existent depuis longtemps et sont de moins en moins utilisés, car les capacités et performances des langages hauts niveau sont maintenant incontournables. Tous ces langages ont cependant le grand défaut de nécessiter un compilateur et/ou un environnement de développement intégré (*Integrated Development Environment* ou IDE) spécial qu'il devient difficile de remplacer et d'extensionner. De plus, la majorité de ces langages ne sont pas orientés objet et rendent donc difficile la gestion et la flexibilité des simulations développées. Simscript [RMMB04], GPSS [Sta01] et Simula [Lan] sont de bons exemples de ces langages spécialisés qui génèrent des simulations pour une architecture bien précise.

3.2 Autres architectures logicielles utiles

3.2.1 ACE

L'*ADAPTIVE Communication Environment* (ACE) est une architecture orientée-objet disponible sous les licences publiques qui implémente plusieurs patrons de conception pour la communication parallèle entre logiciels. ACE est conçu à partir de plusieurs adapteurs en C++ entourant les communications sur plusieurs systèmes d'exploitation. Il est donc un logiciel des couches intermédiaires indépendant du système d'exploitation. Les services offerts par ACE incluent la démultiplexion des événements, le routage des événements, la gestion des signaux, la communication interprocessus, la mémoire partagée, le routage des messages, la configuration dynamique des services distribués, l'exécution parallèle et la synchronisation [Groa].

Comme ACE redéfinit les mêmes services sur plusieurs plates-formes, il était donc l'outil le plus réutilisable possible pour s'assurer une compatibilité sur Windows, l'environnement de développement de KARMA, et sur QNX, l'environnement d'exécution de SEMAC. De plus, cette architecture implémente plusieurs patrons de conception pour la programmation parallèle et la communication interprocessus, outils essentiels au démarrage rapide du projet.

La figure 3.5 présente les différentes couches du cadre d'applications ACE et leurs interactions.

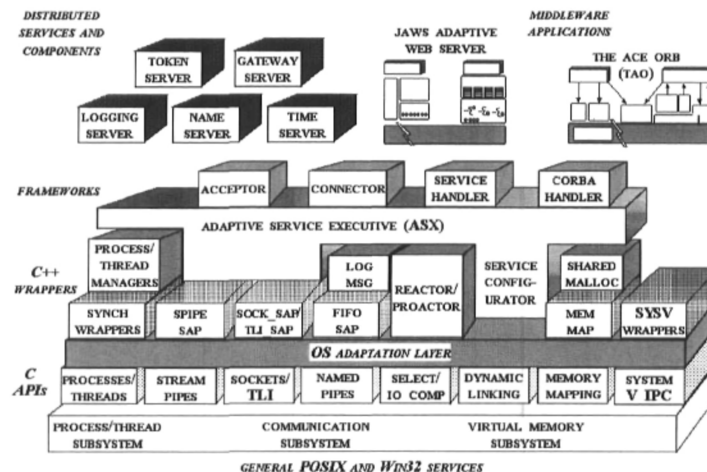

FIG. 3.5 – Les composants clés dans ACE et leurs relations hiérarchiques

3.2.2 TAO

The Ace Orb (TAO) et est en fait une implémentation du standard CORBA en utilisant ACE. Ce logiciel des couches intermédiaires permet à des clients d'invoquer des opérations sur des objets distribués sans avoir à se soucier de l'emplacement de ces objets, du langage dans lequel ils sont programmés, du système d'exploitation et du matériel sur lequel ils sont exécutés et des protocoles de communication utilisés.

La figure 3.6 présente les différentes couches du cadre d'applications TAO et leurs interactions.

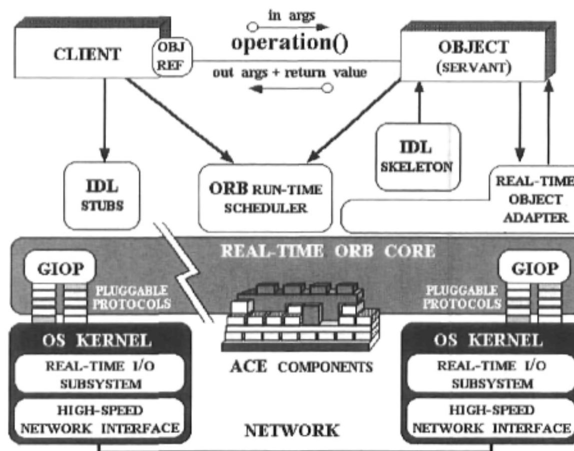

FIG. 3.6 – La configuration logicielle de TAO

3.2.3 MPI

Message Passing Interface (MPI) est une architecture logicielle pour le transfert de messages (*message-passing*) proposé comme un standard par un comité d'utilisateurs. MPI a été développé pour assurer de grandes performances dans des environnements parallèles (multiprocesseurs) et distribués (plusieurs ordinateurs reliés). MPI est disponible en version gratuite et en version payante (implémentations spécifiques par des compagnies privées). MPI propose des solutions de communication interprocessus par l'entremise de sa librairie de fonctions et de macros utilisables en C, Fortran et C++. La librairie permet des communications 1 à 1, 1 à n, n à 1 et n à n.

L'utilisation de MPI aurait pu être possible dans ce projet pour développer les moyens de communication entre les différents processus parallèles et distribués dans la

simulation. Cependant, comme l'architecture distribuée en est à sa première itération et que MPI peut être relativement compliqué à maîtriser, il était préférable d'explorer des outils de gestion de communications plus simples et faciles à implémenter.

Chapitre 4

Fondements de la simulation discrète

Une simulation est un système numérique ou analogique qui doit reproduire ou émuler le comportement d'un autre système sur une période de temps définie. La simulation permet d'analyser de manière plus précise un système avec plus ou moins de précision. La précision dépend de plusieurs facteurs, dont en autres, de la fidélité avec laquelle le système physique à analyser ou à modéliser a été reproduit. Pour être valide et représenter avec efficacité le système physique analysé, une simulation doit comporter certains éléments :

- Une représentation physique de l'état du système à simuler
- Une manière de modifier la simulation pour représenter l'évolution du système dans le temps (Section 4.2)
- Une représentation du temps (Section 4.1)

Pour répondre au premier critère, une simulation utilise des variables d'état qui représentent le système physique simulé. Ces variables d'états permettent, par la suite, l'analyse du système à des temps précis de la simulation. Pour répondre au deuxième critère, la simulation doit permettre et assurer le changement de ces variables d'état à chaque pas de temps à chaque événement. Finalement, le troisième critère est rempli par l'utilisation d'une variable à l'intérieur de la simulation qui est chargée de représenter le temps physique écoulé dans le système analysé.

4.1 Représentation du temps

La représentation du temps dans une simulation est très importante puisqu'elle permet de lier les changements à l'intérieur de la simulation elle-même. Elle permet aussi de lier la simulation au monde extérieur et au temps-réel. Dans une simulation par événements discrets (*Discrete Event Simulation* ou DES), il existe trois types de temps

- le temps physique
- le temps simulation
- le temps horloge

4.1.1 Temps physique

Le temps physique, T_P (*physical time*), représente le temps écoulé dans le système physique (le système réel). Ce temps est de peu d'utilité pour la simulation et ne sert qu'à obtenir une mesure de la durée de l'acte dans le monde réel. Par exemple, pour la simulation d'un voyage en voiture entre Québec et Montréal, le temps physique s'étend de 12h00 à 15h00 et a une durée totale de 3h00.

4.1.2 Temps simulation

Le temps simulation, T_S (*simulation time*), représente le temps écoulé dans la simulation (le système émulé). Cette abstraction utilisée par la simulation sert à modéliser le temps physique du système réel. Par exemple, les événements de la simulation (ou les pas de temps) du voyage en voiture se déroulent pendant 3h00 avec une précision de l'ordre de la seconde. La précision du temps simulation dépend de la résolution des événements étudiés dans la simulation. Ce concept est très important à l'intérieur de la simulation, car il permet de garder un certain ordre dans l'exécution des événements. Une définition plus complète de ce temps peut être trouvée dans le livre de Fujimoto [Fuj00].

4.1.3 Temps horloge

Le temps horloge, T_H (*wallclock time*), représente le temps-réel écoulé durant la simulation. Ce temps est habituellement calculé à partir de l'ordinateur sur lequel la simulation est effectuée. La précision de ce temps est reliée directement à la résolution de l'horloge électronique de l'ordinateur est question[1]. Le temps horloge n'est habituellement pas synchronisé avec le temps de simulation (ce cas particulier est nommé exécution en temps-réel, voir section 4.2.1), l'exécution de la simulation peut se dérouler soit plus rapidement ou moins rapidement que le système réel. Par exemple, la simulation du voyage en voiture pourrait avoir été exécutée en 20 minutes sur un ordinateur particulier.

4.2 L'avancement du temps

Dans une simulation, l'écoulement du temps peut se faire de trois manières différentes :

- En temps-réel
- En temps-réel proportionnel
- Aussi vite que possible

La progression du temps dans la simulation peut être directement ou indirectement couplée au temps horloge. La manière d'écouler le temps est choisie à partir du type de simulation effectuée et des contraintes qu'elle engendre. Dans le cas d'une simulation utilisée pour un environnement virtuel, il est impératif que la simulation puisse s'exécuter en temps-réel pour que l'environnement soit réaliste pour les utilisateurs. Dans d'autres cas, comme lorsque les interactions avec les usagers sont minimes ou inexistantes, la technique *aussi vite que possible* est privilégiée à cause de sa simplicité d'implémentation et la rapidité d'exécution.

Fujimoto présente l'écoulement du temps par une équation générale qui permet de présenter les deux types de temps-réel cités ci-haut [Fuj00]. L'équation 4.1, qui est appelé H2S pour temps horloge à temps simulation, est basée sur un point de référence du temps simulation (T_{Start}) et sur une différence du temps horloge ($T_H - T_{HStart}$) pour donner le temps courant de la simulation (T_S). Dans cette équation, `Scale` est une

[1]Par exemple, sur Windows, la fonction `QueryPerformaceCounter` peut être utilisée pour obtenir le nombre de cycles d'horloge écoulés depuis le démarrage de l'ordinateur.

constante qui permet d'accélérer ou de ralentir le temps horloge.

$$T_S = H2S(T_H) = T_{Start} + Scale * (T_H - T_{HStart}) \tag{4.1}$$

Une explication plus détaillée et une simplification de l'équation générale sera donnée pour les cas d'écoulement du temps possible dans une simulation temps-réel et temps-réel proportionnel.

4.2.1 Exécution en temps-réel

Une exécution en temps-réel signifie que le temps simulation s'écoule à la même vitesse que le temps horloge. L'équation 4.2 représente cette relation.

$$T_S = T_H \tag{4.2}$$

Le temps dans la simulation doit à tout moment être égal au temps écoulé sur l'ordinateur exécutant la simulation. Pour être plus pratique, la simulation doit avoir terminé les calculs reliés à un événement ou un pallier de temps avant que le temps horloge ne soit rendu au temps d'exécution du prochain événement ou pallier de temps prévu, comme le présente la figure 4.1.

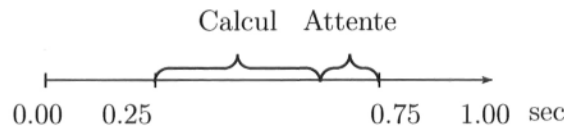

FIG. 4.1 – Exécution en temps-réel

Cette technique est couramment utilisée pour des systèmes embarqués qui doivent interagir avec du matériel qui possède des contraintes de temps très précises et dans les cas où une interaction externe avec l'humain est nécessaire. Les systèmes embarqués avec de fortes contraintes temporelles sont souvent catégorisés comme étant temps-réel dur (*hard real-time*) puisqu'à aucun moment une échéance de temps ne peut être dépassée sans conséquence grave pour le système ou les usagers. De tels systèmes sont par exemple, les systèmes de contrôle dans un avion. Les systèmes temps-réel mou (*soft real-time*) sont des systèmes pour lesquels il est sans conséquences graves, mais avec dégradation des performances, de dépasser une contrainte de temps. Le terme Mou signifie qu'il y a une flexibilité dans les contraintes de temps-réel. Un environnement virtuel est un bon exemple de ces systèmes pour lesquels l'usager n'aura pas d'autres

conséquences que d'avoir une expérience moins immersive et moins réaliste. Dans le projet présent, la relation entre KARMA et SEMAC peut être catégorisée d'applications temps-réel mou.

4.2.2 Exécution en temps-réel proportionnel

Une exécution en temps-réel proportionnel signifie que le temps simulation s'écoule à une vitesse proportionnelle à celle du temps horloge. L'équation 4.3 représente cette relation.

$$T_S = Scale * T_H \tag{4.3}$$

Cette technique est peu utilisée puisqu'elle implique d'avoir un excellent contrôle sur la relation expliquée entre le temps horloge et le temps simulation. Cette technique s'apparente à la manière dont un DVD peut être lu en vitesse accélérée ou en vitesse ralentie. Évidemment, une constante négative est impossible puisque pour être valide, le temps doit s'écouler de manière positive.

4.2.3 Exécution aussi vite que possible

Pour des simulations ayant pour but une analyse numérique, une visualisation post-simulation et aucune interaction externe, la progression du temps la plus courante est appelée *aussi vite que possible* (*as fast as possible*). Cette technique est la plus simple à implémenter puisqu'elle ne nécessite presque aucun lien avec le temps horloge et la vitesse de la simulation est alors limitée par l'ordinateur sur laquelle elle est exécutée. Il est important de comprendre que la vitesse de cette simulation est aussi déterminée par la complexité des calculs et des modèles à exécuter et que, finalement, la simulation peut être très lente par rapport au temps physique du système réel.

Il est aussi à noter qu'une simulation se déroulant moins rapidement que le temps physique, ne peut pas être exécutée en temps-réel puisqu'elle est simplement trop chargée en calculs. Dans ces circonstances, il ne reste qu'une solution : ajouter de la puissance de calcul à l'ordinateur exécutant la simulation. Dans le cas contraire, une simulation s'exécutant plus rapidement que le temps Physique peut être ralentie pour s'adapter au temps horloge et ainsi s'exécuter en temps-réel. Dans le projet présent, la parallélisation de l'environnement de simulation de KARMA, la technique *aussi vite que possible* a été

choisie et implémentée dans la première phase. En résumé, il est beaucoup plus facile de ralentir l'exécution d'une simulation que de l'accélérer.

4.3 L'exécution et l'évolution dans le temps

L'évolution d'une simulation à événements discrets dans le temps peut se faire de deux manières :

- Par pas de temps
- Par événements

Ces deux techniques les plus couramment utilisées se distinguent par la manière dont elles permettent l'avancement du temps dans la simulation. Les deux techniques possibles seront couvertes puisqu'au cours du projet de parallélisation, KARMA est passé de la première technique (moins performante) à la deuxième (plus performante). Le passage fut un grand pas dans l'évolution de l'architecture de KARMA puisque l'exécution par événements est la seule technique à pouvoir être utilisée dans un environnement de simulation parallèle et distribuée.

4.3.1 Exécution par pas de temps

L'exécution d'une simulation par pas de temps (*time-stepped execution*) signifie que l'avancement du temps se fait exclusivement par incréments de temps fixes. Les changements des variables d'état des modèles ne peuvent s'effectuer qu'à ces moments prédéterminés appelés pas de temps. La figure 4.2 présente ce concept.

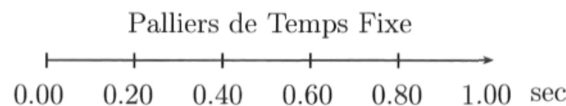

FIG. 4.2 – Exécution par pas de temps

Cette technique est généralement utilisée pour une simulation dans laquelle toutes les variables d'état du système complet sont recalculées à chaque pas de temps. Habituellement, l'incrément doit correspondre à la plus petite résolution numérique des modèles que l'on désire simuler. Dans la première version de KARMA, la technique

utilisée était celle des pas de temps. Chaque modèle possédait une période de temps à laquelle il devait s'exécuter. Cette période était déterminée par les modeleurs. Par exemple, plusieurs modèles dans KARMA ont une période allant de 0.01 à 0.05 s, certains, nécessitant une exécution moins fréquente, ont une période de l'ordre de 1 s.

Le pseudo-code de l'algorithme basé sur le principe de la simulation par pas de temps est présenté à la figure 4.3.

```
Tant que (la simulation est en cours)
    Attendre que le dernier pas de temps soit achevé
    Calculer l'état du système à la fin du pas de temps
    Avancer la simulation au prochain pas de temps
```

FIG. 4.3 – Algorithme d'une simulation à pas de temps précis

Dans la première version de KARMA, cet algorithme était utilisé avec certains ajustements propres aux besoins spécifiques de l'architecture et des interactions nécessaires aux engagements d'armes.

Plus la simulation devient grande et plus cette technique est coûteuse en temps de calcul, car normalement, tous les modèles n'ont pas à être recalculés à tous les pas de temps, mais seulement quand un élément d'intérêt se produit comme le changement d'une variable d'entrée. Cet élément d'intérêt est ce qu'on appelle un événement. Il est aussi à noter que l'algorithme oblige le passage à tous les incréments de pas de temps, même si aucun calcul n'est nécessaire, ce qui ralentit énormément la vitesse globale de la simulation.

L'algorithme par pas de temps peut aussi être utilisé dans une situation nécessitant une exécution en temps-réel. Pour ce faire, il suffit d'attendre (ralentir la boucle) que le temps horloge équivalent se soit écoulé avant de passer au pallier suivant. Cette modification implique que tous les calculs de tous les pas de temps soient terminés avant le temps horloge équivalent sinon la simulation subira un retard (*lag*).

4.3.2 Exécution par événements

L'exécution d'une simulation par événements, signifie qu'il n'est pas nécessaire de calculer de nouvelles valeurs pour les variables d'état du système à tous les pas de temps, mais seulement lorsque quelque chose d'intéressant survient dans la simulation. L'élément d'intérêt est nommé un événement. L'événement est une abstraction utilisée dans la simulation pour représenter des actions instantanées dans le système physique.

La figure présente une ligne du temps qui montre comment peut varier l'avancement du temps simulation (T_S) à l'intérieur de la simulation, c'est-à-dire par bonds non égaux. La longueur des bonds est influencée par les entrées d'un modèle et par le temps que prennent les différents modèles dans la chaîne le précédant pour calculer leurs nouvelles variables d'état. Ces calculs sont uniques et différents pour chaque modèle. Le changement de valeur d'une variable d'état représente en soi un événement puisque d'autres modèles peuvent baser leur entrée sur cette donnée. L'événement annonce donc aux autres modèles qu'ils doivent eux aussi s'exécuter pour refléter les derniers changements.

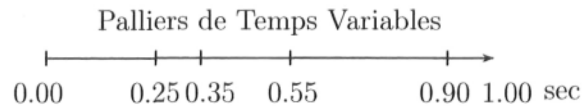

FIG. 4.4 – Exécution par événements

Le pseudo-code de l'algorithme basé sur le principe de la simulation par événements est présenté à la figure .

```
Tant que (la simulation est en cours)
    Retirer l'événement avec la plus petite estampille temporelle
    Ajuster le temps de la simulation au temps de l'événement
    Exécuter l'événement retiré
```

FIG. 4.5 – Algorithme d'une simulation à pas de temps variable

Ce type d'algorithme repose sur une structure de donnée principale qu'on appelle généralement la liste des événements futurs (*Futur Event List* ou FEL). Cette structure doit trier les événements qu'elle contient en fonction de leur estampille temporelle, de la plus petite à la plus grande.

Dans la deuxième version de l'algorithme d'exécution de KARMA, les événements ont été intégrés à l'intérieur des modèles eux-mêmes et la liste des événements a été rajoutée pour permettre aux modèles de se rappeler à un temps futur dans la liste en question. Le temps de rappel des modèles, leur période, n'a cependant pas changé par rapport à l'algorithme pas de temps.

En résumé, une simulation discrète doit contenir trois structures de données importantes :

- Des *variables d'état* qui décrivent les changements dans le système physique
- Une *liste des événements* qui doivent s'exécuter dans le futur simulé

- Une *horloge globale* qui permet de situer l'instant courant de la simulation sur l'axe du temps

La simulation discrète doit aussi d'assurer de respecter certaines règles pour reproduire correctement les relations causales dans le système physique. Premièrement, la simulation doit exécuter uniquement des événements se situant dans le futur de la simulation. C'est-à-dire que le temps d'exécution des événements doit uniquement croître (ou être égal) tout au long de la simulation. Aucun événement ayant un temps d'exécution plus petit que le temps courant ne doit être exécuté. Deuxièmement, la simulation doit toujours exécuter l'événement avec le plus petit temps d'exécution disponible. Ces deux propriétés vont assurer que la simulation exécute les événements dans un ordre temporel croissant et que le temps d'exécution ne décroisse jamais.

4.4 La simulation parallèle et distribuée

La littérature scientifique décrit la simulation parallèle et distribuée comme étant un regroupement de simulations séquentielles, chacune modélisant une partie différente du système physique et s'exécutant potentiellement sur plusieurs processeurs [Fuj00]. Chaque simulation séquentielle différente est appelée un processus logique (*logical process*). Un système physique trop complexe peut alors être décomposé en plusieurs petits systèmes physiques[2] communiquant entre eux par une ou diverses techniques de communication interprocessus, discutées à la section 8.1, et étant modélisés par des processus logiques. Au cours d'une simulation parallèle ou distribuée, les processus logiques produisent des événements qui engendrent d'autres événements qu'ils soient destinés pour le même processus logique ou pour tout autre. La communication entre les différents processus logiques est référée comme étant le passage de messages.

La simulation parallèle implique l'usage d'un ordinateur possédant plusieurs processeurs et la simulation distribuée implique l'usage de plusieurs ordinateurs généralement reliés par un réseau de communication local (LAN ou WAN) avec des protocoles internet comme IP/UDP. Les deux systèmes se comportent relativement de la même manière, mais la distribution engendre des problèmes plus complexes de synchronisation entre les différentes composantes à cause des communications réseau plus lentes que les accès mémoires de différents processeurs et des références de temps différentes d'un ordinateur à l'autre. Ces deux systèmes sont cependant comparables du point de vue de l'exécution et de la gestion de la simulation. Deux algorithmes majeurs (et

[2]Tous les types de modèles ne sont pas forcément parallélisable, les plus faciles sont les modèles fonctionnels. L'outil de simulation KARMA est uniquement composé de modèles fonctionnels.

quelques variantes) sont disponibles pour l'exécution de telles simulations : l'exécution conservatrice et optimiste. Tous ces principes et concepts précis de la simulation discrète parallèle et distribuée seront discutés plus en détail tout au long de la description du projet.

Chapitre 5

Architecture de simulation de KARMA

Avant de présenter les travaux accomplis, il est important d'élaborer plus en détail sur l'architecture interne de KARMA : les liens entre les modèles simulés, les techniques utilisées pour échanger les variables d'état entre les modèles, la manière de gérer l'exécution séquentielle des modèles et la gestion du temps de simulation.

5.1 Liens entre les modèles dans KARMA

KARMA a été développé dans le but de permettre une modularité accrue des modèles de simulation. Pour cette raison, la majorité des modèles se divisent en deux catégories ; les BaseEntity et les Parts. Les BaseEntity sont des modèles pouvant exister seuls à l'intérieur de la simulation. Les Parts ne peuvent pas exister seules et doivent composer un BaseEntity pour pouvoir être exécutés. La figure 5.1 montre bien la distinction entre les deux types d'entité possibles pour une simulation d'engagement d'armes dans KARMA.

Le design de l'architecture interne du simulateur (figure 5.2) permet de voir que même si les modèles sont instanciés dans différentes catégories (BaseEntity et Parts), ils héritent tous des mêmes services, défini dans la classe de base Root, qui leur permet de s'exécuter dans un même environnement appelé le théâtre.

La fonction virtuelle `run` est le point d'entrée dans le modèle pour commander

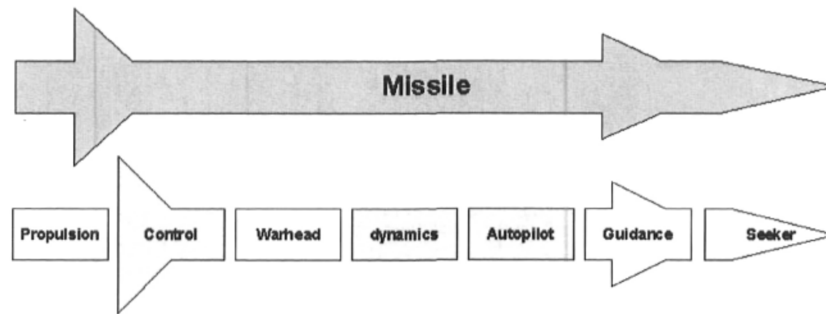

FIG. 5.1 – Composition d'un modèle de missile (Baseentity) à l'aide de Parts [BL03]

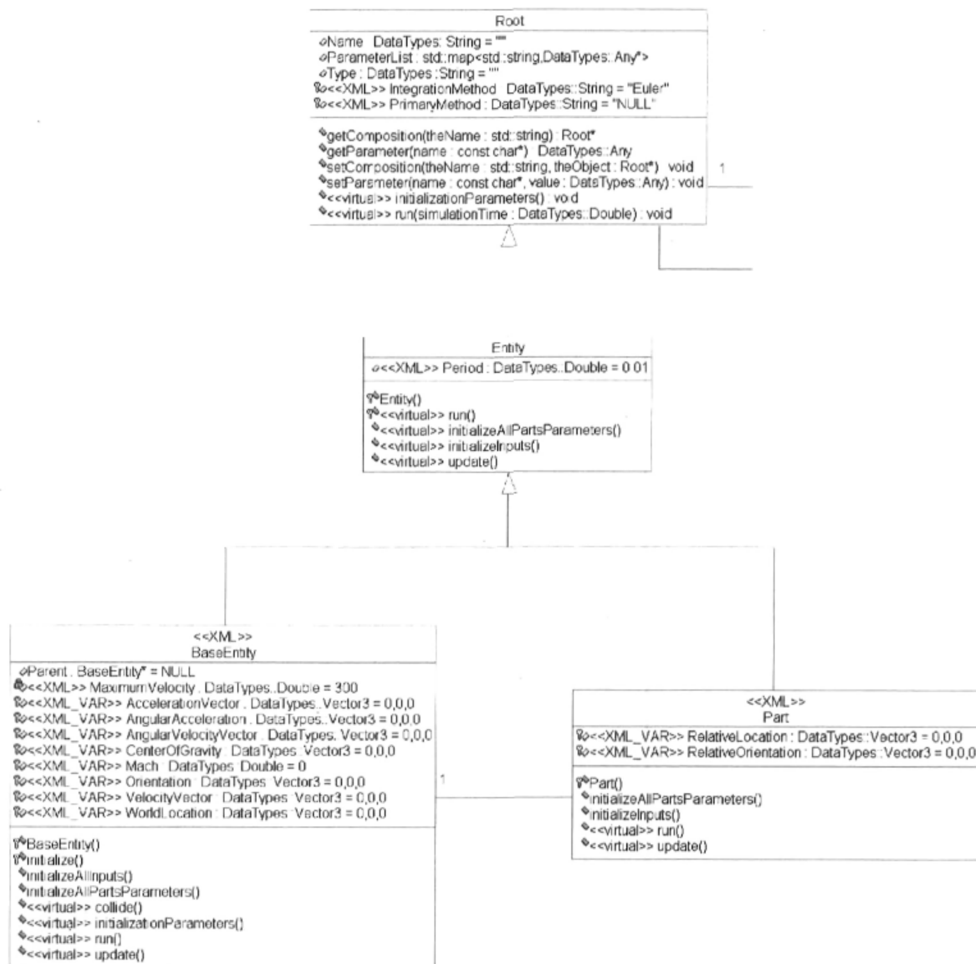

FIG. 5.2 – Diagramme de classe : `Root`, `Entity`, `Baseentity` et `Parts`

son exécution et elle reçoit en paramètre le temps de simulation au moment de son exécution. Ce temps doit être précis, car l'exécution de certains modèles doit se faire à une fréquence particulière pour que les résultats obtenus à la fin de la simulation soient valides. Dans l'outil de simulation KARMA, certains modèles ont une période de 0.05 s et donc le temps de la simulation doit, au minimum, avoir une résolution de 0.01 s.

5.2 Échange des variables d'états

Dans tous les services offerts par la classe Root présentée à la figure 5.2, deux méritent une plus grande attention, car ils contribuent à permettre l'échange des variables d'états entre tous les modèles instanciés dans la simulation. Les deux fonctions :

- `DataTypes::Any getParameter (const char* name)`
- `void setParameter (DataTypes::Any)`

accèdent à une structure de donnée de type dictionnaire (*std::map*) située dans la classe Root. Cette structure est unique à chacun des modèles. L'accès dans le map se fait par le nom du paramètre. Le type de variables d'état conservées par la structure de donnée est un paramètre physique du modèle simulé ; par exemple, WorldLocation, Velocityvector, Orientation and AccelerationVector. Cette technique facilite l'échange des données à l'intérieur du théâtre dans la simulation et évite l'encombrement d'un système de messages, mais est contraire au principe d'encapsulation et rend plus difficile, comme il sera présenté plus tard, la séparation des entités simulées dans un contexte plus flexible de simulation distribuée.

5.3 Exécution séquentielle des modèles

Dans un environnement de simulation séquentiel, deux techniques sont couramment utilisées pour gérer l'exécution des modèles. La première est le l'exécution par pas de temps (*time-stepped execution*) et la deuxième est l'exécution par événements (*event-driven execution*) [Fuj00]. Pendant les travaux effectués, KARMA a évolué de la première technique à la deuxième. Dans l'architecture actuelle, l'exécution est gérée par un algorithme qui boucle sur une structure de données devant ordonner les différents modèles par leur prochain temps d'exécution et par leur priorité relative. L'algorithme n'a alors qu'à choisir le premier élément de la structure, à l'exécuter, à avancer le temps de la simulation en conséquence et à recommencer jusqu'à ce que le temps de fin de

simulation soit atteint. L'exécution du modèle se fait comme décrit précédemment, par la fonction Run avec comme paramètre le temps de simulation courant. Bien entendu, l'algorithme dans KARMA a été adapté aux besoins spécifiques de l'application et contient aussi une panoplie de tests et de conditions sur les modèles et leurs variables pour bien engendrer les diverses interactions possibles et nécessaires à la réalisation d'une simulation d'engagement d'armes.

Chapitre 6

Processus de parallélisation

Cette section a pour but de présenter globalement l'état actuel de l'environnement de simulation KARMA et de schématiser comment l'environnement initial doit être modifié pour arriver à valider l'objectif principal du présent projet. Cette section a aussi pour but de situer la place des prochains chapitres qui vont présenter en détail les travaux de conception.

6.1 Simulation séquentielle

Dans une simulation séquentielle, à chaque pas de temps, les modèles s'exécutent séquentiellement dans un ordre préétabli par l'application principale. La figure 6.1 schématise ce principe. L'environnement de simulation KARMA applique ce principe.

Lorsque la simulation démarre, l'application principale entre dans une boucle qui se termine lorsque le temps de simulation final est atteint. À l'intérieur de cette boucle, l'application principale détermine à chaque pas de temps quels modèles doivent être exécutés. Lorsque l'application exécute un modèle, elle ne fait rien d'autre et attend que le modèle termine ses calculs et lui rende le contrôle.

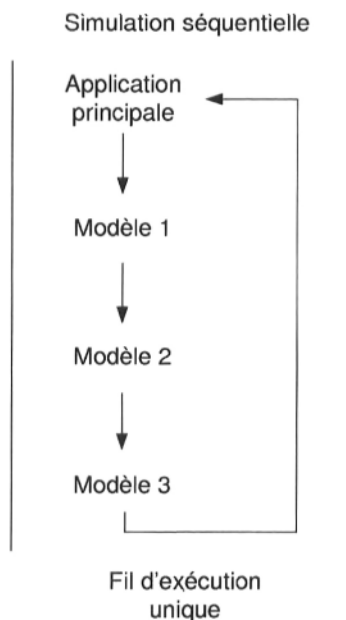

FIG. 6.1 – Schéma présentant l'exécution actuelle d'une simulation séquentielle dans l'environnement KARMA

6.2 Simulation parallèle

Dans une simulation parallèle, à chaque pas de temps, les modèles, qui n'ont pas de liens restrictifs entre leurs sorties et les entrées d'autres modèles, s'exécutent parallèlement sans regard pour l'ordre puisque ces modèles sont considérés comme étant concurrents. La figure 6.2 schématise le principe que l'environnement de simulation devrait appliquer pour augmenter sa puissance de calcul. Le gain engendré par ce principe ne sera visible que sur un ordinateur possédant plus d'un processeur, pour permettre l'exécution parallèle des différents modèles.

La parallélisation de l'environnement séquentiel initial nécessite cependant le développement de nouveaux modules. Le premier module nécessaire concerne la création des processus logiques indépendants de l'application principale qui pourront exécuter les différents modèles qui leur seront attribués. Ce module est le sujet du chapitre 7. Le second module concerne le problème de la communication entre l'application principale et les différents processus logiques exécutant les modèles. Puisque l'application est divisée en plusieurs fils d'exécution, une communication doit être établie pour permettre une synchronisation des différents fils. Le module de la communication est le sujet du chapitre 8. Le dernier module concerne l'implémentation d'un nouvel algorithme dans l'application principale pour exploiter les différents fils d'exécution pour les modèles

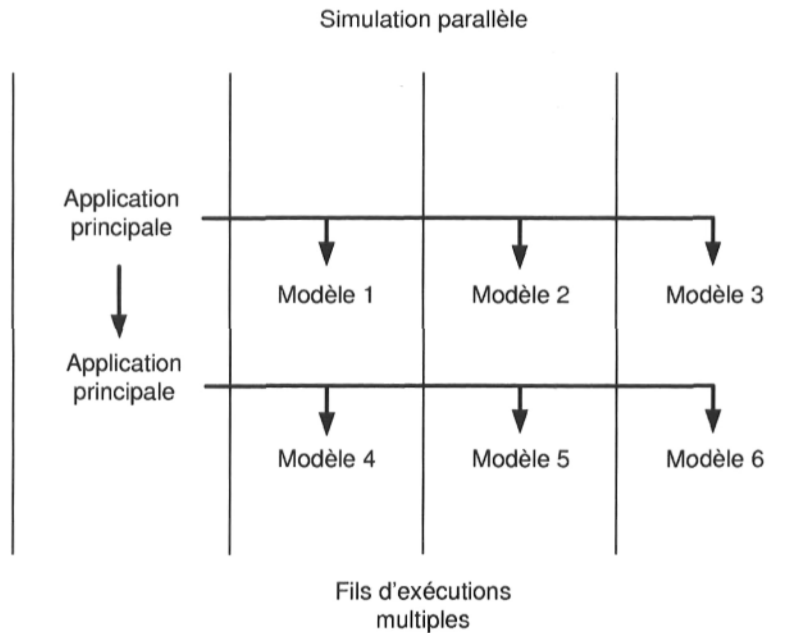

FIG. 6.2 – Schéma présentant l'exécution souhaitée d'une simulation parallèle dans l'environnement KARMA

et la gestion du temps de simulation. Le nouvel algorithme doit aussi être en mesure de déterminer les modèles pouvant être exécutés en parallèle. Le module de l'application principale est le sujet du chapitre 9. Ensuite, le chapitre 10 traite de l'application des concepts développés pour l'environnement parallèle à l'architecture de simulation KARMA. Les résultats des travaux seront finalement présentés au chapitre 11.

Chapitre 7

Les processus logiques

7.1 Architectures

Dans le monde de la simulation séquentielle, tout est effectué à l'intérieur du même programme (pouvant contenir une multitude de classes) ; c'est-à-dire qu'un seul processus est responsable de coordonner toute la simulation à tour de rôle. Le plus souvent, les simulations séquentielles sont exécutées sur une architecture matérielle de type Von Neumann. Ce type d'architecture comporte un seul processeur et une seule mémoire centrale, comme le montre la figure 7.1 (a), et sont aussi appelés SISD (*Single Instruction Single Data*). Dans cette architecture un processeur ne peut exécuter qu'un seul flux de données. De nos jours, ces architectures peuvent simuler le parallélisme grâce aux systèmes d'exploitation qui exécutent à tour de rôle plusieurs flux de données d'une manière transparente pour l'utilisateur.

Pour ce qui est des simulations parallèles, les architectures présentées aux figures 7.1 (b) et (c) se nomme respectivement SIMD (*Single Instruction Multiple Data*) et MIMD (*Multiple Instruction Multiple Data*). L'architecture MIMD est la plus couramment utilisée avec des environnements appelés multiprocesseurs où plusieurs programmes exploitent plusieurs processeurs pour faire des calculs en parallèle. Pour s'adapter à cette architecture, un programme peut se diviser en plusieurs tâches, des fils d'exécution, pour effectuer ses opérations en parallèle. L'architecture SIMD est bien adaptée pour le calcul vectoriel ou parallèle où un grand nombre de données sont divisées en plus petites parties qui sont ensuite individuellement traitées par un processus identique. Un seul flux de donnée dirige les opérations de plusieurs unités de calcul pour exécuter les mêmes manipulations simultanément sur un grand nombre de données. Cette architecture trouve ses usages dans certains domaines particuliers comme la simulation

discrète, mais est généralement peu répandue.

La simulation distribuée peut quant à elle être représentée par l'addition de plusieurs architectures SISD ou SIMD, car le concept de base est d'utiliser plus d'un ordinateur pour effectuer les calculs.

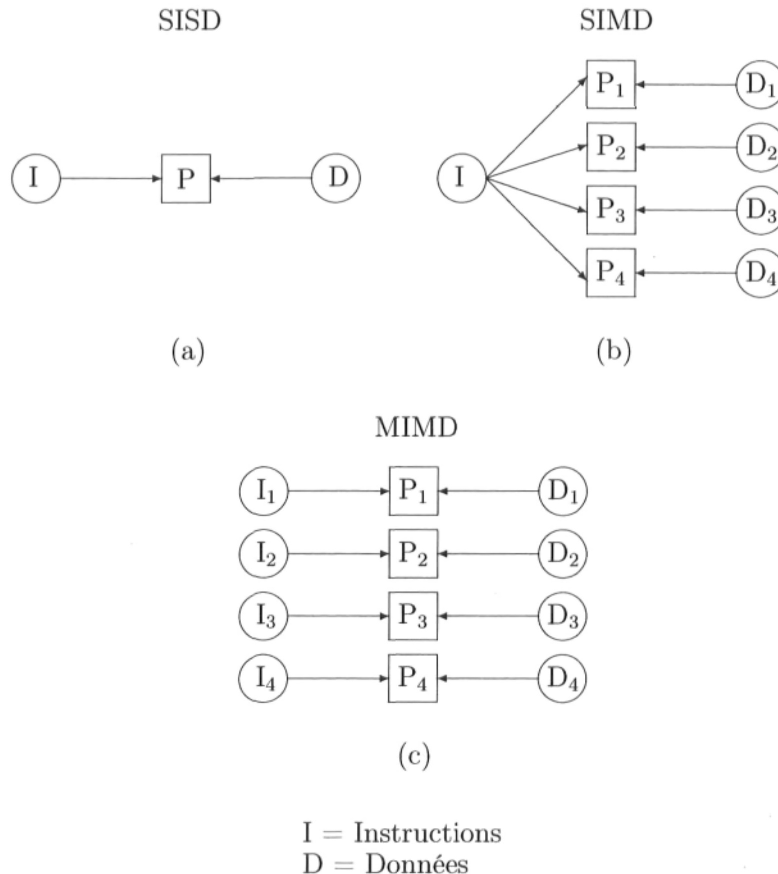

FIG. 7.1 – Architectures des ordinateurs [Fis95]

7.2 Définition

Paul A. Fishwick décrit la simulation numérique comme étant la discipline de designer un modèle conceptuel d'un actuel ou théorique système physique, d'exécuter de ce modèle sur un ordinateur et d'analyser les données de sortie de l'exécution [Fis95].

Dans son livre, M. Fujimoto décrit la simulation parallèle et distribuée comme étant

un regroupement de simulations séquentielles, chacune modélisant une différente partie du système physique et s'exécutant ou non sur plusieurs processeurs [Fuj00].

La simulation discrète par événements (*discrete event simulation*) se définit comme la représentation chronologique par une séquence d'événements de l'opération d'un système. Chaque événement, qui se produit à un instant précis dans le temps, dénote un changement d'état dans le système.

Le présent projet reprend donc en partie les trois définitions précédentes puisque nous évoluons dans un environnement de simulation discrète événementielle parallèle et distribuée. Pour vulgariser, on peut décrire la simulation parallèle comme étant une multitude de simulations séquentielles ayant des liens entre elles. Chaque simulation séquentielle est appelée un processus logique (*logical process*). Dans notre projet, un processus logique représente habituellement une entité distincte (ou un composant) à l'intérieur de la simulation globale. Chaque processus logique obtient alors des résultats lors de son exécution, les calculs des variables d'état, et doit les communiquer aux autres processus logiques qui pourraient les utiliser comme entrées pour générer leurs propres variables d'état.

7.3 Fils

Pour prendre en charge tous les différents modèles de la simulation parallèle et distribuée, le simulateur a besoin de plusieurs processus ou de plusieurs fils. La différence majeure entre les deux est que les processus possèdent un espace mémoire distinct tandis que les fils peuvent partager des parties de leur espace mémoire. L'utilisation de fils pour implémenter le parallélisme, par l'entremise des processus logiques, est un choix qui va de soi à cause de l'avantage indéniable qu'ils procurent grâce à leur espace mémoire commun. Le choix d'utiliser des fils pose alors une seconde question, soit le choix d'implémenter le concept de la réserve de fils ou le concept des fils sur demande.

Le concept des fils sur demande signifie que les fils sont créés à l'instant où l'utilisateur en a besoin et qu'ils sont détruits dès la fin de leur travail. Le concept de la réserve de fils signifie que les fils sont tous créés à l'avance dans un groupe et qu'ils restent inactifs tant qu'on ne les utilise pas. Quand le fil a fini son travail, il retourne dans le groupe jusqu'à ce que l'utilisateur en ait besoin à nouveau. Sans analyser en détail tous les avantages et désavantages[1] de ces approches, il faut noter le plus intéressant en faveur du concept de la réserve de fils ; soit qu'il permet de connaître à l'avance (c.-

[1]Voir [CW98] pour plus de détails sur les deux concepts

à-d. au démarrage) les ressources engendrées par l'application. Il n'y a pas de création dynamique de fils et donc aucune surprise sur un éventuel manque de ressources.

Dans le projet de parallélisation de KARMA, les besoins sont de pouvoir générer un nombre fixe de fils au démarrage de la simulation, le nombre de fils étant fixé par le nombre d'entités présentes dans le scénario de l'engagement. Les fils ont pour but de prendre en charge un modèle BaseEntity et ses modèles Parts, de permettre leur exécution tout au long de la simulation et de se détruire à la fin de la simulation.

7.4 Design conceptuel

Selon les besoins déterminés, le concept de la réserve de fils a été choisi pour implémenter les processus logiques de la simulation parallèle. Cependant comme le cadre d'applications ACE avait été choisi pour coder le logiciel, l'implémentation finale s'est retrouvée à mi-chemin entre les deux concepts de base discutés ci-haut.

Pour les traitements multifils, ACE propose des fonctions de base pour créer des fils comme `spawn` ou `spawn_n`, mais aussi des objets de plus haut niveau comme des tâches (*tasks*) ou des objets actifs (*active objects*) [SSB01]. Les objets actifs sont en fait un patron de conception de programmation génie logiciel qui permet des appels de fonctions asynchrones. Les tâches sont représentées par la classe `ACE_Task` qui est une classe de haut niveau en orienté objet pour gérer un ou plusieurs fils selon une interface prédéfinie. Cette classe évite au programmeur d'avoir à programmer uniquement les fils en langage C et en appels de fonctions successifs. De plus, cette classe contient aussi un mécanisme qui facilite la communication avec d'autres tâches.

7.4.1 ACE_Task

La figure 7.2 illustre la structure de la classe `ACE_Task`.

Pour utiliser le cadre d'applications, il faut tout d'abord dériver une classe du modèle `ACE_Task` et spécifier la synchronisation désirée, soit `ACE_NULL_SYNCH` (aucun traitement multifil) ou `ACE_MT_SYNCH` (traitement multifil). Le choix de la synchronisation affecte la queue des messages (ACE_Message_Queue) en y ajoutant des mutex pour restreindre l'accès à un seul fil à la fois. Tout ce travail est fait par le cadre d'applications et est transparent pour l'utilisateur. Pour pouvoir utiliser le cadre d'applications cor-

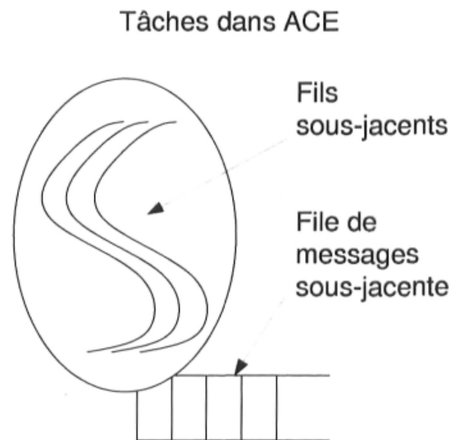

FIG. 7.2 – Structure de la classe `ACE_Task` contenant un ou plusieurs fils et le lien de communication `ACE_Message_Queue` [Syy]

rectement, l'utilisateur doit redéfinir certaines fonctions virtuelles pour l'initialisation, l'exécution et la destruction selon les besoins. Toutes les fonctions disponibles dans le cadre d'applications et leurs modes d'utilisations sont décrits en détail dans [SH02], [SH03] et [HJS03].

L'utilisation principale de cette classe est donc d'avoir un ou plusieurs fils effectuant la même fonction ou le même algorithme comme pour le simulateur parallèle. Cette classe `ACE_Task` a été utilisée pour implémenter le protocole DEVS présenté à la section 7.1.3. Chaque objet **DEVSThreadPool**(notre classe dérivée de `ACE_Task`) instancie le nombre de fils nécessaire pour simuler les modèles qui lui sont attibués. Pour l'instant, un seul fil est nécessaire et contient le modèle Baseentity en plus de toutes ses Parts. La fonction principale (`svc`) effectue une boucle centrée sur la file de communication, l'objet `ACE_Message_Queue`, une file d'attente et utilise une fonction interne (`getq`) pour récupérer les messages sur la file. L'algorithme traite alors chaque message selon le type et les informations contenues.

7.4.2 ACE_TSS

Comme la classe `ACE_Task` peut activer plusieurs fils à l'intérieur de son espace mémoire, tous les fils peuvent aussi avoir accès aux mêmes variables et aux mêmes espaces mémoires. Cela signifie que si plusieurs fils accèdent à la même variable de la classe en cours d'exécution et qu'ils modifient la valeur de cette variable, ils accèdent tous au même espace mémoire et écrasent à tour de rôle la valeur de la variable. À la fin de l'exécution de tous les fils, la valeur finale de la variable sera celle que le dernier

fil aura écrite.

Cette caractéristique peut être indésirable si le besoin est que chaque fil possède certaines variables uniques. L'environnement parallèle de KARMA requiert cette propriété puisque chaque fil doit pouvoir simuler un modèle différent. Pour résoudre ce problème, il faut utiliser le patron (*pattern*) comportemental *Thread-Specific Storage* (TSS) [SHP97]. Ce patron permet à plusieurs fils d'utiliser un seul point d'accès global pour récupérer des données spécifiques sans rajouter un temps d'accès de verrouillage pour chaque requête. La figure 7.3 montre la structure du patron comportemental TSS.

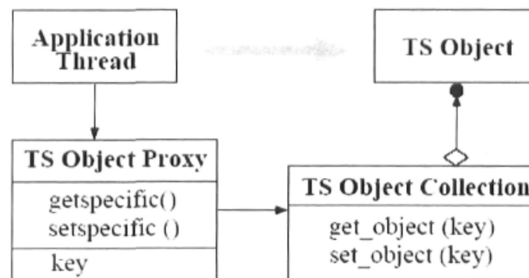

FIG. 7.3 – Structure des participants dans le patron *Thread-Specific Storage* [SHP97]

Ce patron de programmation est intégré dans le cadre d'applications ACE et son utilisation se fait par l'entremise du modèle `ACE_TSS`$< Type >$. Pour créer des variables uniques à chaque fil dans une tâche, il suffit de créer une classe contenant les variables désirées (on peut également y ajouter des fonctions) et d'instancier un objet à partir de ce modèle. Voici un court exemple :

```
//classe contenant les variables uniques
class VariablesUniques;

//Objet unique à chaque thread
ACE_TSS<VariablesUniques> ObjetUnique;
```

Par la suite, il suffit de récupérér les variables et les fonctions comme pour toute autre classe, c'est-à-dire avec l'opérateur d'accès (.). La figure 7.4 présente un design UML™d'une classe `ACE_Task` et l'utilisation du patron `ACE_TSS`. Cette figure montre qu'une structure a été créée pour contenir les pointeurs des modèles associés à ce fil en particulier. Le dictionnaire est construit à partir du pointeur du modèle, de type BaseEntity, qui est passé à la classe `ThreadPool2` par l'entremise d'un message. Cette structure est créée à l'initialisation pour accéder plus rapidement aux pointeurs des modèles (BaseEntity et ses Parts) lors de l'exécution de la simulation.

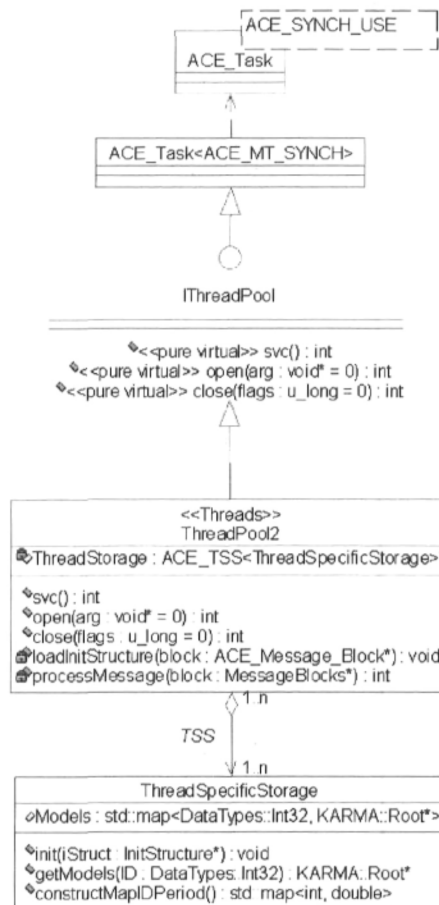

FIG. 7.4 – Diagramme de classe UML™ d'une partie du boîtier `LogicalProcess`

7.4.3 Protocole DEVS

La figure 7.5 décrit la structure d'un DEVS séquentiel de base.

$$DEVS = \{X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta\}$$

où

X est l'ensemble des entrées
Y est l'ensemble des sorties
S est l'ensemble des états séquentiels
$\delta_{ext} : Q \times X \rightarrow S$ est la fonction de transition externe
$\delta_{int} : S \rightarrow S$ est la fonction de transition interne
$\lambda : S \rightarrow Y$ est la fonction qui génère les sorties
$ta : S \rightarrow \Re_0^+ \cup \infty$ est la fonction d'avancement du temps
$Q = \{(s, e)|s \in S, 0 = e = ta(s)\}$ est l'ensemble total des états

FIG. 7.5 – Définition d'un DEVS classique

Ce formalisme mathématique introduit par Bernard P. Zeigler [ZPK00], représente l'objet de base de son protocole. Ce formalisme couplé aux algorithmes d'exécution permet de déterminer avec exactitude les changements qui vont survenir dans le modèle s'il est compatible à la théorie DEVS, c'est-à-dire qu'il possède les caractéristiques énumérées dans la figure 7.5. Ce formalisme n'est cependant applicable que pour les modèles séquentiels, c.-à-d. ceux pour lesquels deux événements ne peuvent survenir au même instant.

Sans entrer dans tous les détails, cette définition d'un DEVS classique implique que le modèle doit posséder 4 fonctions principales :

- La fonction externe δ_{ext} qui sert à exécuter l'ensemble des entrées X
- La fonction interne δ_{int} qui sert à exécuter des événements internes
- La fonction $\lambda(s)$ qui génère des sorties à partir de l'état s du modèle et toujours après l'exécution de δ_{int}
- La fonction $ta(s)$ qui sert à calculer le prochain temps d'exécution d'un événement interne à partir de l'état s

Les transitions entre les différents états s du modèle sont régies par les événements internes et les événements externes.

Les événements internes se produisent lorsque le modèle au temps t_i est dans un état s depuis $e = ta(s)$ (e représente le temps écoulé) et qu'il est sur le point d'entreprendre une transition interne au temps $tn = t_i + (ta(s) - e)$. Si aucun événement externe, comme de nouvelles entrées, n'arrive avant le temps tn, le modèle restera dans l'état s jusqu'au temps tn et par la suite fera sa transition de l'état s à l'état $s' := \delta_{int}(s)$ et la valeur de e sera remise à 0. Avant d'entreprendre la transition interne, le modèle générera des sorties $\lambda(s)$. En dernier lieu, le modèle planifiera sa prochaine transition interne au temps $tn + ta(s')$.

Les événements externes se produisent, lorsque des changements aux entrées surviennent peu importe le temps. Au moment de l'arrivée des entrées, le modèle, au temps t_i qui est dans l'état s depuis $e = ta(s)$, exécutera la fonction de transition externe et passera de l'état s à l'état $s'' := \delta_{ext}(s, e, x)$. Le modèle restera dans cet état jusqu'au prochain événement externe ou si $ta(s'')$ vient à terme (ce qui générera une transition interne). Encore une fois le temps écoulé e est remis à 0.

7.4.4 Adaptation des Modèles KARMA au Formalisme DEVS

Pour pouvoir utiliser le formalisme des DEVS classique dans l'outil de simulation KARMA, il reste encore une étape d'adaptation des modèles au formalisme et le développement des simulateurs de base. L'interface des modèles de simulation de KARMA a été brièvement présentée à la section 5.1 et la fonction virtuelle `run` étant la seule porte d'entrée possible pour l'exécution du modèle, il a fallu concevoir un adaptateur pour harmoniser les deux objets. La figure 7.6 présente cet adaptateur.

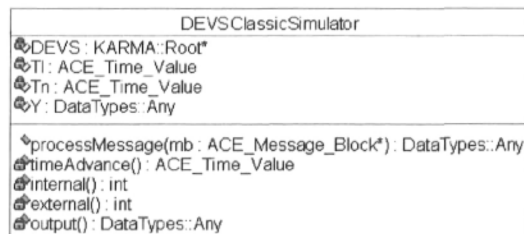

FIG. 7.6 – UML^{TM}de l'adaptateur DEVS pour les modèles KARMA basé sur le simulateur classique DEVS

Cet adaptateur a été créé à partir du simulateur DEVS classique de base décrit par Zeigler [ZPK00]. Ce simulateur doit généralement fonctionner en collaboration avec un parent ou un coordonnateur qui permet l'échange des messages entre d'autres parents ou coordonnateurs. La figure 7.7 résume l'interaction entre les simulateurs DEVS et les

coordinateurs DEVS. Cette interaction est caractérisée par l'envoi de certains messages spécifiques au protocole DEVS.

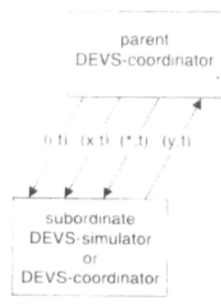

FIG. 7.7 – Résumé du protocole du simulateur DEVS [ZPK00]

Ce protocole commence par l'envoi d'une méthode d'initialisation par le message (i, t) au temps t par le parent à tous les subordonnés. Les événements sont produits par deux types de messages à l'intention des simulateurs DEVS, les événements internes représentés par les messages $(*, t)$ et les événements externes avec entrées x représentés par les messages (x, t). Les sorties des modèles sont quant à elles représentées par les messages (y, t) envoyés en direction du coordonnateur qui pourra alors les redistribuer aux modèles concernés.

La figure 7.8 présente l'algorithme qui régit les interactions internes et externes du simulateur DEVS classique. Cet algorithme est basé principalement sur les messages reçus, les messages envoyés et le temps des événements déjà planifiés. Selon le protocole, les événements internes doivent survenir exactement au prochain temps prévu tn et les événements externes doivent survenir entre le dernier événement et le prochain prévu par la fonction d'avancement du temps $ta(s)$. Le protocole permet aussi de détecter les erreurs de synchronisation qui pourrait être présentes dans l'échange des informations et des événements avec le coordonnateur, mais curieusement aucune solution n'est proposée pour les résoudre. De plus, même si le protocole ne le spécifie pas clairement, les deux variables de temps tn et tl doivent être communiquées au coordonnateur pour qu'il transmette les événements internes au bon moment au simulateur classique DEVS.

Le simulateur classique DEVS a été conçu dans le but d'être le plus simple possible tout en étant le plus flexible possible. Ses actions sont précisément déterminées selon les informations qu'il reçoit, mais son coordinateur n'est pas déterminé avec précision et ses interactions sont les plus simples possible. L'interchangeabilité dans les protocoles du coordinateur permettra donc au système entier et à tous les simulateurs classiques DEVS de se comporter de différemment. Le coordonnateur en question sera alors responsable d'implémenter des algorithmes de plus haut niveau et de contrôler le type de

```
Devs-Simulator
    variables :
        parent      //parent coordinator
        tl          //time of last event
        tn          //time of next event
        DEVS        //associated model
                    with total state (s,e)
        y           //current output value of the associated model
    when receive i-message (i,t) at time t
        tl = t − e
        tn = tl + ta(s)
    when receive *-message (*,t) at time t
        if t! = tn then
          error : bad synchronization
          y = λ(s)
          send y-message (y, t) to parent coordinator
          s = δ_int(s)
          tl = t
          tn = tl + ta(s)
    when receive x-message (x, t) at time t with input value x
        if not (tl ≤ t ≤ tn) then
          error : bad synchronization
          e = t − tl
          s = δ_ext(s, e, x)
          tl = t
          tn = tl + ta(s)
    end Devs-Simulator
```

FIG. 7.8 – Algorithme du simulateur classique DEVS [ZPK00]

simulation exécutée, comme une simulation conservatrice ou optimiste. Un des avantages les plus intéressants de ce simulateur de base est qu'il pourra être utilisé autant dans une simulation séquentielle que dans une simulation parallèle, ce qui n'est pas le cas de tous les protocoles proposés par la littérature dans le domaine.

L'implémentation de cet adaptateur, que présente la figure 7.6, nécessite cependant certains changements radicaux au niveau de l'interface et de l'implémentation des modèles KARMA. La fonction virtuelle `run` devra dorénavant retourner les variables d'état à l'intérieur d'une structure unique pour tous les modèles, par exemple : l'utilisation d'un dictionnaire (`std::map <std::string, DataTypes::Any>`) pourrait résoudre le problème. Ce changement permettra de se conformer au formalisme $\lambda : S \rightarrow Y^b$, mais aussi d'utiliser une technique de transfert des données de type poussée au lieu de tirée. La technique de *poussée* (*push*) des données, dans un contexte parallèle ou distribué, a l'avantage de n'avoir qu'un message d'information à transmettre, au lieu de deux dans la technique de *tirée* (*pull*) où le transfert de la requête et de l'information nécessite deux messages. Dans l'implémentation de l'architecture KARMA, les modèles utilisent la fonction `getParameter`, vue dans la section 5.2, pour récupérer les variables d'état.

7.4.5 Simulateur DEVS conservateur

Pour assurer la coordination de tous les différents modèles DEVS de base, la simulation a besoin d'un protocole supplémentaire pour échanger les entrées et sorties et déterminer quels événements doivent être considérés comme étant parallèles et quand ils doivent être exécutés. Cette tâche de synchronisation et de gestion des événements pour un environnement parallèle ou distribué doit alors être attribuée à un second protocole. Ce genre de protocole comporte deux catégories majeures : les protocoles conservatifs et optimistes. Les premiers évitent impérativement toute violation du principe de causalité, les deuxièmes s'assurent de pouvoir les détecter et de les réparer par la suite.

Dans l'implémentation du projet avec KARMA, la première catégorie de protocole est celle qui a été choisie, car elle est plus simple à implémenter pour une première itération. La structure générale du simulateur parallèle est présentée à la figure 7.9 et il est possible de rapidement constater comment le nouveau protocole de deuxième niveau gère l'échange d'information entre les différents simulateurs DEVS. L'idée principale à la base du simulateur conservatif DEVS est de maintenir un réseau d'information concernant deux données cruciales à l'avancement du temps de la simulation soit le plus petit temps de sortie (*earliest output time* : *EOT*) et le plus petit temps d'entrée (*earliest input time* : *EIT*). Ces deux variables représentent les estimations de temps des prochains événements générés par les modèles et sont distribuées par le système de

null message. Ce système doit être maintenu en parallèle avec la communication et le traitement des événements.

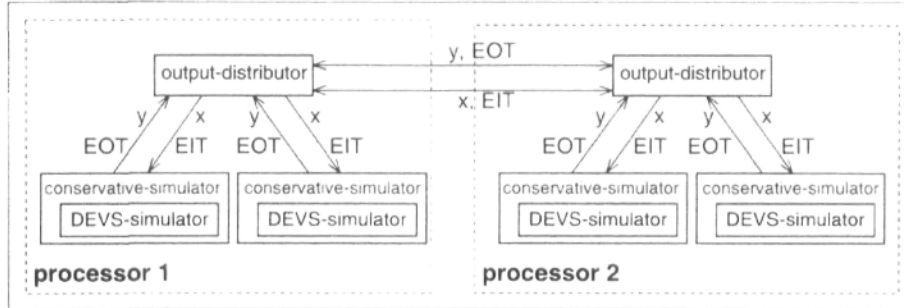

FIG. 7.9 – Structure du simulateur parallèle DEVS conservatif [ZPK00]

La variable EOT pour un modèle quelconque précise le temps auquel les prochaines variables d'état venant de la fonction de génération de sorties $y = \lambda(s)$ seront disponibles pour le ou les modèles ayant ces mêmes variables en entrées. Lorsqu'un modèle reçoit toutes les valeurs de EOT pour toutes ses entrées, il peut déterminer la valeur de la variable EIT qui est la plus petite des valeurs de EOT reçues. La formule utilisée est indiquée à l'équation 7.1.

$$EIT_{i,ip} = min\left\{EOT_{o,op}|((o,op),(i,ip)) \in IC\right\} \tag{7.1}$$

La variable $EIT_{i,ip}$ détermine le temps auquel l'entrée ip (*input port*) d'un modèle i recevra sa prochaine donnée à partir de tous les composants lui fournissant une entrée (*Input Component* ou IC). À partir de cette donnée, l'équation 7.2 décrit comment le minimum de tous les EIT de chaque entrée est calculé. Le simulateur conservatif DEVS peut alors déterminer quels événements dans sa liste ne causeront pas de violations du principe de causalité et les envoyer au simulateur DEVS classique pour qu'il les exécute.

$$EIT_i = min_{ip}\left\{EIT_{i,ip}\right\} \tag{7.2}$$

Le calcul des deux équations précédentes pour trouver la valeur du EIT est très aisé. Le problème d'implémentation survient lors du calcul de la valeur du EOT, c'est-à-dire la projection d'un intervalle de temps pour lequel le modèle ne produira pas de valeurs de sorties. Le EOT est le nom donné par le formalisme de Zeigler [ZPK00]

pour représenter la variable résultant du calcul de l'anticipation d'un modèle. Ce calcul, décrit par l'équation 7.3, se représente par une combinaison de l'état présent du modèle $q = (s, e)$, du temps du prochain événement interne tl et de la valeur du EIT. Plus simplement, ce calcul peut être réduit à choisir le minimum entre le temps du prochain événement tn et le $EIT + D$ où D représente le temps minimum de propagation du modèle. Le temps minimum de propagation est le plus petit temps possible entre la réception d'une donnée et la génération d'une sortie pour un modèle précis. Un problème majeur naît de l'évaluation de la valeur du délai de propagation D d'un modèle précis (chaque modèle possède sa propre valeur) car cette constante change en fonction de la vitesse d'exécution de l'ordinateur sur lequel la simulation est exécutée. Une solution à ce problème serait d'évaluer dynamiquement en présimulation le temps d'exécution de chaque modèle et de conserver cette analyse.

$$EOT_o = (lookahead_o(s_o, e_o), tl_o, EIT_o) = min\,\{tn_o, EIT_o + D_o\} \tag{7.3}$$

Pour faciliter une première implémentation des protocoles de simulation conservatifs et pour évaluer les performances, l'hypothèse faite à l'équation 7.4 sera appliquée dans le design du simulateur. Dans le projet avec KARMA, cette hypothèse reviendra à limiter la liste des événements sans danger à ne s'exécuter qu'à l'intérieur d'une même période de simulation pour tous les modèles. Cette hypothèse pourrait aussi avoir des effets non désirables comme le ralentissement du temps d'exécution total en créant des temps morts non désirés où certains processeurs n'effectueront pas de travail (*idle time*).

$$min\,\{tn_o, EIT_o + D_o\} = tn_o \tag{7.4}$$

Comme le montre la figure 7.9, le simulateur conservatif DEVS est relié de très près au simulateur classique DEVS et le coordinateur (*output-distributor*) ne sert qu'à transmettre les données de sorties et les estimations de temps EIT et EOT. Le simulateur conservatif DEVS, dans un environnement parallèle et distribué, détermine d'une manière autonome les événements qui doivent être traités. L'algorithme présentant le protocole du simulateur DEVS conservatif est décrit à la figure 7.10.

Le simulateur contient une boucle principale et doit aussi réagir à quatre événements externes. Les quatre événements externes sont en lien avec la réception de quatre sortes de messages venant du coordinateur (*output-distributor*) ; le message d'initialisation (*i*-message), le message d'estimation de temps pour un port précis EIT_{ip}, le message

```
Conservative-Devs-Simulator
  variables :
    parent              //parent output distributor
    devs − simulator    //associated DEVS simulator or coordinator
    tn                  //time of internal event (maintained by the devs-simulator)
    tl                  //time of last event (maintained by the devs-simulator)
    Iq                  //queue to store the external input messages
    EIT                 //earliest input time estimate
    {EIT_ip}            //earliest input time estimate for all input ports ip
    EOT                 //earliest output time estimate
    lookahead           //function to compute the lookahead of the model

  when receive i-message (i, t) at time t
    send i-message (i, t) to devs − simulator
    EIT = 0, EIT_ip = 0
    compute new EOT = lookahead (q, tl, EIT) and send it back
      to parent
  when receive new EIT_ip estimate for input port ip
    compute EIT = min_ip{EIT_ip}
    compute new EOT = lookahead (q, tl, EIT) and send it back
      to parent
  when receive x-message (x, t) at time t with input value x
    store (x, t) in input-queue Iq
  when receive y-message (y, t) at time t from subordinate
    devs − simulator send y-message (y, t) to parent
      s = δ_int(s)
  event-loop :
    while true do
      (x, tx) is first in Iq //(x, tx) = (null, ∞) if Iq is empty
      if (tn ≤ EIT&&tn ≤ tx) then //process internal event
        send(∗, tn) message to your devs − simulator
      else if (tn ≤ EIT&&tn < tx) //process external event
        remove first of input queue Iq
        send(x, tx) message to your devs − simulator
      compute new EOT = lookahead (q, tl, EIT) and send it back
        to parent
      end while
  end event-loop
end Devs-Simulator
```

FIG. 7.10 – Algorithme du simulateur conservatif DEVS [ZPK00]

d'occurence d'un événement externe (x-message) et le message de sortie venant du simulateur classique DEVS (y-message).

À l'initialisation, le simulateur conservatif doit réagir au i-message en le transférant vers le simulateur DEVS et en calculant, puis en propageant son estimation EOT à partir de la fonction *lookahead*. Quand le simulateur reçoit un message externe contenant des entrées (x-message), il ne le traite pas sur le champ, mais l'insère dans sa liste des événements (Iq). À chaque fois que le simulateur reçoit une nouvelle estimation du EIT, il calcule une nouvelle valeur de sa propre estimation EOT et doit la propager par la suite via le coordonnateur. Les données qu'il reçoit du simulateur classique DEVS ($y-$message) sont directement retransmises au coordonnateur pour être propagées aux autres simulateurs conservatifs pour générer d'autres événements externes. Le coeur du simulateur est évidemment la boucle infinie qui permet le traitement des événements internes et externes contenus dans la liste (Iq). Les événements sont traités en ordre et sont envoyés au simulateur classique DEVS lorsqu'ils sont considérés sans danger par les estimations de temps EIT reçues des autres simulateurs. Les événements doivent être classés en ordre croissant de leur estampille temporelle dans la liste pour éviter toute violation du principe de causalité. À chaque boucle, le simulateur conservatif calcule une nouvelle estimation de son EOT et le propage. L'algorithme du simulateur conservatif néglige cependant certains éléments dont il faut aussi discuter.

Premièrement, l'algorithme ne résout pas le problème des événements simultanés. Ce problème survient lors de la réception d'un message externe et de son insertion dans la liste des événements Iq. La question qui se pose est comment classer deux événements externes qui possèdent la même estampille temporelle et lequel doit être prioritaire à l'autre ? Cette question est réglée de différentes manières dans la littérature scientifique [Fuj00], mais la solution privilégiée est de déterminer à l'avance une priorité statique à chaque modèle selon l'expertise scientifique des modeleurs. De cette manière, les modèles ont été modifiés dans KARMA pour leur ajouter une variable spécifiant une priorité relative en lien avec les BaseEntity et leurs Parts. Cette variable est une très petite valeur de temps[2] qui est ajoutée à l'estampille temporelle du message lors de son classement dans la liste des événements. Cette priorité doit cependant être enlevée lorsqu'on traite ces messages pour ne pas affecter les comparaisons avec les estimations de temps.

Deuxièmement, l'algorithme exécute une boucle sans fin qui ne possède pas de mécanisme pour s'arrêter. La solution la plus courante dans le monde de la simulation parallèle et distribuée est de modifier la boucle pour qu'elle s'arrête d'elle-même lors-

[2]Cette valeur doit être plus petite que la résolution de l'horloge de simulation, par exemple si l'horloge de simulation a une résolution de 0.01 s, la valeur de la priorité peut être de l'ordre de 1^{-5}.

qu'elle dépasse un temps de simulation limite. Habituellement, une simulation est exécutée pour un certain temps défini par l'utilisateur au début de la simulation, ce temps est alors propagé dans un message d'initialisation (*i*-message) à tous les simulateurs parallèles avant le départ de la simulation. Le fonctionnement du simulateur original de KARMA emploie cette technique. Cependant, comme le protocole conservatif se base sur d'autres simulateurs pour permettre l'avancement du temps dans la simulation, il serait alors impossible d'arrêter simplement un simulateur lorsque son temps est dépassé sans avertir les autres simulateurs concernés, c'est-à-dire ceux reliés par les estimations de temps EIT et EOT. Avant d'arrêter un simulateur, il faut aussi modifier le protocole pour qu'il propage une dernière valeur de son estimation EOT qui doit être plus grande que le temps limite de la simulation. L'arrêt d'un simulateur parallèle ne se fait donc pas instantanément, mais bien à la manière d'une cascade, un simulateur entraînant l'autre.

Un point à souligner dans l'algorithme du simulateur conservatif DEVS est l'importance du temps estimé de la prochaine sortie EOT et de la fonction *lookahead*. Comme discuté précédemment, l'utilisation de cette technique de propagation des estimations de temps par les variables EOT et EIT (une variante du protocole des *null messages*) est la pierre angulaire des gains qu'il est possible de tirer de ce protocole parallèle. Cependant, le temps système (*overhead*) produit par cette technique peut être très grand comparé à l'exécution seule des événements par les simulateurs classiques DEVS. Plusieurs études ont été faites sur les manières de réduire ce temps système et elles analysent précisément quand il est nécessaire de calculer et de propager cette valeur sans entraîner de calculs inutiles et redondants.

L'implémentation de ce protocole s'est faite encore une fois à l'aide du cadre d'applications ACE et de la classe `ACE_TASK`. L'interface de la classe `DEVSThreadPool` (qui représente le simulateur conservatif DEVS) est donc très semblable à celui de la classe `ACE_TASK`. L'implémentation du protocole du simulateur conservatif DEVS requiert cependant quelques changements. Par exemple, il est difficile de coder un fil exécutant une boucle infinie et réagissant aussi à des interruptions externes comme les messages venant du distributeur. L'implémentation du protocole doit donc reposer sur une boucle principale comportant deux listes, la première étant la liste des événements et la deuxième la queue de messages incluse dans l'objet ACE_TASK, la `ACE_Message_Queue`. L'algorithme implémenté dans la classe `DEVSThreadPool` boucle tout d'abord sur la file de messages. Si un message est présent, il agit selon le type de message en accord avec l'algorithme de la figure 7.10 et, par la suite, il continue sont exécution en vérifiant la liste des événements.

Le design UML™du simulateur conservatif DEVS et du simulateur classique DEVS

est présenté à la figure 7.11. Le lien entre les deux algorithmes DEVS n'a pas à être implémenté de manières plus complexes car le simulateur classique DEVS (algorithme figure 7.[illegible]) ne fait que réagir à l'arrivée de messages et ne fait rien autrement. Une simple agrégation et des appels directs à partir de la seule fonction publique `processMessage` sont suffisants. Cette fonction retournera toujours une sortie. Il peut ne pas y avoir de données de sorties sur les variables d'états ($y = \lambda(s)$), mais le simulateur classique DEVS doit toujours communiquer les variables tn et tl au simulateur conservatif DEVS.

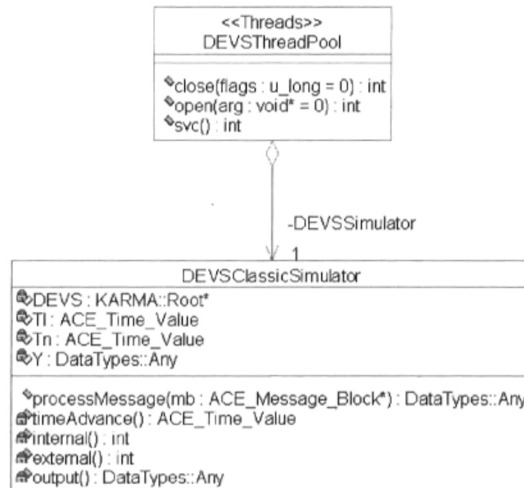

FIG. 7.11 – Diagramme de classe du simulateur conservatif DEVS et du simulateur classique DEVS

7.4.6 Le parallélisme dans KARMA

Cette section a pour but de présenter l'exécution séquentielle des modèles dans une simulation de KARMA et de la comparer à ce qui est réalisable dans la version parallèle.

L'exécution des modèles dans une simulation séquentielle avec KARMA se fait par périodes désignées à l'avance dans les propriétés de chaque modèle. Les modèles BaseEntity ont généralement une période de 0.01 s ou 0.05 s, ce qui assure un rafraîchissement régulier de la position tridimensionnelle dans le théâtre d'engagement. Les modèles Parts ont une période d'exécution égale ou plus grande que celle de leur BaseEntity. Les Parts qui ont des liens stricts entre les entrées et les sorties possèdent aussi une priorité (temporelle) qui assure leur exécution dans un ordre précis.

La figure 7.12 présente la position en x et y dans l'espace des modèles dans le

scénario où deux avions partent perpendiculairement et le missile utilise une navigation proportionnelle pour atteindre sa cible, le Aircraft3DOF. Dans ce scénario spécifique, le Missile possède une période de 0.01 s, le AircraftLauncher et le Aircraft3DOF possèdent une période de 0.05 s.

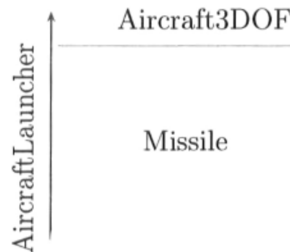

FIG. 7.12 – Scénario d'engagement entre 2 avions

Le tableau 7.1 (a) montre l'exécution séquentielle qui s'établit entre les différents modèles BaseEntity pour le scénario présenté auparavant. L'ordre d'exécution des modèles BaseEntity est établi à l'initialisation par l'ordre alphabétique du nom des modèles et reste identique au courant de la simulation, car les modèles se rappellent séquentiellement dans cet ordre dans la liste des événements. À certains temps de la simulation, en rapport avec la période d'exécution des modèles, les trois modèles BaseEntity doivent s'exécuter. Par exemple, pour deux modèles ayant respectivement 0.05 s et 0.10 s de période, les instants d'exécution simultanés seraient des multiples de 0.10 s. À ces instants déterminés, la simulation séquentielle pourrait avantageusement utiliser plus d'un processeur puisque plus d'un modèle a besoin de s'exécuter.

Le tableau 7.1 (b) montre l'exécution des modèles Parts qui se trouvent à l'intérieur du Missile et l'utilisation de la variable priorité qui ajoute un supplément à estampille temporelle pour assurer l'ordre d'exécution et leur classement adéquat dans la liste des événements centrale. Dans l'exemple du tableau 7.1 (b), le Missile est équipé de sept Parts qui possèdent une priorité de 1 à 7 puisqu'elles ont toutes une relation entre les entrées et les sorties à respecter. Il est à noter que l'ajout de la priorité à l'estampille temporelle, les trois dernières décimales, doivent rester invisibles à la simulation pour ne pas perturber les comparaisons de temps qui s'effectuent. Il est donc impératif de n'utiliser cette variable qu'avant l'insertion dans la liste des événements et de l'enlever lorsque le modèle est retiré de la liste.

Comme précisé précédemment, le premier niveau de parallélisme disponible dans la simulation séquentielle avec KARMA se situe entre les différents modèles BaseEntity. Avec l'utilisation de plus d'un processeur pour exécuter plus d'un modèle, la simulation devrait gagner du temps lors du traitement des événements simultanés. La figure 7.13

Temps	Nom du Modèle	Temps	Nom du Modèle
0.00	Aircraft3DOF	0.00000	Missile
0.00	AircraftLauncher	0.00001	Sensor
0.00	Missile	0.00002	Seeker
0.01	Missile	0.00003	Guidance
0.02	Missile	0.00004	Autopilot
0.03	Missile	0.00005	Control
0.04	Missile	0.00006	Propulsion
0.05	Aircraft3DOF	0.00007	Airframe
0.05	AircraftLauncher	0.01000	Missile
0.05	Missile	0.01001	Sensor
0.06	Missile	0.01002	Seeker
0.07	Missile	0.01003	Guidance
0.08	Missile	0.01004	Autopilot
0.09	Missile	0.01005	Control
0.10	Aircraft3DOF	0.01006	Propulsion
0.10	AircraftLauncher	0.01007	Airframe
0.10	Missile	0.02000	Missile
(a)		(b)	

TAB. 7.1 – Exécution séquentielle des modèles pour le scénario spécifié

présente le gain possible venant de l'utilisation de plus d'un processeur pour simuler plusieurs modèles BaseEntity et l'effet sur le temps total de la simulation. Le deuxième niveau de parallélisme se trouve dans l'exécution des Parts à l'intérieur d'un même BaseEntity. L'exploitation du parallélisme de ce niveau représente cependant plus de problèmes que le premier puisqu'il implique d'exécuter les Parts d'un même BaseEntity sur plus d'un processeur et ne sera pas traité dans ce projet.

Comme il est possible de le constater, l'élimination des redondances temporelles est aisée pour un cas de N modèles distribués sur N processeurs, mais il se complique pour des cas de N modèles avec M processeurs ou N est plus grand que M (N > M). La figure 7.13 explique visuellement les gains en temps possibles, mais ne tient pas compte du temps d'exécution différent de chaque modèle. Comme chaque modèle possède des calculs différents, son temps d'exécution est aussi très différent. Dans l'exemple du scénario d'engagement avec avions perpendiculaires, le modèle qui exige le plus de calculs est le Missile, car il est entre autres composé d'un aérodynamisme à 6 degrés de liberté (6DOF). La figure 7.13, indique que certains trous temporels sont créés au niveau des deux premiers processeurs (P_1 et P_2) pour l'exécution parallèle. Pour simplifier l'implémentation de la première version du programme, ces problèmes d'ordonnancement

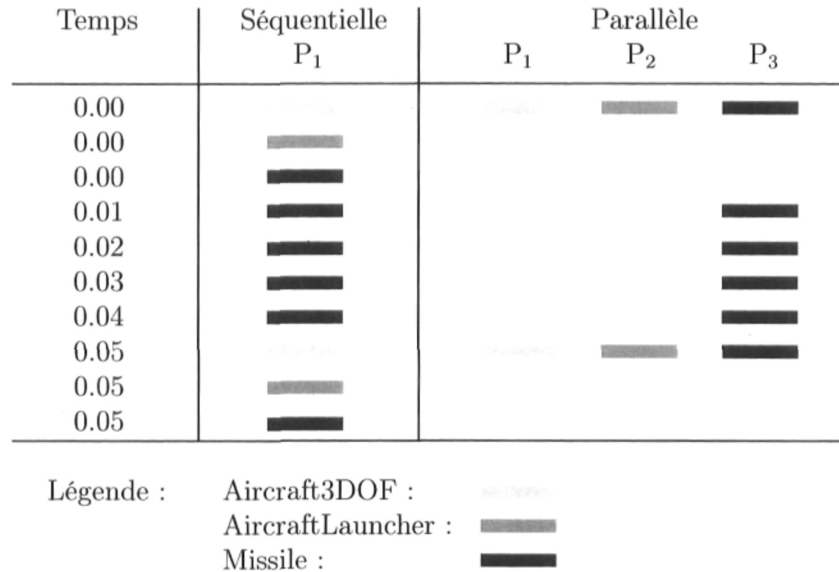

FIG. 7.13 – Exemple de l'exploitation du parallélisme sur plus d'un processeur pour le scénario spécifique de l'engagement des 2 avions.

(*scheduling*) bas niveau ont été laissés au système d'exploitation. Cependant, dans un futur contexte d'optimisation ou de simulation temps-réel, le problème de la gestion statique ou dynamique des tâches pourrait survenir. Ce problème ne fait pas partie du présent projet, mais sera quand même abordé dans la section suivante.

7.5 Limites et optimisations

La première limite à l'efficacité du parallélisme a été évoquée précédemment et est mieux connue sous le nom d'équilibrage de charge (*load balancing*). Ce problème, très courant dans le milieu des systèmes d'exploitation, revient à être capable de minimiser le temps mort dans l'exploitation de plusieurs processeurs en jouant avec le déplacement des tâches à effectuer sur ces différents processeurs. Le résultat d'un mauvais équilibrage des différents processeurs est que les tâches accomplies sur ce dernier (trop occupé) ralentissent et bloquent celles sur les autres processeurs (dans l'attente des résultats). L'équilibrage de charge peut être calculé de manière statique à partir du scénario qui est chargé dans KARMA lors d'une phase d'initialisation à l'aide d'algorithmes et par la suite exécuté [SRK01]. Cependant, si le scénario contient une génération dynamique de modèles, cette solution ne convient plus et il ne reste que le *dynamic load balancing* où les tâches dont distribuées de manière dynamique sur les processeurs disponibles. Cette solution plus complexe requiert des études approfondies dans le domaine du temps-réel

qui dépasse l'ampleur du présent projet. Pour les besoins de KARMA et pour simplifier le développement, un ordinateur avec deux processeurs sous Windows a été utilisé. Ce système d'exploitation ne permet cependant pas de manipulations au niveau de la gestion des tâches autres que la priorité accordée à celles-ci. Windows fonctionne avec une technique nommée SMP (*Symetric Multi Processor*) qui divise également (ou le plus possible) les tâches sur les différents processeurs disponibles. Lors d'un éventuel port sur un système d'exploitation plus configurable comme Unix®[Sysb], l'ajout d'un algorithme permettant un équilibrage des charges plus approprié de l'exécution des modèles serait une excellente optimisation.

La deuxième limite potentielle au parallélisme est le temps système que pourrait créer le choix du protocole $EOT - EIT$ semblable à celui des *null message*. L'engorgement des liens de communication pourrait ralentir la transmission des éléments importants comme les événements externes et surcharger les processeurs. Les études sur ce protocole montrent que, pour qu'il soit efficace, il faut qu'il y ait beaucoup d'événements générés par les modèles. À ce moment, les estimations de temps transmises $EOT - EIT$ peuvent désigner une fenêtre de temps étant sécuritaire par rapport à la causalité et les simulateurs conservatifs peuvent alors exécuter les événements contenus dans cette fenêtre. Si trop peu d'événements sont disponibles, presque aucun événement n'est alors désigné comme sécuritaire par le protocole et les messages sont envoyés, mais n'ont pas vraiment d'utilité, d'où le temps système non nécessaire. Encore une fois, plusieurs recherches se sont portées sur ce protocole, introduit pour la première fois par Chandy et Misra [CM79] en 1979, pour améliorer ses performances. Ces recherches ont surtout tenté d'analyser le meilleur moment pour générer et distribuer ces estimations. Les solutions avancées varient entre l'envoi de temps à autre, l'envoi seulement quand les processeurs sont tous bloqués ou l'envoi sur demande [ZPK00]. Toutes ces solutions offrent des avantages, mais démontrent aussi des faiblesses [Lub89] [Mis86] [Nic91]. Chaque solution est adaptée à un type précis d'environnement avec des conditions précises de simulation.

La troisième limitation aux performances du simulateur parallèle viendra probablement de l'hypothèse (Éq. 7.1) formulée précédemment quant à la manière de calculer l'estimation EOT de chaque modèle à l'aide de la fonction *lookahead* puisque les performances du protocole parallèle conservatif sont basées sur une bonne évaluation de l'anticipation. Cette évaluation permet d'estimer correctement la fenêtre de temps dans laquelle les événements sont sécuritaires. Une estimation trop petite de cette fenêtre amène le nombre d'événements estimés comme étant parallèles à diminuer, diminuant aussi le degré de parallélisme. Bien que son augmentation amène une croissance du parallélisme, une estimation trop grande de cette fenêtre n'est pas non plus profitable, car elle désignerait erronément certains événements parallèles. Théoriquement, l'hypothèse

faite sur l'estimation EOT devrait apporter le parallélisme désiré comme montré à la figure 7.13. En fixant la fenêtre égale à chaque période de temps, tous les événements se trouvant dans cette fenêtre, c'est-à-dire la même période, seront considérés comme étant parallélisables, ce qui est effectivement le cas. Parfois, les modèles développés ne peuvent pas être compatibles avec la notion de l'anticipation, car l'exécution de ces modèles ne comporte pas de notion de temps. Certains modèles comme un senseur détectent ou non la présence de leur cible sans faire de calculs mathématiques. Les modèles qui livrent une réponse immédiate possèdent donc une anticipation nulle. Pour ces cas, l'utilisation d'un protocole de simulation comme les *null messages* est impossible puisqu'une chaîne infinie avec une anticipation nulle, entre ces modèles particuliers, peut exister et donc créer un interblocage. Pour contourner ce problème, certaines recherches se sont portées sur le développement d'un protocole supportant la notion de l'anticipation nulle (*Zero lookahead*) [Pag98]. Ce type de protocole [Fuj97] utilise une technique de requêtes centrales pour faire avancer le temps des éléments distribués plutôt que de laisser les modèles l'avancer eux-mêmes. Une implémentation de ce type de protocole peut être utilisée dans les services fournis par le HLA par l'entremise d'appels aux fonctions `nextEventRequest` et `timeAdvanceRequest`. Ces protocoles ont été développés pour résoudre le problème qui survient lorsque la valeur de l'anticipation n'est pas intrinsèque au fonctionnement du modèle lui-même. Ces protocoles sont aussi surtout utilisés et efficaces pour des environnements de simulation distribués.

Chapitre 8

La communication

Pour réussir à faire fonctionner un simulateur parallèle comportant plusieurs processus, il faut implémenter un système de communication pour soutenir les protocoles d'échanges de données et d'informations entre les différents processus. Dans le cas d'un protocole comme celui des DEVS, qui doit envisager un environnement distribué autant que parallèle, ce système de communication doit jouer un rôle précis et complexe. Pour relier les différents simulateurs conservatifs DEVS, le système de communication doit remplacer le *output-distributor* comme vu à la figure 7.9. Ce rôle est déterminé à l'avance par le protocole et doit répondre à des besoins précis.

- Transférer les estimés de temps EOT et EIT
- Transférer les messages venant des autres simulateurs, x-message
- Transférer les données (entrées et sorties), y-messages

Cependant, pour une utilisation dans une simulation locale (sur un même ordinateur) contenant certains fils, ce système de communication doit remplir moins d'obligations à cause de l'environnement de simulation moins complexe. Par exemple, comme les fils se situent dans le même espace mémoire, il n'est pas nécessaire de transférer les données entre les processus. L'utilisation d'une structure commune et l'application d'une exclusion mutuelle permettent d'éviter les accès simultanés aux données. Les besoins de ce système de communication sont donc moins grands que pour un protocole formel impliquant une simulation distribuée et la principale tâche de ce système est de transférer les commandes venant de l'ordonnanceur

Le plus simple des deux systèmes de communication a été construit en premier dans le but de concevoir un simulateur principalement parallèle s'exécutant dans un environnement local multiprocesseurs. Le design du deuxième système de communication a fait

son apparition relativement tard dans l'élaboration de la première version du simulateur pour répondre à la demande d'extension future du simulateur dans un environnement distribué. Pour cette raison et pour certains soucis de performance, deux systèmes de communication et de messages ont été créés. Le premier, le plus simple, tire profit des capacités du cadre d'applications ACE et de l'objet `ACE_Task`. Il utilise la file de messages intégrée pour communiquer avec chaque objet contenant un ou plusieurs modèles. Le deuxième, plus complexe, repose sur un patron de communication réseau pour démultiplexer les messages et les rediriger vers chaque objet `ACE_Task` implémentant le protocole conservatif DEVS.

8.1 Communication interprocessus (IPC)

Les divers mécanismes de communication inter processus qui sont implémenté dans ACE, comme dans plusieurs systèmes d'exploitation, sont les suivants.

- tuyau (*pipe*)
- signaux
- sémaphores
- mémoire partagée
- interface de connexion (*socket*)
- file de Messages

Puisque les fils sont en exécution constante dans un espace mémoire commun, il est impératif d'utiliser des outils de synchronisation si les fils ont besoin de communiquer entre eux. Cette communication, de préférence asynchrone pour éviter de bloquer les différents fils, peut être réalisée de différentes manières. La figure 8.1 illustre le principe de synchronisation de deux fils par un troisième objet.

L'utilisation des tuyaux est un procédé bien connu dans le monde Unix et Linux [Sysa] où la sortie d'un premier processus devient l'entrée pour un second processus. Cependant, la documentation sur les diverses implémentations des tuyaux dans ACE prévient les utilisateurs des problèmes de portabilité entre les différents systèmes d'exploitation. Pour cette raison, les tuyaux n'ont pas été pris en considération pour implémenter le système de communication.

L'utilisation des signaux se fait par l'entremise du système d'exploitation. Un signal est envoyé par un processus à l'aide d'une fonction du système d'exploitation qui s'occupe de le redistribuer aux autres processus s'exécutant sur le même ordinateur.

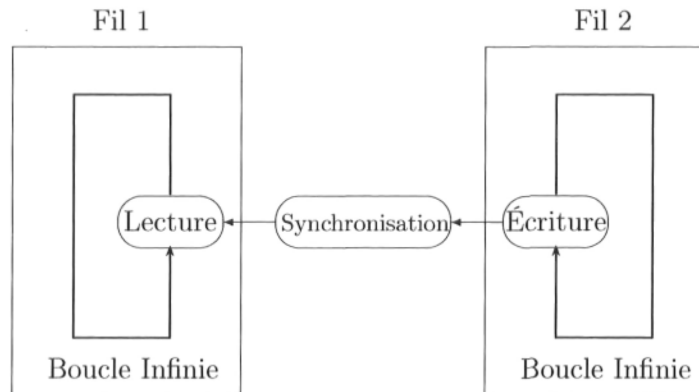

FIG. 8.1 – Synchronisation de deux fils à l'aide d'un mécanisme IPC semblable au principe du producteur-consommateur

Le problème empêchant l'utilisation de ce procédé est que le système d'exploitation Windows ne supporte (ou émule) que très peu de signaux. Entre autres, Windows ne supporte pas les signaux SIGUSR1 et SIGUSR2 sur lesquels ACE se base. Pour cette raison, les signaux n'ont pas été pris en considération.

Les sémaphores permettent aux processus de s'informer sur le statut d'une information en particulier. Ils sont souvent utilisés pour surveiller et contrôler la disponibilité de ressources partagées. Cependant, aucune réelle information ne peut être transférée entre les processus. Pour cette raison, les sémaphores n'ont pas été pris en considération.

La gestion de la mémoire dans ACE peut se faire de plusieurs manières différentes en instanciant l'une des multiples classes de réserve de mémoire (*memory pool*) disponibles. Certaines de ces classes offrent des services de mémoire partagée en se basant sur les interfaces de chaque système d'exploitation. L'utilisation des services de mémoire partagée pour transférer les données a été analysée dès le début du projet. Son utilisation s'est avérée non nécessaire pour l'implémentation du simulateur multiprocesseur local, car les fils offrent déjà un espace mémoire commun qui agit comme une mémoire partagée. Dans le projet du simulateur distribué, l'utilisation de la mémoire partagée, qui se devrait également d'être distribuée sur plusieurs ordinateurs, implique la résolution de certaines complications technologiques comme la réplication des données qui réduisent l'intérêt de son utilisation. Pour ces raisons, la mémoire partagée n'a pas été prise en considération.

Les interfaces de connexion fournissent une communication bidirectionnelle point à point entre deux processus. Les interfaces de connexion sont très versatiles et sont

l'une des bases pour la communication interprocessus et intersystèmes. Ils permettent autant une communication locale que distribuée en utilisant le protocole Internet. Pour un projet nécessitant une implémentation distribuée, les interfaces de connexion sont la seule solution possible en utilisant un protocole de communication réseau comme TCP/IP ou UDP. Bien entendu, toute solution de haut niveau utilisant les interfaces de connexion à leur base est aussi une bonne solution. Dans le contexte du design du système de communication flexible permettant la distribution sur le réseau, l'utilisation des interfaces de connexion a été retenue pour l'implémentation de la communication.

Les files de messages sont des outils très efficaces pour transférer des messages interprocessus dans un environnement local multiprocesseurs. Le type de communication principalement utilisé avec les files de messages est celui du producteur/consommateur, où un processus remplit la file de messages et où un autre les retire pour les traiter. Comme ce système de communication interprocessus était intégré à l'avance dans le cadre d'applications ACE en étant une partie d'un objet `ACE_Task` utilisé pour implémenter les fils, l'utilisation des files de messages (`ACE_Message_Queue`) a été retenue pour la communication dans le simulateur parallèle local.

ACE redéfinit une interface commune à plusieurs outils de communication interprocessus sur plusieurs plates-formes. Cette manière de procéder qui permet une grande portabilité vient au détriment d'une rigidité qui empêche d'améliorer les classes pour garder une compatibilité accrue sur tous les systèmes d'exploitations. Quelquefois, les classes ne peuvent supporter que certaines opérations de base et l'utilisateur doit compenser cette faiblesse en implémentant lui-même certains mécanismes de gestion de plus haut niveau dans l'environnement particulier de développement de l'application. Dans presque tous les outils de communication interprocessus, le développeur doit s'assurer de la sécurité concurrentielle des outils de synchronisation pour éviter toute corruption des données au moment de l'exécution du programme. Le programmeur doit entre autres éviter les accès simultanés de plusieurs fils aux objets communs en utilisant des mutex et des gardes.

8.2 Communication locale

La communication locale interprocessus et inter processeurs est permise par chaque système d'exploitation via son interface de programmation d'applications. Le logiciel des couches intermédiaires ACE ne fait que redéfinir une interface commune et assurer la portabilité de cette interface et des opérations offertes. En général, sur la majorité des systèmes d'exploitation, les files de messages sont implémentées en utilisant un

module spécial du système d'exploitation. La figure 8.2 montre le principe général de la synchronisation de deux fils à l'aide d'une file de messages. Chaque fil a une référence sur un bout de la file, le premier écrit et le deuxième lit.

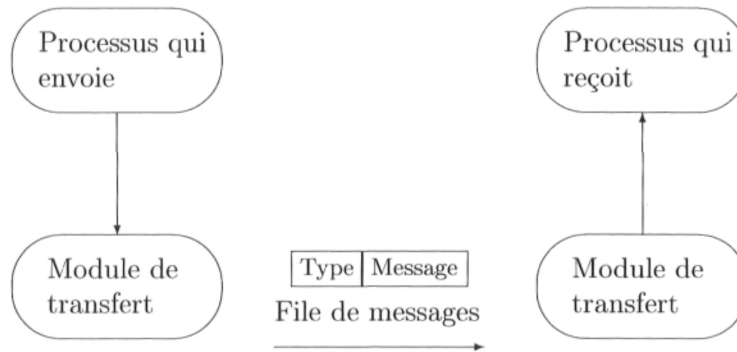

FIG. 8.2 – Structure de la communication avec les files de messages [Mar99]

8.2.1 Architecture

Dans la première version du simulateur parallèle, le coeur de l'application réside dans l'algorithme qui reçoit les événements de chaque modèle qui s'exécute, les classe dans une liste et les retire par la suite pour les faire s'exécuter par les différents fils disponibles. La majorité de ces interactions se font par le transfert de messages par le biais de files contenues dans chaque objet `ACE_Task`.

La figure 8.3 présente la structure de la communication dans le simulateur parallèle local. Cette structure est principalement basée sur l'algorithme d'exécution d'une simulation à pas de temps variable [15]. L'algorithme d'exécution par événements, légèrement modifié pour s'ajouter aux besoins spécifiques de l'outil de simulation KARMA, est contenu dans le module de l'application principale. Puisque l'application principale est chargée de générer les fils et de les initialiser avec les modèles, elle détient aussi un pointeur sur chaque objet `ACE_Task` et peut donc l'utiliser (à l'aide des fonctions `putq` et `put`) pour accéder à la file de messages incluse dans ces objets. Par la suite, chaque fil exécute un algorithme qui boucle sur la file de messages qui lui est propre. L'algorithme utilise un appel synchrone pour récupérer les messages dans la file et bloque (wait) si aucun message n'est présent. L'exécution du fil est donc suspendue jusqu'à l'arrivée d'un message qui le réveille. Cette technique permet de libérer les processeurs pour permettre à d'autres fils de s'exécuter au besoin. Lorsque le modèle s'exécute, pour calculer de nouvelles valeurs pour ses variables d'états, il génère un nouvel événement

qui est envoyé à l'application principale. Cet événement est alors classé dans la liste des événements ; il sera retiré et renvoyé au fil concerné au moment opportun.

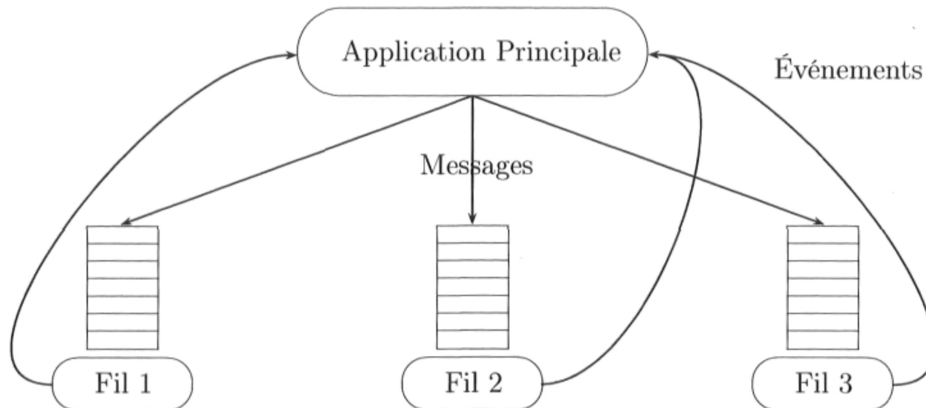

FIG. 8.3 – Structure de la communication du simulateur local parallèle

Ce système de communication unidirectionnel 1 à N[1] ne permet pas l'envoi de messages d'un modèle à l'autre. Présentement, il ne permet pas non plus l'envoi de messages vers l'application principale. Les événements sont créés directement à l'intérieur de chaque modèle et une fonction de l'ordonnanceur est appelée avec leur prochain temps d'exécution. Cette technique, qui remonte au simulateur séquentiel KARMA, bien que performante, n'est pas du tout compatible avec le formalisme DEVS $y = \lambda(s)$ et freine aussi la flexibilité recherchée dans la création possible d'un simulateur distribué. Ces limites seront traitées plus en détail à la fin de ce chapitre.

8.2.2 Design conceptuel

Tel que mentionné précédemment, le système de communication pour le simulateur parallèle local fait usage des files de messages pour transférer les messages entre l'application principale, celle qui fait la gestion des événements, et les fils, ceux qui exécutent les calculs des modèles dans un processus séparé. L'implémentation de ce système de communication s'est fait à partir de la file de messages incluse dans chaque objet de la classe `ACE_Task`. L'utilisation de la file de messages est transparente à l'extérieur de la classe `ACE_Task`, ce qui rend sont utilisation aisée. Elle est accessible par deux fonctions publiques :

```
template<ACE_SYNCH_DECL>
int ACE_Task<>::putq ( ACE_Message_Block*,
```

[1] N étant le nombre de fils générés

```
                              ACE_Time_Value* timeout = 0
                            )
```

et

```
int ACE_Task_Base::put ( ACE_Message_Block*,
                         ACE_Time_Value* timeout = 0
                       ) [virtual]
```

La fonction `putq` insère un message (`ACE_Message_Block`) directement dans la file et possède une limite de temps pour faire l'insertion. Si la variable `timeout` devient zéro, cela équivaut à n'imposer aucune limite de temps pour compléter l'opération. Il est à noter que comme la file de messages peut être utilisée par plusieurs fils, son accès peut être bloqué temporairement par un autre fil. La fonction `put` est héritée d'une superclasse, `ACE_Task_Base` et peut être redéfinie pour permettre d'exécuter un traitement supplémentaire dans la classe ACE_Task. Le message peut être traité immédiatement ou empilé sur la file de messages pour être traité ultérieurement dans la fonction `svc` par le fil. À noter que ces deux fonctions s'exécutent dans l'espace mémoire de l'application ayant fait les appels et non dans celui du fil. Ces fonctions sont donc synchrones pour l'application qui fait les appels. L'exemple suivant présente la génération d'un fil et le passage d'un message par la fonction `putq`.

```
ACE_Task* ThreadPool ;
ThreadPool.activate (THR_NEW_LWP,1) ;
ACE_Message_Block* mb =
  new ACE_Message_Block(ACE_OS::strlen("Message 1\n")) ;
ThreadPool.putq (mb, 0) ;
```

Le cadre d'applications ACE permet de faire les appels aux files de messages de deux manières différentes : les opérations synchrones ou asynchrones. Les opérations asynchrones permettent au processus empilant un message sur la file de compléter d'autres opérations sans attendre que le deuxième processus dépile le message ou lui renvoie une réponse. Cette technique assure une plus grande rapidité d'exécution pour la simulation et une meilleure gestion des ressources.

La manipulation des messages de la file de messages à l'intérieur de la classe `ACE_Task` est tout aussi facile qu'à l'extérieur. Deux fonctions publiques sont disponibles pour ce travail :

```
template<ACE_SYNCH_DECL>
int ACE_Task<>::getq ( ACE_Message_Block*& mb,
```

```
                              ACE_Time_Value* timeout = 0
                            )
```

et

```
template<ACE_SYNCH_DECL>
int ACE_Task<>::ungetq ( ACE_Message_Block* mb,
                         ACE_Time_Value* timeout = 0
                       )
```

La fonction `getq` retire le premier message sur la file de messages. Cet appel est synchrone et bloque si aucun message n'est présent sur la file de messages ou jusqu'à ce que le temps spécifié par la variable `timeout` soit dépassé. La fonction `getq` retourne le nombre d'éléments dans la file de messages si l'appel est un succès, sinon un code d'erreur est retourné. La fonction `ungetq` retourne un message sur le dessus de la file, son fonctionnement est semblable à celui de `getq`. À noter que la fonction `putq` discutée précédemment est aussi disponible et ajoute un message à la fin de la file.

Le cadre d'applications ACE contient aussi d'autres types de files de messages dont il est pertinent de discuter même si elles n'ont pas toutes été utilisées dans la version finale du simulateur parallèle local. Les files de messages (*messages queues*) ont été modélisées à partir de celles contenues dans *UNIX System V*. Les files de messages dans ACE peuvent être divisées en deux grandes catégories, les files statiques et les files dynamiques. Elles possèdent pratiquement toutes des algorithmes différents pour gérer l'insertion et le retrait des messages qui peuvent s'apparenter aux grandes catégories connues comme les files d'attente, les piles ou les files prioritaires. Toutes les files de messages sont des conteneurs qui peuvent être accédés par des itérateurs.

Les files de messages statiques, nommées (`ACE_Message_Queue`), ont été créées avec un objectif d'usage simple pour répondre aux besoins généraux d'une application. Les files de messages dynamiques, nommées (`ACE_Dynamic_Message_Queue`) ont été créées plus spécifiquement pour des applications de type temps-réel. La différence majeure entre les deux est basée sur la priorité des messages. Dans les files statiques, quand une priorité est donnée, elle ne changera plus. Par contre, dans une file de messages dynamique, la priorité change dynamiquement au cours du temps en fonction de certains paramètres comme le temps d'exécution (*execution time*) et le temps limite (*deadline*) [Syy].

L'attrait principal des files de messages dynamiques serait leur utilisation dans un contexte de simulation avec un ordonnanceur temps-réel. Elles n'ont pas été implémentées dans la version du simulateur parallèle local, car leur utilisation est relativement

complexe, mais une analyse plus poussée des algorithmes gérant le classement des messages pourrait être bénéfique dans le but d'améliorer les performances dans un contexte temps-réel. Les files de messages dynamiques possèdent deux stratégies différentes pour classer les messages ; une basée sur le relâchement (*laxity-based*) et une sur le temps limite (*deadline-based*). La stratégie basée sur le temps limite utilise le paramètre du temps limite sur chaque message pour ordonner les priorités. Le message sur la file ayant le temps limite le plus près du temps présent dans le temps aura la plus grande priorité. La stratégie basée sur le relâchement utilise deux paramètres, le temps limite et le temps d'exécution , pour ordonner les priorités des messages. Les fonctions publiques pour utiliser ses files sont :

```
virtual
int dequeue_head( ACE_Message_Block*& first_item,
                  ACE_Time_Value* timeout = 0
                )
```

et

```
virtual
int enqueue_prio( ACE_Message_Block* new_item,
                  ACE_Time_Value* timeout = 0
                )
```

La fonction `enqueue_prio` insère un message selon ses paramètres et la stratégie désignée lors de l'instanciation de la file de messages dynamique. La fonction `dequeue_-head` retire le message se trouvant sur le dessus de la file et ayant la plus grande priorité selon la stratégie choisie. L'usage de ces files de messages dynamiques pourrait, dans une simulation temps-réel effectuant des générations d'entités dynamiques, être un atout majeur pour augmenter les performances.

8.3 Communication distribuée

Pour répondre aux besoins plus complexes de communication d'un système distribué, peu de solutions autres que celles utilisant un lien réseau peuvent être suggérées. Certaines solutions matérielles comme la mémoire réflective reliée par un lien optique ont été vite écartées puisqu'elles impliquaient un développement centré sur un matériel unique, donc peu flexible et surtout peu portable.

[2]Ces paramètres sont accessibles par les fonctions publiques de l'objet ACE_Message_Block : set_deadline et set_execution_time

L'utilisation de la communication réseau a donc été le choix évident pour implémenter le transport des informations. En décidant d'utiliser le réseau pour communiquer, l'utilisation du mécanisme de communication interprocessus des interfaces de connexion devient incontournable. Les interfaces de connexion existent à l'intérieur d'un domaine de communication. Une interface de connexion est une abstraction qui fournit une structure d'adresse et un groupe de protocoles. Seules les interfaces de connexion appartenant à un même domaine peuvent communiquer entre eux. En tout, vingt-trois domaines sont disponibles pour communiquer, mais seulement deux sont couramment utilisés : le domaine Unix et le domaine internet. Le domaine intéressant pour communiquer entre deux ordinateurs dans un environnement distribué est le domaine internet. Ce domaine utilise le protocole TCP/IP pour transférer les données. Les interfaces de connexion peuvent aussi très bien communiquer entre différents processus sur un même ordinateur comme pour toute autre forme de communication interprocessus [Mar99]. Comme pour le cas de la communication locale interprocessus, plusieurs outils ont été évalués dans le choix de l'architecture du système de communication distribué.

Tout d'abord, le logiciel des couches intermédiaires nommé TAO (The ACE Orb) [Grob], développé par le groupe *Distributed Object Computing* (DOC) et présenté à la section 3.2.2, constitue une implémentation temps réel du standard de communication CORBA [OMG95]. Beaucoup d'efforts ont été portés dans l'implémentation de ce standard qui offre une interopérabilité entre diverses applications logicielles s'exécutant sur diverses plateformes et divers systèmes d'exploitation. Cependant, cette très grande flexibilité rend cet outil lourd et complexe dans son utilisation. De plus, le projet présent ne démontre pas un besoin nécessitant une aussi grande flexibilité, car le développement du prototype se fait un environnement contrôlé pour un cas particulier.

D'autres groupes de recherche académique, comme PADS (Parallel And Distributed Systems) à Georgia Tech, ont développé le FDK, présenté à la section 3.1.2, qui contient des modules de simulation numérique réutilisables, dont un offre des services de communication pour une simulation distribuée [Cor97]. Ce genre de logiciel est développé dans le but de soutenir le déploiement de simulations distribuées à grande échelle. Leur architecture est très rigide et relativement complexe à maîtriser pour le développement d'un prototype affichant une envergure plus modeste.

D'autres standards de communication comme *Message Passing Interface* (MPI) [Sta] et *Parallel Virtual Machine* (PVM) [Mac], qui permettent à un ensemble hétérogène d'ordinateurs Unix ou Windows reliés par un réseau d'agir comme un ordinateur unique parallèle, ont aussi été évalués dans la section 3.2.3. Ces protocoles bien que très performants et éprouvés ne permettent de passer que des messages textes entre les différents ordinateurs. Il serait cependant beaucoup plus intéressant de pouvoir transférer des

objets de plus haut niveau comme le permet standard CORBA.

Un outil tiré du domaine du jeu multijoueur nommé RakNet, développé par RakkarSoft, offre un API réseau qui utilise des communications UDP et des objets haut-niveau sur Windows, Linux et Unix, ont aussi été analysés [bR]. Cependant, l'intégration de RakNet aurait nécessité des changements majeurs à l'architecture des modèles de KARMA, car son mode de fonctionnement oblige l'héritage à certaines classes précises de son cadre d'application pour permettre l'utilisation des fonctionnalités disponibles. Cet outil se prête donc mal à un réusinage (*refactoring*) d'application.

Une trop grande diversification des services dans l'architecture aurait nui à l'intégration globale de ceux-ci dans un seul outil de simulation. Il était préférable de garder une meilleure homogénéïté dans le cadre d'application utilisé pour faciliter le développement et l'intégration du prototype. Pour cette raison, pour garder notre portabilité et faciliter la compatibilité avec le reste de l'outil de simulation, le patron Acceptor-Connector tiré du cadre d'applications ACE a finalement été choisi comme design pour le système de communication distribué et il sera décrit en détail dans la section suivante.

8.3.1 Architecture

Le patron Acceptor-Connector est décrit plus en détail dans [SH03], [SSB01] et [HJS03] car seuls les points intéressants pour le projet seront abordés ici. Ce patron est basé sur une fonction native à chaque système d'exploitation pour démultiplexer des événements[3] à partir d'un objet appelé le réacteur. Le patron Acceptor-Connector permet de découpler la manière de se connecter et d'initialiser des services coopératifs dans une application réseau du traitement qu'ils doivent ensuite accomplir après s'être connectés. Concrètement, l'utilisation de ce patron est d'écouter sur certains ports de l'ordinateur et d'attendre que différents processus se connectent. Lorsque ces connexions seront établies, elles seront transférées à un autre objet qui pourra en assurer le traitement. De cette manière, le traitement et l'initialisation des communications peuvent être découplés, ce qui augmente le potentiel d'extension et de réutilisation du code développé.

La première architecture du patron Acceptor-Connector a été implémentée pour fonctionner localement avec comme adresse réseau <localhost> (127.0.0.1). La figure 8.1 présente cette architecture où le réacteur surveille une liste désignée de ports réseau

[3]Ici événement ne réfère pas à la simulation, mais à des situations d'intérêts pour le système d'exploitation comme l'arrivée d'une communication réseau ou le dépassement d'un compteur

et attend les communications. Lorsqu'une communication est détectée, elle est passée à un gestionnaire (*handler*) qui exécute le traitement sur la connexion, c'est-à-dire la réception des données et reconstruit le message et le transfert à l'objet `ACE_Task` auquel le message est destiné. Après ses calculs, l'objet `ACE_Task` initie une connexion sur un port spécifique pour transmettre ses données à un autre objet `ACE_Task`.

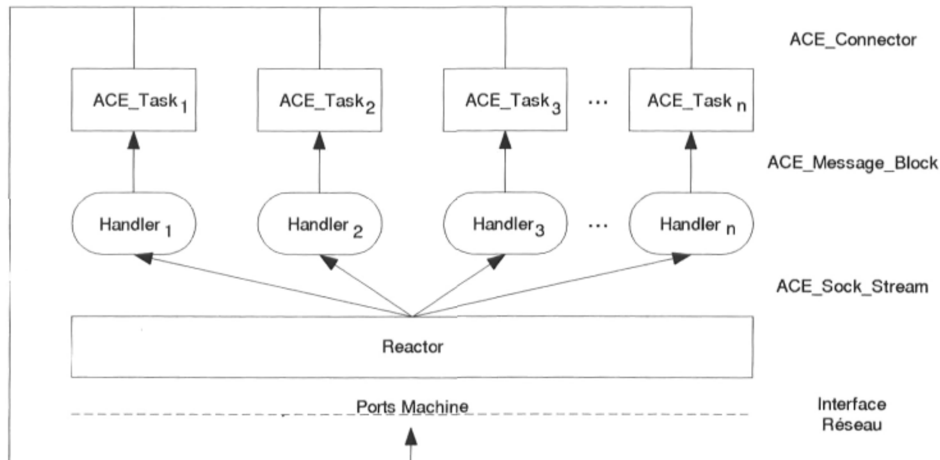

FIG. 8.4 – Représentation graphique de l'architecture de communication réseau Acceptor-Connector

Pour que la communication entre objets `ACE_Task` soit possible, il faut implanter un système de correspondance entre chaque fil qui est instancié et un port réseau particulier. Toutes les communications adressées à un même port sont retransmises au même objet `ACE_Task`. Une deuxième correspondance est aussi nécessaire entre les différents modèles pour créer la topologie de communication entre les entrées et les sorties. Ces correspondances font partie d'une phase présimulation et devraient être incluses dans les fichiers XML représentant le scénario d'engagement d'armes à simuler.

8.3.2 Design conceptuel

Le cadre d'applications ACE propose diverses implémentations pour le réacteur qui fait partie du patron Acceptor-Connector. Le rôle du réacteur est de démultiplexer les événements à partir d'une fonction spécifique de chaque système d'exploitation, d'où les diverses implémentations. Dans le cas de la platesforme de développement utilisée, Windows XP, le réacteur par défaut est le `ACE_WFMO_Reactor`. Il est basé sur la fonction `WaitForMultipleObjects` et n'est évidemment disponible que pour Windows. Sur les autres plateformes, la fonction `Select` est la base des réacteurs. Le `ACE_WFMO_ Reactor`

offre certains avantages sur le `ACE_Select_Reactor` et le `ACE_TP_ Reactor`[4] qui ont motivé son choix. Par exemple, la fonction native `WaitForMultipleObjects` peut être accédé simultanément par plusieurs fils sur les mêmes descripteurs (*handles*) ce qui augmente ses capacités de parallélisme contrairement à `Select`. De plus, sur Windows, la fonction `select` ne peut que gérer des interfaces de connexion et des événements de compteurs contrairement à `WaitForMultipleObjects` qui peut gérer plusieurs autres types d'événements, ce qui apporte une plus grande flexibilité.

La partie la plus compliquée à implémenter dans ce patron fut celle de l'acceptor car la structure élaborée précédemment, à la figure [illegible], supposait que chaque gestionnaire aurait accès au pointeur de l'objet `ACE_Task` lui étant associé pour qu'il soit capable de lui remettre les messages après les avoir reçus via la connexion réseau. Pour être capable de créer l'objet `ACE_Svc_Handler` et lui remettre le pointeur via l'objet `ACE_Acceptor`, il a fallu créer deux nouvelles classes redéfinissant certaines opérations virtuelles du cadre d'applications ACE. La figure [illegible] présente les deux classes dans l'environnement du cadre d'applications ACE et les fonctions virtuelles surdéfinies pour atteindre la solution.

Le rôle de la classe `ToolkitAcceptor` est d'accepter une connexion et de créer un objet `InputHandler` pour gérer cette connexion. Le rôle de la classe `InputHandler` est de traiter les informations reçues par la connexion et de les transmettre, sous la forme d'un message, à l'objet `IthreadPool*` lui étant associé par l'entremise de la file de messages. L'utilisation du réacteur et du patron Acceptor-Connector forme, en quelque sorte, l'algorithme d'exécution de l'application principale. Le réacteur est une boucle infinie (ou presque) qui est basée sur une fonction qui démultiplexe les événements dans un système d'exploitation. Il faut donc appeler cette fonction dans une boucle pour être sûr de ne pas manquer d'événements générés par le système d'exploitation. Pour tester ce patron, une première implémentation locale a été effectuée. Voici un exemple de son utilisation :

```
IThreadPool* tp2 = new DEVSThreadPool() ;
ACE_INET_Addr portToListen (60002) ;
ToolkitAcceptor* acceptor = new ToolkitAcceptor(tp2) ;
acceptor->open(portToListen) ;
tp2->open(0) ;
ACE_Reactor::instance ()->run_reactor_event_loop () ;
ACE_Thread_Manager::instance ()->wait () ;
```

Ce court exemple démontre la facilité d'utilisation de la partie Acceptor du patron

[4] Plus de détails sur ces deux réacteurs peuvent être trouvés dans [SH03]

FIG. 8.5 – Structure UML™de la partie Acceptor de la communication réseau

de communication et de l'intégration du cadre d'applications ACE. Au départ, l'objet `DEVSThreadPool` est créé ainsi que l'adresse du port correspondant sur laquelle il recevra des informations. L'objet `ToolkitAcceptor` est instancié et initialisé avec le port réseau et le pointeur du `IThreadPool*`. Ensuite, la réserve de fils est initialisée, ce qui démarre le ou les fils. En dernier lieu, le singleton `ACE_Reactor` est appelé et la boucle démarre la démultiplexion des événements. La dernière ligne assure que l'application principale ne se terminera pas avant que tous les fils instanciés aient terminé leur travail et soient détruits. L'oubli de cette ligne causerait une fuite de mémoire dans l'application. La figure présente un diagramme de séquence qui explique l'exécution du patron de l'`Acceptor`.

Il est important de noter que la classe `InputHandler`, présentée à la figure , est aussi un fil[5] et que le traitement de la connecxion réseau par l'interface de connexion (l'objet `ACE_SOCK_Stream` dans le cadre d'applications ACE) se fait dans la fonction *svc* tout comme le fait la classe `ACE_Task`. Le fait d'utiliser d'autres fils augmente la possibilité de parallélisme et libère l'application principale pour sa tâche de démultiplexion d'événements.

La deuxième partie du patron est le Connector. Son design est relativement simple puisque ce n'est qu'une classe, mais les opérations effectuées avec cet objet sont relativement complexes. À l'inverse du `InputHandler`, il doit prendre un message de sortie du `DEVSThreadPool` et transférer les données sur le réseau, à l'intention d'un port en particulier, à l'aide des fonctions de base des interfaces de connexion comme `send` et `send_n` qui envoient un nombre fixe d'octets stockés dans une mémoire tampon (*buffer*). La complexité des opérations vient du fait que son implémentation repose sur des fonctions bas niveau difficiles à coder. La figure présente le diagramme de classe de l'objet `ClientConnector` et sa relation avec la classe `DEVSThreadPool`.

L'objet `ClientConnector` offre deux fonctions publiques qui permettent de se connecter à une interface de connexion, (`connect` et `reconnect`), et une fonction qui permet d'accéder à la connexion par l'entremise d'un descripteur, (`peer`). Présentement, le travail de décomposition du message et de transfert sur le réseau est situé dans la classe `DEVSThreadPool`, mais il pourrait être nécessaire de le transférer à la classe du `ClientConnector` pour une meilleure encapsulation des comportements.

Pour faciliter l'implémentation du système de communication sur le réseau, l'utilisation d'une astuce est nécessaire. Comme tous les messages ont la même structure, il est important de toujours envoyer tous les champs (dans le même ordre) pour garder un

[5]Initialisé avec le paramètre THR_JOINABLE, qui requiert l'appel au `ACE_Thread_ Manager` et à sa fonction `wait` pour éviter les fuites de mémoire.

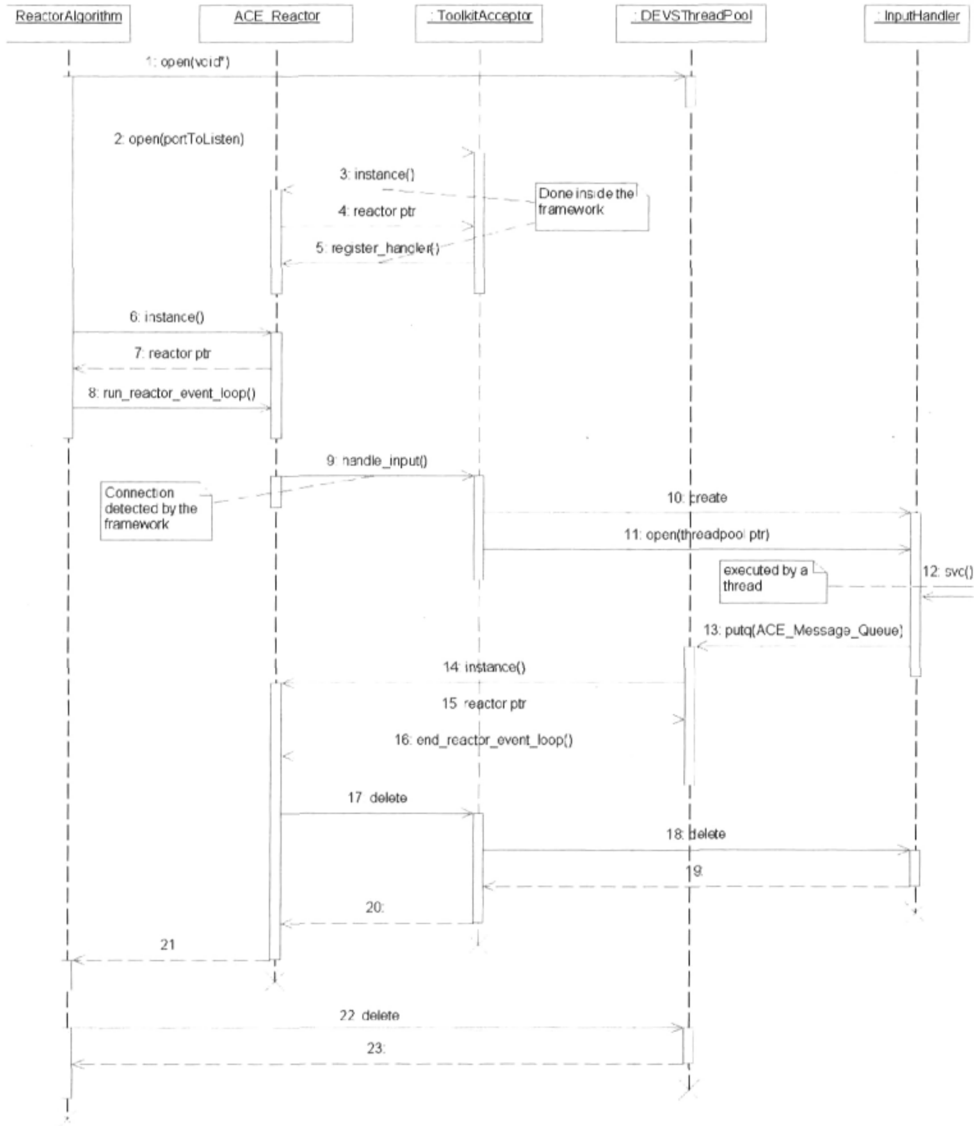

FIG. 8.6 – Diagramme de séquence de l'`Acceptor`

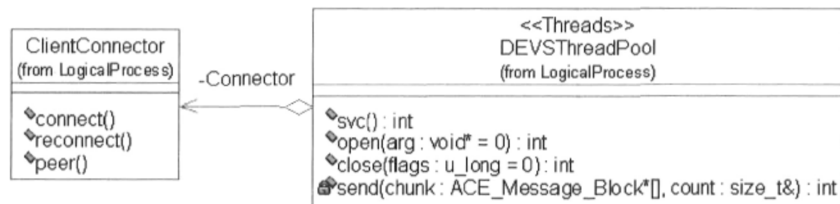

FIG. 8.7 – Structure UML™de la partie `Connector` de la communication réseau

ordre dans la réception des données. Si certains champs ne comportent pas de données, la chaîne de caractère (*string*) NULL doit être envoyée dans le but de simplifier la reconstruction des messages. Cette astuce peut cependant être pénalisante et engorger le réseau puisque les messages auront souvent une taille plus grande que leur besoin réel. La conversion des données est aussi un autre problème de taille, puisque, présentement les données ne sont pas toutes du même type (int, double, float). Ce problème reste en suspens puisqu'à ce jour, aucun test poussé n'a été effectué sur ce concept, seulement quelques chaînes de caractères ont été transférées pour valider la faisabilité du design.

8.4 Messages

Le cadre d'applications ACE propose une classe standard pour le transfert des messages, qui offre une compatibilité avec toutes les classes de communication que le cadre d'applications ACE redéfinit. Ces objets sont basés sur les classes `ACE_Message_Block` et `ACE_Data_Block`. La classe `ACE_Message_ Block` permet de transférer des chaînes de caractères, mais aussi des structures plus complexes et même des pointeurs entre les différents processus. Pour transférer des objets plus complexes, il faut spécialiser l'objet `ACE_Message_Block` pour obtenir plus d'efficacité. La manière la plus facile d'utiliser ces objets est de dériver une classe de `ACE_Message_Block` et d'y déterminer le nombre et le type de données à transmettre. La figure 8.8 montre la structure UML™nécessaire pour la spécialisation.

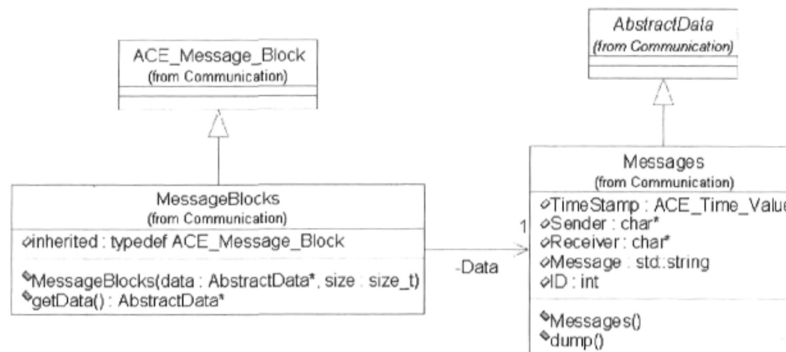

FIG. 8.8 – Structure UML™de la spécialisation de l'objet `ACE_Message_Block` et l'inclusion d'une structure complexe par le pointeur `AbstractData`

Il faut simplement créer une classe spécialisée, `MessageBlocks`, au lieu de garder la classe générique `ACE_Message_Block`, et y attacher une classe implémentant une structure plus complexe, `AbstractData` et `Messages`.

Procéder de cette manière peut être dangereux et il est important d'empêcher les opérations de copie de la classe spécialisée. En effet, si la copie d'un objet `MessageBlock` était possible ; le pointeur interne `AbstractData*` serait lui aussi copié et il pourrait arriver des cas où la destruction de deux objets `MessageBlock` détruirait plus d'une fois l'espace mémoire accédé par le même pointeur `AbstractData*`. En protégeant l'opérateur d'égalité et le constructeur par copie, ces opérations dangereuses peuvent être évitées.

```
Protected :
  MessageBlocks &operator= (const MessageBlocks &) ;
  MessageBlocks (const MessageBlocks &) ;
```

Bien que ce système de construction de messages fonctionne très bien dans l'environnement du simulateur local, il affiche un manque de flexibilité et n'est surtout pas compatible avec le formalisme DEVS. Dans la première itération du simulateur parallèle, ce système de messages n'a été créé que pour transporter les commandes venant de l'ordonnanceur vers l'application principale. Ces commandes varient entre $< RUN >$ et $< STOP >$ et doivent aussi transporter une estampille temporelle, la commande $< STOP >$ n'est envoyée que lorsque le temps de simulation final est atteint.

Le manque de flexibilité du design vient du fait que ces messages ne permettent pas de transférer de données, comme cela devrait être possible pour une simulation dans un environnement distribué. De plus, il faudrait aussi que ces messages soient compatibles avec le formalisme DEVS qui prévoit que quatre types de messages formels puissent être envoyés en plus des estimations de temps EOT et EIT qui doivent voyager entre les différents simulateurs dans un environnement distribué. Voici donc la liste de ce qu'il est nécessaire de transférer dans les deux environnements :

- Les messages d'initialisation : i-message
- Les messages de sortie : y-message
- Les événements externes : x-message
- Les événements internes : $*$-message
- Les messages de temps : EOT-message et EIT-message
- Les commandes diverses : c-message

Il est nécessaire de décomposer davantage les messages pour apercevoir toutes les données qui doivent être transférées, pour faire des regroupements et construire une seule classe satisfaisant les besoins des deux environnements.

Tout d'abord, les données sont ce qui est le plus difficile à regrouper, parce qu'un

modèle peut avoir une ou plusieurs sorties et que ces sorties ne sont pas nécessairement toutes du même type (double, float, int). Pour résoudre ce problème, il faut employer des outils déjà construits et utilisés dans le simulateur KARMA, les `DataTypes`[6] . Chaque variable d'état de chaque modèle possède un nom particulier. Cette caractéristique pourrait être utilisée pour stocker les données dans un container de la STL comme le dictionnaire en utilisant la classe de base des DataTypes pour le type des données : `std::map< DataTypes::String, DataTypes::Any >`. Il est à noter que dans l'environnement distribué, certains messages seulement ont besoin de transférer des données, car dans l'environnement multiprocesseurs, l'espace mémoire partagé évite de telles manipulations.

Deuxièmement, tous les messages doivent contenir une estampille temporelle. Le problème est que le cadre d'applications ACE utilise un type de données nommées `ACE_Time_Value` [SH02] différent de celui des modèles KARMA qui est le `Double` venant des `DataTypes`. L'utilisation d'un des deux types de données dans le contexte de l'autre est présentement impossible. Il a donc fallu créer une fonction permettant la conversion aux endroits souhaités. La conversion d'un `DataTypes::Double` à un `ACE_Time_Value` est facile, car il est possible d'obtenir le `double` original du `DataTypes` et par la suite construire l'objet `ACE_Time_Value` grâce à son constructeur acceptant un `double` comme argument. La conversion inverse est plus compliquée venant du fait que la structure `ACE_Time_Value` contient deux champs, celui des secondes avant la décimale et celui des microsecondes après la décimale (10^{-6}). Cette conversion peut s'effectuer comme suit :

```
//Conversion between ACE_Time_Value and DataTypes::Double
ACE_Time_Value tmp = message->getTimeStamp () ;
double timestamp = tmp.sec () ;
timestamp += tmp.usec () / (double)1000000 ;
```

Cette conversion n'a alors qu'à se situer dans le code du `ThreadPool` juste avant que le modèle KARMA ne soit invoqué. Pour le reste des échanges dans le simulateur, il est préférable de garder l'estampille temporelle en `ACE_Time_Value`.

Pour le reste de la majorité des données à transférer, elles peuvent l'être dans un format texte qui annoncera les commandes et le type de message envoyé. Pour ces besoins, il serait préférable d'utiliser une structure comme celle des `<DataTypes::String>` de l'architecture de KARMA pour améliorer la compatibilité.

[6]Les `DataTypes` sont un ensemble de classes de données bâties sur la STL qui permet d'avoir une classe de base pour garder différentes données dans un même conteneur. Les `Datatypes` redéfinissent certains types de données comme le int, double, vector, matrix, bool, char et string

Cependant, il reste toujours un problème à régler ; le message d'initialisation. Dans le simulateur parallèle multiprocesseurs, il est nécessaire de transférer les pointeurs sur les modèles KARMA BaseEntity d'après la liste construite dans le théâtre d'engagement. Cette liste vient du scénario XML qui est traité par un analyseur syntaxique (*parser*) XML et qui représente la simulation à exécuter. Le problème est que les modèles KARMA sont actuellement instanciés par l'application principale et non par chaque fil. Les pointeurs sont alors transférés vers chaque fil qui peut accéder aux objets puisque les fils et l'application principale partagent un espace mémoire commun. Présentement, les messages ont donc besoin d'un champ supplémentaire du type de données <`KARMA::Root`>, mais éventuellement, les scénarios devront pouvoir être séparés pour que le simulateur distribué soit fonctionnel. Ce champ devrait théoriquement pointer vers un fichier XML pour que chaque fil (processus) puisse aller y chercher le ou les modèles qu'il doit simuler et les instancier séparément. Il est à noter que tout ce processus est exécuté avant la simulation et ne ralentit donc pas le temps de simulation global.

La seule manière de construire un système de messages compatible avec les deux environnements est de regrouper tous les paramètres possibles et de n'utiliser que ceux qui sont nécessaires dans un cas particulier (en laissant les autres paramètres vides ou NULL). Si le besoin se fait sentir, d'autres paramètres pourraient être ajoutés, comme une identification de la provenance du message ou un identifiant unique à chaque message pour assurer la non-redondance des envois.

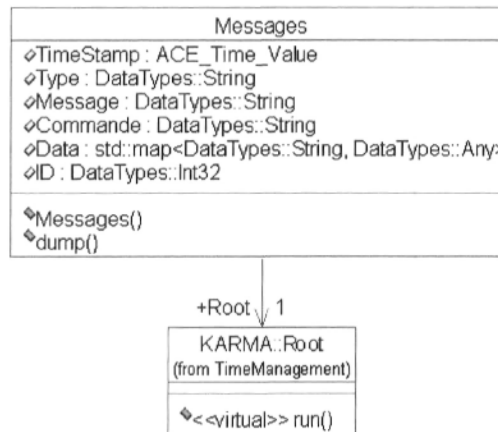

FIG. 8.9 – Design UML™de la classe `Messages`

Présentement, comme le montre la figure 8.9 qui présente la structure UML™du message convenant à tous les besoins décrits, la classe `Message` doit inclure une référence à la classe `KARMA::Root` pour transporter le pointeur du modèle. Ceci cause aussi

certains problèmes de dépendances circulaires entre les différents boîtiers (*packages*) de bibliothèques de liens dynamiques du simulateur parallèle. Ce problème peut entraîner des complications au niveau du code, mais surtout de la compilation. Sans être une nuisance, il serait nécessaire de régler ce problème par souci de perfection.

Éventuellement, la taille des messages pourrait poser un problème si trop de données ont besoin d'être transférées en même temps sur le réseau (par l'entremise des interfaces de connexion). Si ce problème survient, l'option d'envoyer une seule variable à la fois (tout en gardant la même structure de message) peut être envisagée.

8.5 Limites et optimisations

Tel que discuté précédemment, les différentes limites pouvant être rencontrées sont majoritairement imposées par les précédents choix de développement plutôt que par les différents logiciels utilisés ou le matériel employé pour exécuter les simulations.

La plus grande limite à l'extension des systèmes de communication vient du fait que les modèles KARMA ont été développés dans une optique où ils effectuent plus de travail qu'ils ne devraient le faire. Par souci d'encapsulation, le modèle ne devrait rien faire d'autre que calculer ses valeurs d'états, à l'aide de nouvelles entrées lui étant fournies, et de les retourner par sa seule fonction publique `run`. Comme chaque modèle est compilé séparément dans une bibliothèque de liens dynamiques, le fait d'y intégrer des pointeurs sur d'autres objets et d'y programmer des opérations supplémentaires comme la génération de ses événements devient une limite à la flexibilité que ce modèle peut éventuellement démontrer. Un bon exemple de cette limite est le fait que chaque modèle possède un pointeur sur l'ordonnanceur[7] et appelle directement une fonction publique de l'interface pour générer un nouvel événement et le faire ajouter dans la liste des événements. Un second exemple est, comme présenté à la section 5.2, le fait que chaque modèle utilise des fonctions héritées comme `getParameter` et `setParameter` pour accéder à ses variables d'état. Ces variables, par souci d'encapsulation, devraient être privées à chaque modèle et, au besoin, être transférées par le système de communication aux autres modèles nécessitant ces données.

Une limite technique venant du patron réacteur peut cependant être établie avec certitude. Les fonctions de démultiplexion des événements du réacteur sur les systèmes d'exploitations Windows et Unix ont des limites connues et documentées. Sur le sys-

[7]l'ordonnanceur est l'objet qui reçoit les événements, qui classe les événements dans une liste selon les priorités et qui ordonne les exécutions lorsque le temps de simulation l'exige.

tème d'exploitation Windows, la fonction `WaitForMultipleObjects` peut démultiplexer au maximum un tableau de 64 descripteurs par fil dont deux sont utilisés intérieurement par le réacteur. Chaque objet `ACE_ WFMO_Reactor` ne peut donc traiter que 62 descripteurs. Le système de communication distribué qui repose sur le patron Acceptor-Connector est donc limité à 62 objets ACE_Task par réacteur. Le nombre de fils ne signifie cependant pas un nombre équivalent de modèles, car présentement, le nombre de fils est égal aux nombres de modèles BaseEntity et non pas au nombre de modèles total dans la simulation (les modèles Parts sont intégrés dans le même fil que le modèle BaseEntity duquel ils sont composés). Une simulation distribuée pourrait donc, en résumé, traiter un total de 62 fils par ordinateur. Sur Unix, la fonction à la base du réacteur, `select`, n'est pas aussi limitée que celle utilisée sur Windows. Le nombre de descripteurs pouvant être traité par l'objet `ACE_Select_Reactor` est déterminé par la macro `FD_SETSIZE` dans le cadre d'applications ACE. Cette macro agit sur la structure `fd_set` du système d'exploitation et contrôle sa taille. La valeur par défaut de cette macro dans ACE est de 1024, il faut aussi noter que l'augmentation de cette valeur nécessite une recompilation du cadre d'applications. La portabilité du réacteur doit aussi être un sujet de discussion puisque ce patron repose sur deux fonctions très différentes sur les systèmes d'exploitation Windows et Unix. La première plateforme utilise un mécanisme nommé déclenchement par arête (*edge-triggered*) qui informe de l'état d'un descripteur uniquement lorsqu'un changement survient et la deuxième, un mécanisme nommé déclenchement par niveau *level-triggered* qui informe de l'état du descripteur à intervalles plus ou moins régulières. Deux systèmes aussi différents peuvent difficilement donner les mêmes résultats dans toutes les conditions même avec la meilleure interface. La littérature des développeurs et utilisateurs du cadre d'applications ACE est d'ailleurs explicite à ce sujet. Il ne faut donc pas s'attendre à avoir deux exécutions parfaitement identiques une fois le port complété, certains ajustements pourraient être nécessaires comme le mentionne Douglas C. Schmidt [SH03].

Les fonctions de bases du cadre d'applications ACE pour la transmission réseau utilisent le protocole TCP/IP pour obtenir des confirmations des envois. L'utilisation de ce protocole assure la réception des données puisqu'il est essentiel que les messages avec les nouvelles données se rendent à destination. Cependant, un tel protocole peut aussi surcharger les communications en transmettant trop de confirmations. Dans un tel cas, l'analyse d'un protocole fiable basé sur UDP pourrait être une solution possible.

Un dernier point à souligner est le manque de dynamisme dans la configuration de la topologie de communication. Présentement, comme une mise en correspondance est effectuée dans les fichiers XML entre les entrées, les sorties et les ports de communication réseau, une reconfiguration pendant l'exécution est interdite. Cependant, le design UML^{TM}possède les capacités qui permettraient une telle chose à l'avenir. L'objet

`ClientConnector` n'aurait besoin que d'une nouvelle adresse IP à laquelle se connecter pour changer la topologie de communication dynamiquement.

En résumé, le système de communication distribué est le point névralgique de la simulation distribuée, mais il peut aussi devenir le goulot d'étranglement du système entier. Il est donc très important que ce système soit fonctionnel et performant.

Chapitre 9

L'Application principale

Bien que les processus logiques utilisent des algorithmes complexes pour gérer indépendamment leur exécution dans plusieurs fils, ils ont aussi besoin d'un point d'attache initial pour les générer et paramétrer tous les détails de la simulation globale. Ce point d'attache, à partir duquel les scénarios sont analysés puis décomposés et les modèles sont initialisés puis distribués dans chaque fil, est l'application principale. Le but de l'application principale est de gérer tous les paramètres à l'initialisation de la simulation pour un environnement précis et ensuite de générer les fils, tout en suivant leur exécution.

Pour réussir cette tâche complexe, l'application principale doit réussir à s'amarrer à l'outil de simulation séquentielle KARMA et réutiliser le plus possible les éléments de lecture des scénarios. Ce travail est accompli dans une classe nommée `SimEnvironment`. L'application principale doit aussi analyser les modèles à sa manière, y retirer les informations pertinentes, initialiser les fils et les générer. Ce travail est effectué dans la classe nommée `Controller`. La classe *Scheduler* accomplit quant à elle le démarrage de l'algorithme choisi pour exécuter le scénario. De plus, l'application générale contient un ensemble de classes dont le but est de gérer les exceptions pouvant survenir à l'initialisation.

9.1 L'environnement

`SimEnvironment` est la classe appelée par le `main` avec les paramètres généraux de la simulation choisis par l'utilisateur. Ces paramètres sont la durée totale de la simulation

et le fichier contenant le scénario à simuler. D'autres paramètres sont présents, mais ne sont pas d'intérêt primordial pour le projet.

L'environnement réutilise une bonne partie du code et des classes développées dans KARMA pour le simulateur séquentiel dans le but de lire les fichiers de scénario d'engagement. Le résultat de l'analyse syntaxique (*parsing*) des fichiers XML est une liste de pointeurs des modèles à simuler. La liste inclut seulement les modèles BaseEntity, mais les modèles Parts sont accessibles par leur BaseEntity grâce à une stratégie de composition. À partir de cette liste, le pointeur de chaque BaseEntity est envoyé dans la classe `Controller` par l'entremise de la fonction `distributeRoot`.

Cette technique, qui tire profit de l'espace mémoire commun des fils, est utilisée lors de la construction du simulateur parallèle multiprocesseurs et devra être modifiée puisqu'elle n'est pas extensible dans le contexte d'une simulation distribuée. Pour atteindre ce but, l'environnement devrait lui-même redéfinir certains travaux d'analyse syntaxique des scénarios pour obtenir certaines informations dans une forme pouvant être distribuée. Par exemple, il faudrait que la classe `SimEnvironment` soit capable de transférer des parties du fichier principal du scénario (ou transférer les informations nécessaires pointant sur les bons fichiers de configuration présents sur tous les ordinateurs) sur chaque ordinateur relié pour que chaque fil soit informé du modèle qu'il doit instancier et de la composition. Chaque fil se chargerait alors de l'instanciation de l'espace mémoire des modèles dont il est responsable. Cette partie reste à faire puisqu'elle s'éloigne largement des concepts de simulation parallèle et distribuée.

La figure 9.1 présente le diagramme de classe UML™de la classe `SimEnvironment` et de ses liens avec KARMA et le simulateur parallèle.

Ce diagramme montre les liens avec les classes du simulateur séquentiel dont la référence concrète sur la classe `Factory` qui permet de construire l'objet désiré de la classe `Loader` qui permet la lecture des fichiers XML. Les autres liens sont faits à l'aide d'objets *singleton* comme le théâtre (`Theatre`) qui permet l'initialisation du terrain et des composants entourant la simulation des modèles. L'ensemble des classes d'exception est aussi utilisé pour traiter les erreurs pouvant survenir dans les classes de KARMA (les exceptions provenant du simulateur parallèle sont gérées dans le `Controller`). Le seul lien au simulateur parallèle est la classe `Controller` présentée dans la section 9.2.

Pour atteindre le but fixé d'obtenir une flexibilité accrue en permettant au simulateur de s'adapter à différents environnements en utilisant uniquement une configuration au temps d'exécution (*run-time*), les deux environnements, celui de KARMA et celui du simulateur parallèle, devraient être unifiés à partir d'un seul exécutable. Le `main`

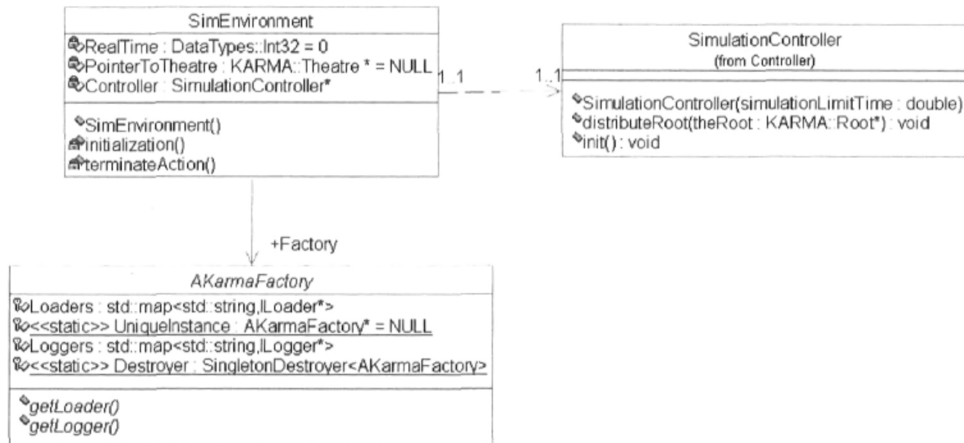

FIG. 9.1 – Diagramme de classe UML™de l'environnement montrant son lien avec le `Controller` et KARMA

devrait simplement filtrer la ligne de commande comme il fait présentement et, selon les arguments, appeler un des deux constructeurs. Une autre solution pourrait être une compilation conditionnelle à l'aide de drapeaux (*flags*), mais cette solution offre beaucoup moins de flexibilité que la première.

9.2 Le contrôleur

La classe `SimulationController` est le seul point d'entrée au simulateur parallèle. Elle agit un peu comme le patron de conception (*design pattern*) Façade pour offrir une interface unifiée de plusieurs sous-systèmes. Son rôle principal est de préparer l'exécution de tous les autres sous-systèmes et donc cette classe est responsable de la majorité de l'initialisation.

Comme le contrôleur est le seul point d'entrée, il est responsable de préparer les modèles et les fils, de générer le bon algorithme d'exécution pour les fils et de démarrer l'ordonnanceur avec le bon algorithme de gestion. La figure 9.3 définit les fonctions de la classe `SimulationController`.

Le constructeur avec argument est responsable d'initialiser certaines valeurs comme le temps limite de simulation. Cette valeur est présentement la seule qui permet de déterminer la fin de la simulation. Avant d'être transférée à l'ordonnanceur, la valeur du temps limite doit nécessiter une conversion de `double` à `ACE_Time_Value`. De plus,

FIG. 9.2 – Diagramme de classe UML™ du contrôleur et de l'ordonnanceur

public :	
`SimulationController (double simulationLimitTime)`	Constructeur qui prend le temps limite de simulation comme argument
`~SimulationController()`	Destructeur
`void init()`	Fonction qui débute l'initialisation et démarre la simulation
`void distributeRoot (KARMA::Root* theRoot)`	Fonction qui reçoit un pointeur sur un modèle KARMA comme argument et construit un message pour un fil en particulier.
Private :	
`void generateThreads()`	Fonction qui active les réserves de fils avec les paramètres voulus.

FIG. 9.3 – Fontions de la classe `SimulationController`

le constructeur initialise certains pointeurs comme celui de la fabrique présenté à la section 9.6) et celui de l'ordonnanceur présenté à la section 9.5.

La fonction `distributeRoot` reçoit un pointeur d'un modèle, analyse sa composition et y retire des informations nécessaires comme la période pour construire une liste d'objets `ThreadRecord`. Cette liste d'objets permet de relier chaque modèle (autant les *BaseEntity* que les *Parts*) au pointeur de la réserve de fils (*threadpool*) dans lequel il est initialisé par son numéro d'identification unique. Cette opération permet la communication entre les modèles et l'application principale.

La fonction `init` est le coeur du contrôleur. Elle initialise l'ordonnanceur, appelle la fonction `generateThreads()` et démarre la simulation en transférant la liste des objets `ThreadRecord` à l'ordonnanceur. Toutes ces opérations sont incluses dans un bloc de contrôle `try...catch()` qui permet de repérer les exceptions qui pourraient survenir lors de l'exécution de la simulation. Cette fonction, après avoir démarré la simulation en appel synchrone, se suspend pour assurer que l'application principale se détruise avant tous les fils générés. Cette dernière opération est nécessaire si les fuites de mémoire veulent être évitées. Lorsque tous les fils sont détruits et que l'application principale continue son exécution, un appel à un compteur sur l'ordinateur est fait pour déterminer le temps-réel total de la simulation.

La fonction `generateThreads` génère quant à elle le nombre nécessaire de fils contenu dans une liste créée dans la fonction `distributeRoot`. L'instanciation des fils est faite à l'aide de la fonction `activate()` et, présentement, ses paramètres sont fixés à l'avance (codés en dur).

9.3 L'ordonnanceur

La classe `Scheduler` n'est principalement qu'une classe enveloppante[1] (*wrapper*) autour du gestionnaire de fils du cadre d'applications de ACE, la classe `ACE_Thread_Manager`[2]. Le design initial de cette classe permettait de suspendre et de redémarrer les fils comme dans un système d'exploitation. Cependant, l'évolution des algorithmes dans les processus logiques rendent ces capacités inutiles.

La figure 9.1 présente le diagramme de classe UML™de la classe `Scheduler`. Cette

[1]Une classe enveloppante est définie comme une classe C++ qui contient un objet pour lequel la classe C++ offre une nouvelle interface

[2]Pour plus de détails voir [SH02], [SH03] et [HJS03]

figure montre le lien avec l'interface `ITimeManagement` qui utilise le patron de conception *Strategy* pour choisir l'algorithme de l'application principale au temps d'exécution. Ce choix est cependant fait au niveau de la classe `Controller` qui analyse les fichiers de scénario et le pointeur est par la suite passé par la fonction `init`.

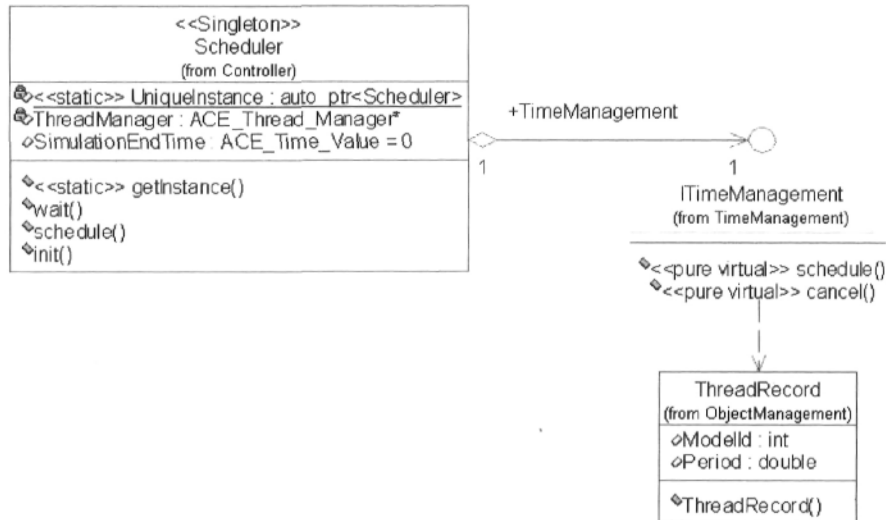

FIG. 9.4 – Diagramme de classe UML™du `Scheduler` et de son lien avec le boîtier `TimeManagement`

La figure 9.5 présente la liste des fonctions de la classe `Scheduler`. La protection du constructeur et la fonction `getInstance` sont des produits du patron de conception *Singleton* qui s'assure de la présence d'une seule instanciation d'une classe. La fonction `wait` est un bel exemple d'une classe enveloppante puisqu'elle n'est en fait qu'un appel direct à la fonction du même nom de la classe `ACE_Thread_Manager`. La fonction `init` est un sous-produit du *Singleton* puisque le passage d'arguments à l'aide du constructeur par argument est alors impossible. Cette fonction est donc la voie de contournement et doit nécessairement être appelée avant de démarrer la simulation par la fonction `schedule`. Cette fonction reçoit comme argument la liste des fils et des paramètres préalablement créés dans la classe `Controller` et ne fait que déléguer l'exécution à l'interface `ITimeManagement` et lui remettant le même argument.

En résumé, l'utilisation de la classe `Scheduler` se fait par la classe `Controller` qui, à partir de ses fonctions publiques, choisit l'algorithme de contrôle et démarre la simulation.

public :	
~Scheduler()	Destructeur
Scheduler* getInstance ()	Point d'entré du patron singleton
void wait()	Fonction relié au ACE_Thread_Manager et permettant l'arrêt de l'application principale
void schedule (std::vector<ThreadRecord*> threadList)	Fonction recevant comme argument la liste des fils. Elle démarre la simulation en invoquant l'objet TimeManagement
void init (ITimeManagement* timeManagement)	Permet l'initialisation du pointeur time-management. Nécessaire puisque le constructeur est protégé
Protected :	
Scheduler()	Constructeur protégé pour le patron singleton

FIG. 9.5 – Fonctions de la classe SimulationController

9.4 La gestion du temps

Le boîtier *TimeManagement* implémente l'algorithme de gestion de l'application principale. Cet algorithme, dans le cas de l'environnement local multiprocesseurs, envoie des commandes aux fils en attende d'exécution et, dans le cas de l'environnement distribué, commande l'activation de l'objet réacteur qui se charge de diriger les communications sur le réseau.

Ce boîtier a un design basé sur le patron de conception *Strategy*. La figure 9.6 présente ce design dans un diagramme de classe.

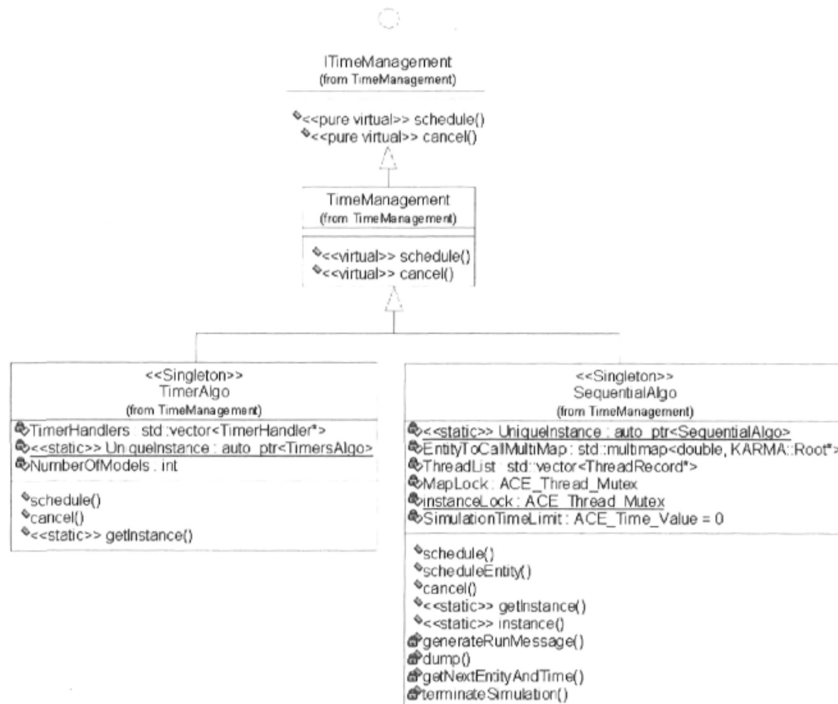

FIG. 9.6 – Diagramme de classe UML™du boîtier `TimeManagement`

L'interface `ITimeManagement` et la classe abstraite `TimeManagement` présente les deux fonctions publiques qui doivent absolument faire partie de chaque classe implémentant différents algorithmes. Les deux fonctions publiques sont présentées dans le tableau 9.7.

L'objet `ThreadRecord` qui est passé en argument, en compagnie du temps limite de simulation, doit contenir toutes les informations nécessaires au sujet de chaque fil pour que les deux algorithmes principaux puissent s'exécuter parfaitement. L'identi-

public :	
virtual void schedule (std::vector<ThreadRecord*> threadlist, ACE_Time_Value simulationTimeLimit)	Fonction démarrant l'algorithme. Elle reçoit comme argument la liste des fils et le temps limite d'exécution
virtual void cancel()	Fonction de cancellation de l'algorithme principal

FIG. 9.7 – Fontions de l'interface du `TimeManagement`

fiant unique de chaque modèle, le pointeur du fil dans lequel ils sont exécutés et le port de communication externe où les messages doivent être envoyés font partie de ces informations. La figure 9.8 présente le diagramme UML™de ce design.

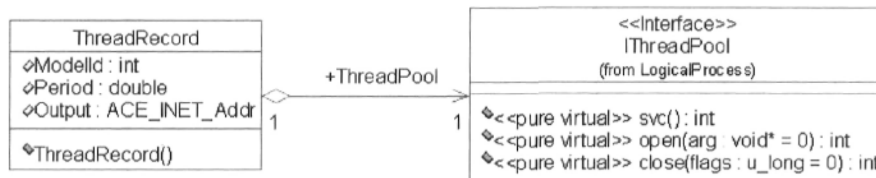

FIG. 9.8 – Diagramme de classe UML™de la classe `ThreadRecord`

L'algorithme principal employé dans l'environnement de distribution a déjà été expliqué en détail dans la section 8.3.2, celle de la communication distribuée . L'algorithme employé dans l'environnement local multiprocesseurs sera donc le seul à être présenté dans cette section.

La classe `SequentialAlgo`, qui implémente l'algorithme parallèle local, tire profit de l'espace mémoire commun des fils et son interface est donc plus détaillée et complexe que celle de la classe de base `ITimeManagement`. La figure 9.9 présente le diagramme de classe de cette interface.

Les fonctions

```
void scheduleEntity (
    double scheduleTime,
    KARMA::Root* entityToSchedule)
```

et

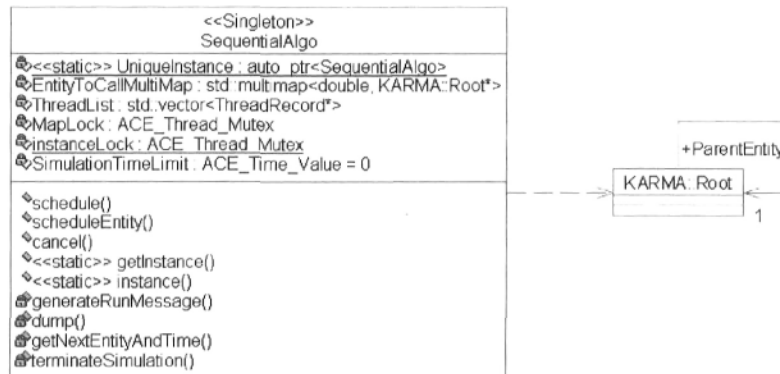

FIG. 9.9 – Diagramme de classe UML™de la classe `SequentialAlgo`

```
static SequentialAlgo* instance()
```

font partie d'une interface commune avec l'ordonnanceur du simulateur séquentiel de KARMA. Ces deux fonctions permettent la substitution de l'ordonnanceur parallèle dans l'architecture de KARMA. Cette approche est nécessaire pour que les modèles se rappellent dans la liste des événements futurs à l'aide de la fonction `scheduleEntity`, puisque chaque modèle possède une référence à l'ordonnanceur depuis la classe de base `Root`. Dans une telle situation, la classe `SequentialAlgo` se doit donc d'être un singleton puisque tous les modèles doivent posséder la même référence. Comme le montre le diagramme de classe, certains outils de synchronisation comme les mutex (`ACE_Thread_Mutex`), doivent être utilisés pour éviter des accès simultanés dans la classe `SequentialAlgo`.

L'algorithme implémenté dans cette classe est celui décrit à la figure 1.5. Une fois la fonction `schedule` appelée par la classe `Scheduler`, la boucle démarre et traite la liste des événements, implémentée dans la structure d'un dictionnaire supportant plusieurs entrées semblables (`std::multimap`) qui garde un pointeur sur chaque modèle instancié et le temps prévu de leur prochaine exécution, jusqu'à ce que le temps final de simulation soit atteint. La clef de classement dans la structure de données est évidemment l'estampille temporelle et est classée en ordre croissant. L'algorithme utilise aussi l'astuce de l'estampille temporelle cachée (classement prioritaire, voir section 7.1.5 pour déterminer l'ordre des événements simultanés.

La figure 9.10 présente le diagramme d'activité de l'algorithme contenu dans la fonction `schedule`. La boucle principale vérifie en premier lieu que le temps de simulation n'a pas atteint la limite dictée par l'utilisateur. Par la suite, il retire le premier élément de la liste des événements futurs. Si l'objet est valide, il change le temps de simulation

pour celui de l'événement retiré, il génère une commande d'exécution, détruit l'objet de la liste et recommence sa boucle. Si le temps de simulation limite a été atteint, la fonction `terminateApplication` envoie une commande de destruction à tous les fils instanciés.

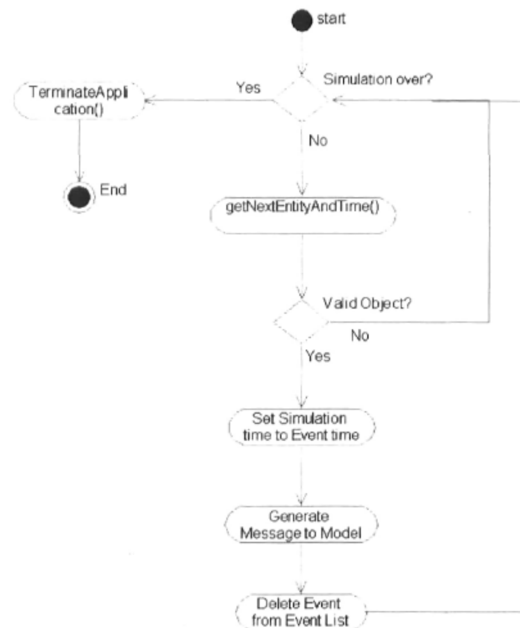

FIG. 9.10 – Diagramme d'activité de l'algorithme implémentée dans la classe `SequentialAlgo`

L'algorithme final implémenté dans la classe `SequentialAlgo` présente un ajustement dû au fait que certains scénarios de simulation KARMA ne génèrent pas une assez grande quantité d'événements. Une des conséquences d'une faible génération d'événements est de voir la liste des événements futurs se vider régulièrement et donc des appels à la fonction `getNextEntityAndTime` peuvent parfois retourner un pointeur NULL. Pour cette raison, un test supplémentaire sur l'élément retiré de la liste des événements doit avoir lieu avant l'utilisation du pointeur.

En résumé, l'algorithme parallèle principal doit suivre le pseudo-code de la figure [illegible], mais son implémentation doit aussi pouvoir s'adapter à des besoins spécifiques pour refléter les différentes situations de son application.

9.5 La gestion des exceptions

Un boîtier de base a aussi été développé pour gérer les exceptions qui pourraient survenir dans le simulateur parallèle et dans l'utilisation du cadre d'applications ACE. ACE ne gère pas les exceptions à cause d'un problème de portabilité, cette tâche est laissée au programmeur. ACE retourne plutôt la valeur 0 ou 1 dans la majorité de ses fonctions pour indiquer la réussite ou l'échec de l'appel en question. La figure 9.11 présente le diagramme de classe du boîtier `Exceptions`.

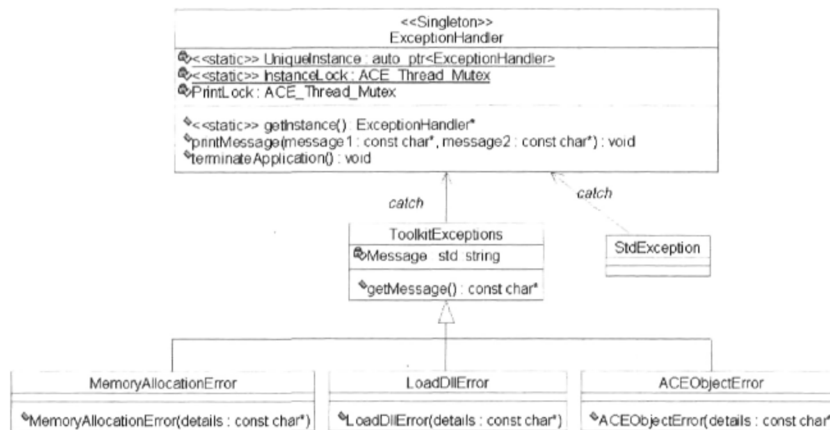

FIG. 9.11 – Diagramme de classe du boîtier `Exceptions`

Ce boîtier comprend une classe abstraite générale d'exceptions, la classe `ToolkitExceptions` et trois classes précisant des cas d'exceptions plus détaillés. Les classes `MemoryAllocationError`, `LoadDllError` et `ACEObjectError` représentent respectivement une erreur d'allocation de mémoire (dans le cas de l'instanciation des fils), une erreur de lecture d'une bibliothèque de liens dynamiques et une erreur lors de l'utilisation d'un objet dans le cadre d'applications ACE (comme les `ACE_Message_Queue`). La classe `ExceptionHandler` est un Singleton et est appelée lorsqu'une exception survient dans un bloc `catch(...)`, elle imprime l'erreur sur la console et ferme l'application. L'utilisation de ce boîtier est plus formelle qu'utile, car présentement toute exception levée entraîne l'arrêt immédiat de la simulation et la fin de l'application.

9.6 Design flexible

Un des objectifs de départ quant à la construction du simulateur parallèle était qu'il soit portable et flexible. Le besoin de portabilité venait du désir d'utiliser le simulateur

autant sur la plateforme de développement Windows que sur la plateforme *QNX*[3] pour diriger une simulation HWIL avec SEMAC. Le besoin de flexibilité venait du désir d'implémenter un simulateur capable de s'adapter à plusieurs environnements par une configuration au moment de l'exécution. L'adaptation aux différents environnements devait se faire par le changement des algorithmes de contrôle de la simulation.

L'objectif de portabilité a été atteint par l'utilisation du cadre d'applications ACE. Ce cadre d'applications peut présentement être porté sur une vingtaine de systèmes d'exploitation différents. ACE réussit ce tour de force en redéfinissant une couche logicielle d'adaptation sur chaque OS (*The ACE OS Adapter Layer*). Cette couche redéfinit les points généraux suivants

- Parallélisme et synchronisation
- Communication interprocessus (IPC) et mémoire partagée
- Mécanismes de démultiplexion d'événements
- Liaisons dynamiques
- Méchanismes d'accès aux fichiers

Cette redéfinition est possible grâce au patron de conception *Bridge* qui découple une abstraction de son implémentation pour que les deux puissent varier indépendamment [GHJV95]. La couche d'adaptation OS du cadre d'applications ACE exploite intensivement ce patron de conception pour permettre cette grande portabilité.

L'objectif de flexibilité a été atteint par l'utilisation du patron de conception *Strategy* qui permet de définir une famille d'algorithmes, d'encapsuler chacun d'entre eux et de les rendre interchangeables [GHJV95]. Ce patron de conception a été utilisé à deux endroits critiques : le design du boîtier `LogicalProcess` et le design du boîtier `TimeManagement`. L'utilisation de ce patron de conception à ces deux endroits précis permet l'interchangeabilité entre les algorithmes pour un environnement local multiprocesseurs ou pour un environnement distribué avec une communication réseau. La stratégie a aussi été couplée avec un autre patron de conception, *Factory Method*, qui permet à une usine de créer de multiples spécialisations d'un même produit général à partir d'un seul point d'entrée. La figure 9.12 présente ce design.

Deux des fonctions publiques de la classe `Factory` :

- `ITimeManagement* getTimeManagement (std::string name)`
- `IThreadPool* getThreadPool (std::string name)`

[3]QNX est un système d'exploitation temps-réel, plus de détails peuvent être trouvés à l'adresse http ://www.qnx.com/

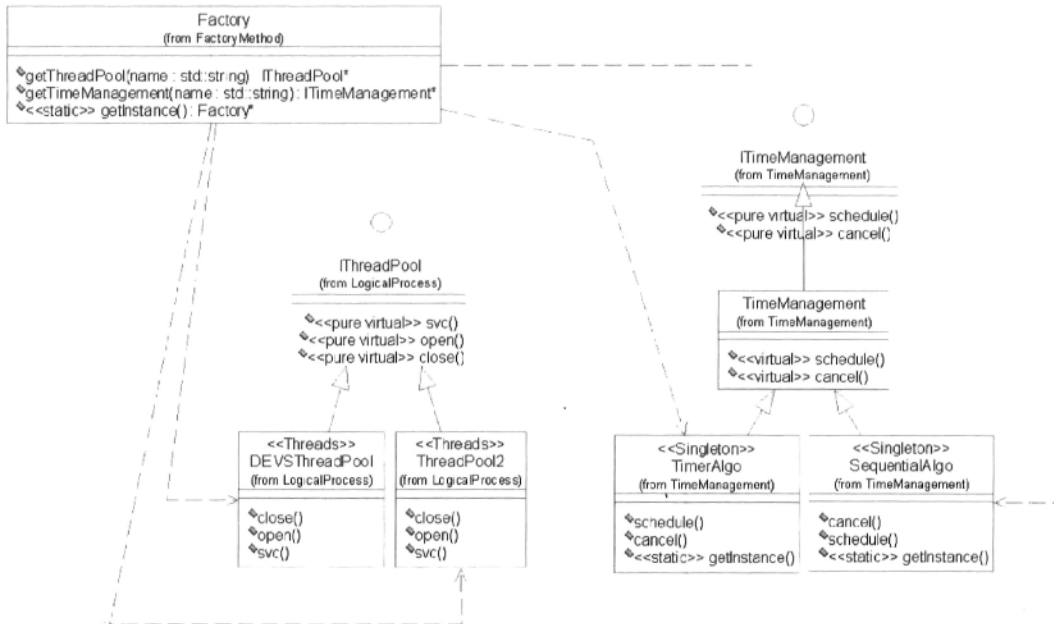

FIG. 9.12 – Diagramme de classe du patron de conception *Factory Method*

permettent d'instancier respectivement soit les classes `ReactorAlgo` ou `SequentialAlgo`, soit les classes `DEVSThreadPool` ou `ThreadPool2`. L'argument passé aux deux fonctions permet de faire la différence entre les produits retournés. L'utilisation des fonctions publiques de la classe `Factory` est très aisée et se fait comme suit :

```
IThreadPool* task = Factory->getThreadPool ("ThreadPool2");
ITimeManagement* algo = Factory->getTimeManagement ("Sequential");
```

Ces choix sont présentement codés en dur dans la classe de base du simulateur `Controller`, mais ils devraient normalement venir du scénario XML passé en paramètre à l'application.

9.7 Limites et optimisations

L'implémentation et le développement actuel du simulateur imposent certaines restrictions sur les scénarios pouvant être parallélisés.

La première limite est l'impossibilité d'utiliser des scénarios dynamiques. Les scénarios dynamiques sont appelés ainsi quand, au courant de leur exécution, les événements forcent l'instanciation de nouvelles entités (BaseEntity ou Parts) qui doivent jouer un rôle dans le reste de la simulation. Le problème qu'engendrent ces scénarios est double. Tout d'abord, le code pour instancier de nouveaux fils est situé dans la classe `Controller` et nécessite des appels aux fonctions `distributeRoot` et `generateThreads` pour fonctionner correctement. Le code qui gère l'exécution des événements est quant à lui encapsulé dans le boîtier `TimeManagement` et aucun lien entre les deux classes n'est présentement implémenté. La deuxième partie du problème est la manière dont les modèles KARMA génèrent ces nouveaux modèles. Présentement, le nouveau modèle est instancié à l'intérieur d'un autre modèle. Cette technique, bien qu'utile et aisée pour un simulateur séquentiel, devient impossible à implémenter dans un simulateur parallèle. La compatibilité future du simulateur parallèle aux scénarios dynamiques n'est cependant pas impossible puisqu'il est possible de rajouter des fils. La difficulté réside dans la procédure adoptée pour que la liste des `ThreadRecord` soit correctement ajustée et que l'algorithme de gestion des fils dans le boîtier `TimeManagement` reconnaisse ce besoin dynamique et engendre les nouveaux modèles. Pour contourner ce problème, il pourrait être nécessaire de déplacer la génération des fils plus près de l'algorithme de contrôle de l'application principale en fusionnant les classes `Controller` et `Scheduler`. De plus, une méthode efficace devra être trouvée pour qu'un modèle informe l'ordonnanceur qu'un nouveau modèle doit être instancié.

Un autre problème, déjà mentionné, est celui du point d'entrée unique (la fonction `distributeRoot`) qui nécessite un pointeur pour construire la liste des `ThreadRecord`. Pour être plus efficace, ce point d'entrée devra être modifié soit pour recevoir le fichier XML directement et en faire le traitement ou soit pour recevoir les informations des modèles déjà retirées du scénario. L'important est de construire correctement la liste d'informations des modèles et des fils pour que les liens de communication soient bien établis.

La classe `SequentialAlgo`, qui implémente l'algorithme de l'application principale, est un aspect du simulateur qui mérite quelques améliorations même s'il est présentement fonctionnel. Le fait que les modèles KARMA possèdent tous une référence à cet algorithme (par le patron de conception *Singleton*) et qu'ils appellent eux-mêmes la fonction `scheduleEntity` pour générer un nouvel événement dans la liste des événements pourrait engendrer certaines complications. Comme cet algorithme est une boucle sans fin, avoir des fils qui interrompent cette boucle par des appels asynchrones n'est pas la meilleure des solutions. L'implémentation d'un tampon de synchronisation pourrait être utile si des problèmes survenaient. Le tampon serait simplement la dissociation du code de la structure de la liste des événements de la classe `SequentialAlgo`.

Du côté des optimisations, les paramètres de génération des fils pourraient être extraits du fichier du scénario plutôt que d'être fixés à l'avance comme dans l'implémentation actuelle. Ce changement ne serait cependant profitable que dans un contexte de simulation temps-réel où il pourrait être nécessaire de faire varier les arguments propres à l'ordonnancement des processus par le système d'exploitation. Il est à noter que ACE ne tente pas d'harmoniser[4] ces paramètres et qu'il est du devoir du programmeur de s'assurer de la validité des arguments qui lui sont transmis.

[4]Il est impératif de s'assurer que les paramètres qui font varier la priorité et les classes de processus soient disponibles sur la plateforme en question.

Chapitre 10

Application du simulateur parallèle à l'architecture *KARMA*

Le projet d'étendre les capacités de l'architecture KARMA en y ajoutant des éléments qui lui permettraient de tirer profit d'un environnement multiprocesseurs était à la base un projet d'envergure. En cours de route, le désir d'inclure des capacités pour gérer un environnement distribué s'est aussi ajouté. À la fin du projet, le design UML™de toutes ces capacités futures a été effectué, mais leur implémentation reste encore incomplète. Le design entier du projet est décrit dans ce mémoire, à partir des analyses d'options, de l'élaboration des concepts et de leur évaluation jusqu'à l'implémentation de certains modules. Cette section a pour but d'indiquer quels modules du simulateur parallèle ont été implémentés, de montrer quels problèmes ont été rencontrés dans l'application du simulateur à l'architecture séquentielle KARMA et de donner les résultats tirés du simulateur parallèle avec les scénarios compatibles.

10.1 Avancement des travaux

Le travail effectué dans ce mémoire peut logiquement se diviser en deux parties. La première aborde le simulateur parallèle employé dans un environnement local multiprocesseurs et la deuxième porte sur le simulateur distribué employé dans un environnement réseau reliant plusieurs ordinateurs. Ces deux parties ne sont réellement qu'un seul logiciel qui tire profit des patrons de conception logicielle pour exécuter différents algorithmes lui conférant ainsi une flexibilité au temps d'exécution.

10.1.1 Simulateur parallèle

Le simulateur parallèle est la portion du projet sur laquelle le plus d'implémentations et de tests ont été effectués. Le design UML™de ce simulateur est complet et son implémentation l'est presque totalement. Ce simulateur repose principalement sur trois boîtiers : `LogicalProcess` qui gère l'algorithme d'exécution des fils, `TimeManagement` qui gère l'algorithme d'exécution de l'application principale et `Controller` qui gère tout ce qui concerne l'initialisation et les liens avec l'architecture KARMA. D'autres classes et boîtiers apportant un support à l'application principale sont décrits dans le chapitre 9.

L'algorithme principal de l'application, la classe `SequentialAlgo` a été longuement testée et modifiée au courant du projet. Elle a d'abord été développée et testée sur une plateforme monoprocesseur pour ensuite être rigoureusement testée une seconde fois sur la plateforme multiprocesseurs où l'algorithme a reçu les dernières modifications assurant sa fonctionnalité. La classe `ThreadPool2` comprise dans le boîtier `LogicalProcess` a subi les mêmes tests que l'algorithme principal et s'est avérée très fiable sur les deux plateformes. Présentement, l'algorithme testé est une version informelle des DEVS où les messages sont analysés et les commandes appliquées. L'application principale a quant à elle été modifiée à plusieurs reprises pour finalement bien se comporter dans son rôle de communication avec les différents fils.

Le simulateur parallèle a fait l'objet d'autant de tests unitaires que de tests globaux sur la validation des résultats rendus avec le scénario d'engagement de base comportant deux avions et un missile. Ce scénario est illustré à la figure 7.12). Les tests effectués sur la plateforme monoprocesseur se sont révélés concluants en affichant des résultats très semblables (plus de détails dans la section 11.1) pour les variables d'état des différents modèles faisant partie de la simulation. Sur la plateforme multiprocesseurs, les tests qui ont pu être effectués se sont limités à la simulation d'un scénario contenant seulement un modèle BaseEntity composé de Parts à cause de problèmes causés par corruption des données lors d'accès mémoire simultanés dans l'architecture KARMA.

Pour que le simulateur parallèle soit complètement fonctionnel, il reste à effectuer certains changements permanents dans l'architecture de KARMA concernant l'accès simultané aux données. Ces changements proposés sont peu intrusifs pour l'architecture originale et cette approche respecte un des objectifs secondaires qui était de minimiser les modifications à celle-ci. Plus de détails sur ces changements seront donnés dans la section 10.3.

10.1.2 Simulateur distribué

Le simulateur distribué est la portion du projet sur laquelle le moins d'implémentations et de tests globaux ont été effectués. Le design UML™de ce simulateur est complet, mais son implémentation n'a été complétée que pour certains boîtiers. Ce simulateur repose principalement sur quatre boîtiers : `LogicalProcess`, `TimeManagement`, `Controller` et `Network` qui gère les communications réseau à partir du patron de conception Acceptor-Connector.

L'algorithme principal de l'application distribuée, la classe `ReactorAlgo` qui implémente la partie Acceptor du design Acceptor-Connector, n'a été testé que de manière sommaire pour valider le concept de communication réseau et la démultiplexion des événements. Le design implémenté et testé représente l'algorithme démarrant le réacteur et activant un fil qui communique avec la classe `InputHandler` par l'entremise du réseau. `InputHandler` est la classe qui doit traiter les communications détectées par l'algorithme principal et le réacteur. Par la suite, la classe `InputHandler` est responsable de reconstruire le message et de le transmettre au fil concerné. Ce test a permis de confirmer la faisabilité du design, mais l'implémentation de la conversion de paramètres (*marshalling* et *demarshalling*)[1], qui représente la pierre angulaire de la réussite de la communication réseau, pourrait s'avérer complexe.

Le développement de la classe `DESVThreadPool` et de l'adaptateur `DEVSClassicSimulator`, qui sont responsables de l'exécution des modèles pour le simulateur distribué, a débuté, mais les algorithmes n'ont pas été implémentés. Ce travail ne devrait pas s'avérer ardu, mais une attention particulière devra être portée à l'implémentation de l'algorithme DEVS pour assurer un maximum de performance. La classe `ClientConnector` est implémentée entièrement, mais n'offre que des services de connexion et de reconnexion à un port réseau. La fonction de conversion de paramètres devrait éventuellement se retrouver dans cette classe pour une offrir une meilleure encapsulation des données.

En résumé, pour que le simulateur distribué soit fonctionnel, il reste à implémenter les fonctions de haut niveau pour l'application principale et pour les fils. Les fonctions bas niveau ont été implémentées et testées de manière unitaire, mais leur performance et leur efficacité n'ont pas été évaluées. Une autre tâche connexe qu'il ne faut pas oublier pour assurer la fonctionnalité du simulateur distribué est la décomposition du scénario principal et son transfert sur les ordinateurs distants. Une autre solution pourrait être l'exécution de plusieurs parties d'un scénario sur chaque ordinateur.

[1] Action de transformer les messages en données pour le réseau et l'inverse, respectivement.

10.2 Problèmes rencontrés

Dans tout projet d'envergure se cachent des problèmes imprévus. Dans le cas du simulateur parallèle, les problèmes ont commencé à se manifester lors des premiers tests sur la plateforme multiprocesseurs. Développer un logiciel parallèle sur un ordinateur comportant un seul processeur est possible puisque les systèmes d'exploitation peuvent exécuter plusieurs processus et fils d'une manière qui peut paraître parallèle en utilisant une technique de partage du processeur appelé découpage de temps (*time slicing*). Cependant, cette technique n'exécute réellement qu'un seul processus à la fois et ne permet pas une analyse exhaustive des accès mémoires, car ils ne sont pas réellement simultanés.

Dans le développement du simulateur parallèle, la majorité des implémentations l'ont été sur un ordinateur ne possédant qu'un seul processeur. Quand les tests ont commencé sur la plateforme multiprocesseurs certains problèmes imprévus, ou plutôt non analysés auparavant, ont fait surface. Deux types de problèmes ont été découverts : les premiers pouvant se régler à court terme et les deuxièmes pouvant se régler à long terme, par rapport aux fonctionnalités du simulateur parallèle.

Le premier type de problèmes porte sur l'exécution générale du simulateur parallèle. Ces problèmes, tout comme les failles dans l'exécution de l'algorithme principal (discutés dans la section 9.4), n'avaient pu être identifiés sur l'ordinateur mono-processeur puisque l'exécution n'était pas réellement simultanée. Ce type de problèmes, découvert sur la plateforme multiprocesseurs, peut cependant être facilement corrigé en modifiant légèrement les algorithmes.

D'autres problèmes plus importants ne relevant pas du simulateur parallèle, mais bien de l'architecture même de KARMA, ne peuvent pas être facilement résolus puisqu'ils impliquent des changements de base qui devront faire l'objet de discussions. Le premier de ces problèmes est la sécurité du traitement multifil du patron de conception *Singleton*. Ce patron qui assure une seule et même référence à un objet dans toute l'architecture est utilisé en abondance dans l'architecture de KARMA. Ce patron implique cependant que, dans un logiciel parallèle sur une plateforme multiprocesseurs, plusieurs processus ou fils pourront accéder au même espace mémoire et aux mêmes variables en même temps. Bien entendu la conséquence majeure est la corruption des données et des espaces mémoires qui résulte en problèmes au temps d'exécution. Ce problème est complexe à isoler, car il ne survient pas à toutes les exécutions puisque les accès simultanés ne sont pas toujours reproductibles. Le résultat de ce problème est que, de temps à autre, l'exécution du simulateur parallèle se termine abruptement et, d'autres

fois, tout se déroule comme prévu puisque ce problème est aléatoire. Ce problème limite présentement l'achèvement du simulateur parallèle pour des scénarios d'engagement formés de deux modèles identiques, à ce moment les accès simultanés se multiplient et le simulateur parallèle ne termine pratiquement jamais son exécution. La solution pour régler ce problème est l'implémentation d'une sécurité pour les traitements multifils dans tous les patrons de conception *Singleton*.

Un autre problème rencontré vient du fait que, dans KARMA, les variables d'état d'un modèle ne sont pas causales , c'est-à-dire qu'elles ne sont pas reliées au temps de simulation de leur création ou de leur changement. Ce problème survient lorsqu'à un temps donné, l'ordre de plusieurs événements parallèles affecte le résultat de la simulation finale. Le fait que le simulateur séquentiel de KARMA soit reproductible n'a pas permis de déceler ce problème auparavant. Les comparaisons faites entre le simulateur séquentiel et le simulateur parallèle sur un seul processeur ont fait apparaître ce problème. Le tableau 10.1 présente ce problème visuellement.

T_S	Exécution A	Exécution B
0.00	Aircraft x = 1	Aircraft x = 1
0.01	Aircraft x = x + 1	Missile y = x + 1
0.01	Missile y = x + 1	Aircraft x = x + 1
Résultat	y = 3	y = 2

TAB. 10.1 – Problème de la causalité des données

Si, dans un scénario quelconque, un avion et un missile s'exécutent au même temps de simulation; les deux exécutions devraient être parallèles. Cependant, puisque le missile utilise la position de sa cible pour se diriger (la position de l'avion), si l'avion s'exécute avant le missile (au même temps de simulation), le missile utilisera les dernières données de position de l'avion calculées au même temps de simulation et non pas ceux du temps de simulation précédent comme il le faudrait. Les modèles ne gardent qu'une seule copie de leurs variables d'état et la dernière valeur sera toujours utilisée même si elle ne le doit pas. Ce problème n'empêche pas le déroulement de la simulation, mais vient corrompre les résultats qui se mettent alors à diverger (plus de détails dans l'annexe B). Parfois les variables varient d'un centième de leur valeur, mais parfois seulement de l'ordre de l'unité.

La solution à ce problème est l'utilisation d'une mémoire tampon qui conserve les données de plus d'un temps de simulation. La taille de la mémoire tampon doit rester relativement petite, il n'est pas nécessaire de garder plus que les deux derniers temps de simulation, le temps présent et le temps précédent. Lorsque les modèles vont chercher leurs entrées, ils n'ont qu'à indexer le temps de simulation pour obtenir celui d'avant leur exécution présente. Le tableau 10.2 présente visuellement la solution au problème.

Temps	Exécution A	Exécution B
0.00	Aircraft $x_0 = 1$	Aircraft $x_0 = 1$
0.01	Aircraft $x_1 = x_0 + 1$	Missile $y = x_0 + 1$
0.01	Missile $y = x_0 + 1$	Aircraft $x_1 = x_0 + 1$
Résultat	$y = 2$	$y = 2$

TAB. 10.2 – Solution à la causalité des données

En résumé, le fait de ne pas lier les variables d'état avec un temps de simulation n'est pas un problème de taille une fois que l'on a découvert la source des divergences numériques, puisque le développement peut continuer tout en connaissant ce problème. Cependant, le problème de la sécurité *multihread* dans le coeur de l'architecture de KARMA a vraiment stoppé le développement du prototype du simulateur parallèle avec les modèles tirés de KARMA dans les dernières étapes du projet.

10.3 Les changements nécessaires à l'architecture de KARMA

Tout d'abord, le principal changement est l'un des deux discutés dans la section 10.2. Il est important d'appliquer une sécurité aux traitements multifils dans l'architecture de KARMA pour interdire les accès simultanés aux espaces partagés. Ce changement peut être apporté simplement en utilisant des outils de synchronisation comme les mutex ou les sémaphores. Ces outils de base peuvent cependant causer certains problèmes sournois comme des interblocages (*deadlocks*) s'ils ne sont pas relâchés correctement après leur utilisation. Pour résoudre ce problème, le cadre d'applications ACE propose un objet nommé `ACE_Guard` qui prévient les interblocages en se détruisant automatiquement quand il devient hors de portée (*out of scope*) de la fonction dans laquelle il a été créé. La sécurité des traitements multifils implémentée dans le simulateur parallèle fait amplement usage des `ACE_Guard` et leur utilisation est relativement simple. Tout d'abord, il suffit de déclarer un mutex dans les variables privées de la classe. Par la suite, il suffit d'invoquer l'objet `ACE_Guard` à l'entrée de toute fonction que l'on désire sécuriser. Voici un exemple :

```
ACE_Guard<ACE_Thread_Mutex> guard (this->_mutex);
```

Il est à noter que pour utiliser les gardes (*guards*) dans une fonction statique (comme dans le cas de `getInstance` du patron de conception *Singleton*), il faut aussi déclarer

le mutex comme étant statique. L'utilisation des gardes n'est cependant pas à toute épreuve et, dans certains cas, il peut être nécessaire de ne protéger qu'une variable commune dans une classe. Un exemple de variable commune à plusieurs fonctions est la liste des événements dans la classe `SequentialAlgo`. Il est impératif que cette structure ne soit pas modifiée par deux processus à la fois, même si la fonction `scheduleEntity` peut être appelée par tous les modèles. Dans un cas semblable, il peut être préférable de protéger l'accès à cette variable commune uniquement par un mutex que d'utiliser un garde pour toutes les fonctions où cette variable est utilisée. L'utilisation de cette technique est faite comme suit :

```
this->_mutex.acquire();
//accès à la structure
this->_mutex.release();
```

Le second changement nécessaire est celui de la causalité des données. La solution donnée auparavant, bien que triviale, assurera que les simulations séquentielles et parallèles soient reproductibles. L'intégration d'une mémoire tampon, un tableau statique contenant deux ou trois entrées, permettra de garder en mémoire les dernières valeurs de chaque variable d'état et donc de permettre véritablement aux événements parallèles d'être exécutés de manière simultanée. Il est à noter que comme la simulation distribuée transmet ses données par le biais du réseau, une estampille temporelle accompagne toujours chaque message. Ces algorithmes contournent donc le problème de la discrétisation des données.

Outre les deux problèmes discutés auparavant, l'architecture de KARMA pourrait bénéficier de certains changements et règles de conception qui pourraient éviter de développer des modules inopérants dans la simulation parallèle et distribuée. Tout d'abord, il serait nécessaire d'ajouter un tampon entre la classe `Root` et les deux ordonnanceurs (le séquentiel et le parallèle) afin de permettre leur interchangeabilité. Présentement, les deux classes : `SequentialScheduler` (simulateur séquentiel) et `SequentialAlgo` (simulateur parallèle) présente la même interface de base à chaque modèle, c'est-à-dire les fonctions `instance` et `scheduleEntity`, mais l'implémentation présente utilise un pointeur sur la classe `SequentialScheduler` définit dans Root.h et instancié dans Root.cpp par `instance`. Pour utiliser l'ordonnanceur dans l'architecture de KARMA, il a fallu changer le type du pointeur à celui de la classe, `SequentialAlgo`, dans Root.h. Ce changement implique cependant toute la recompilation des modules et des modèles qui incluent cette classe de base, c'est-à-dire la presque totalité des boîtiers de l'architecture et la totalité des modèles. Il est donc impératif d'implémenter une solution plus flexible et réutilisable que celle présente.

Une règle de conception qui doit être appliquée dans tous les développements de modèles pour l'architecture KARMA est qu'aucun modèle ne doit être responsable de générer autre chose que des sorties. Tous les nouveaux événements devraient être générés par l'adaptateur DEVS pour se conformer à la théorie formelle. Un exemple particulier de la non-application de cette règle est un modèle nommé *AircraftWaypoint*, représentant un avion pouvant lancer un missile (ou autre menace) sur sa cible. Dans certains cas, ce modèle peut initialiser lui-même un modèle de missile, interne à sa composition, pour le lancer sur sa cible. Ce genre de technique de programmation bien que possible dans le simulateur séquentiel est totalement interdit pour un simulateur parallèle pour plusieurs raisons. Premièrement, comme l'application principale n'a pas généré de nouveau fil pour ce modèle, celui-ci se retrouve dans le même fil que le modèle qui l'a appelé et aucune parallélisation n'est alors possible. Deuxièmenent, bien qu'il s'appelle automatiquement dans l'ordonnanceur parallèle (la classe `SequentialAlgo`), l'application principale ne sait pas dans quel fil il se trouve et ne peut donc pas lui envoyer de commandes ; ce modèle ne sera jamais exécuté. Il est très important de conserver la liste d'information sur les modèles (la liste de `ThreadRecord`) à jour et la seule manière cohérente de le faire est à partir de l'application principale. Ce problème fait partie d'un plus grand problème qu'est la simulation des scénarios dynamiques discutés dans la section 9.7. Une partie de la solution serait de faire générer un événement particulier à l'intention de l'application générale qui reconnaîtrait la demande de génération d'un modèle particulier et qui pourrait y répondre advenant que les capacités soient implémentées.

10.4 Environnements d'analyses et scénarios utilisés

Pour réussir à analyser les performances, la fiabilité et la validité des simulations engendrées par l'environnement parallèle, les travaux ont d'abord débuté sur une plateforme monoprocesseur pour se transporter par la suite sur une plateforme multiprocesseurs.

Il est important de spécifier que pour les analyses effectuées avec le prototype de l'environnement de simulation, les scénarios d'engagement d'armes utilisés ont dû être modifiés par rapport aux originaux pour faciliter le développement et l'analyse des résultats. Sur les deux plateformes, les scénarios n'intégraient pas le calcul des collisions entre entités, mais possédaient un module environnemental calculant la transmittance atmosphérique. Le retrait des collisions avait pour but de réduire le couplage entre les différentes entités participant à la simulation. Le module de calcul de la transmittance atmosphérique a été gardé pour assurer un temps de calcul raisonnable à chaque entité

pour chaque fils.

Les tests effectués, autant sur la plateforme monoprocesseur que multiprocesseurs, ont utilisé des scénarios contenant d'un à trois modèles de base avec plusieurs variantes de leur composition. La figure 7.12 montre le scénario de base contenant les trois entités et le déroulement de la simulation.

Sur la plateforme monoprocesseur (processeur Pentium M 1.4 GHz et 512 megs de RAM), les tests ont confirmé la validité globale des simulations. Outre les légères divergences numériques expliquées précédemment, les simulations les plus complexes exécutées contenaient simultanément jusqu'à trois modèles de base (BaseEntity) avec chacun une composition complète pouvant contenir jusqu'à une douzaine de modèles Parts. Plus de détails sur les modèles et leur composition peuvent être trouvés à l'appendice A. Plusieurs variantes de scénarios, à partir des trois modèles de base, ont été utilisées pour tester et valider le simulateur parallèle. Tous ces scénarios sont une variante du scénario contenant deux avions et un missile où le missile, lancé par un avion, prend en chasse le deuxième avion. Les tests ont varié entre simplement un modèle avec une composition réduite à une composition étoffée et deux à trois modèles avec une composition réduite à trois modèles avec une composition complète. Les différents scénarios utilisés ont permis de vérifier l'ordre d'exécution entre les différents modèles de base et l'ordre d'exécution entre les différents modèles formant la composition (d'un seul niveau) d'un modèle de base.

Sur la plateforme multiprocesseurs (deux Pentiums 4 et 1 gig de ram), les tests n'ont pas vraiment permis d'évaluer toute la puissance du simulateur parallèle à cause des problèmes de sécurité des traitements multifils dans l'architecture de KARMA. Ces problèmes ont forcé la fin des travaux de développement et de tests du simulateur vu le peu de temps restant au projet et l'ampleur du problème à régler pour poursuivre les tests et analyses. Les scénarios qui ont pu être simulés dans cet environnement contenaient uniquement un modèle de base avec une composition non conflictuelle, i.e. ayant une composition n'utilisant pas de sous parties (Parts) s'exécutant aux mêmes temps de simulation. Les scénarios qui ont révélé les bogues (*bugs*) dans la simulation étaient ceux dans lesquels se trouvaient plus d'un modèle BaseEntity. Dans ces scénarios, plusieurs accès simultanés aux mêmes espaces mémoires s'effectuaient dans l'architecture KARMA puisque la majorité des modèles sont créés par héritage et accèdent donc aux mêmes bibliothèques de liens dynamiques.

Chapitre 11

Conclusion

Cette section présente et explique succinctement les résultats obtenus et se termine par une partie présentant les travaux futurs à explorer en plus du suivi qui devrait être fait sur le système de simulation parallèle et l'architecture de simulation KARMA.

11.1 Résultats obtenus

Les résultats obtenus et présentés dans cette section sont tirés des scénarios exécutés sur les deux plateformes disponibles.

Les premiers résultats fiables permettant une discussion viennent des tests réalisés sur l'ordinateur portable monoprocesseur. Bien que cet ordinateur ne soit pas le plus rapide, des comparaisons entre les deux environnements de simulation ont pu faire ressortir certaines informations intéressantes. Il faut cependant prendre en compte le peu d'échantillons qui ont été recueillis puisque l'exécution du simulateur parallèle sur l'ordinateur monoprocesseur n'était pas le but du présent projet, mais seulement utile à titre de comparaison. Ces tests ont cependant, permis de valider l'ordre d'exécution des entités dans les simulations et les résultats obtenus à la fin de la simulation.

Le tableau 11.1 présente l'exemple du temps d'exécution en moyenne d'une entité dans une simulation dans les deux environnements de simulation.

Ces résultats présentent un fait surprenant : le simulateur parallèle effectue la simulation du même scénario en moins de temps que le simulateur séquentiel de KARMA et

Simulateur	**Moyenne** (sec)	**Variance** (sec)
séquentiel	0.193	0.001
parallèle	0.130	0.001

TAB. 11.1 – Temps d'exécution comparatifs pour le modèle AircraftLauncher entre les deux simulateurs pour un scénario de 2 s

ce 30 % plus vite, en moyenne, pour ce modèle. Normalement, les résultats auraient dû être inversés. Pour une même simulation, le fait d'utiliser des fils pour simuler le modèle et d'utiliser une synchronisation aurait dû ralentir le temps total d'exécution du simulateur parallèle. La différence entre les deux simulateurs vient non pas de l'architecture KARMA, mais bien de la boucle principale qui dirige chaque simulateur. Comme le simulateur parallèle ne supporte pas les scénarios dynamiques, sa boucle d'exécution principale est exemptée de plusieurs tests programmés avec les structures *if-else*. Cet exemple démontre bien l'effet que la boucle principale peut avoir sur le temps total de simulation si elle n'est pas basée sur l'exécution des événements et combien il est très important qu'elle soit soigneusement optimisée pour assurer le plus grand temps de calcul possible pour les modèles. Bien que le tableau précédent ne présente que des résultats obtenus pour un seul modèle, la démonstration de la rapidité de l'environnement de simulation parallèle sur l'environnement séquentiel s'est révélée être une constante pour tous les modèles à une échelle plus ou moins grande, comme expliqué précédemment.

L'analyse tirée de l'exemple du modèle `AircraftLauncher` ne peut cependant pas s'appliquer en général à tous les modèles de KARMA et à tous les scénarios. Moins un modèle fait de calculs et plus sa période est grande donc plus l'effet de la boucle principale se fait sentir sur le temps total de la simulation. Il faut aussi prendre en compte que le calcul du temps total de simulation vient de deux environnements différents et donc que les endroits où les mesures sont prises ne sont peut-être pas tout à fait équivalents même si une similitude entre les deux simulateurs a tenté d'être établie. Cependant, le constat général dégagé, qui est que la boucle principale peut ralentir la simulation si elle n'est pas optimisée, est toujours valide.

Le tableau 11.2 présente les résultats obtenus sur la plateforme multiprocesseurs venant de l'exécution de plusieurs scénarios. Les données comparent l'exécution d'un scénario contenant un modèle et un deuxième scénario contenant deux modèles du même type que le premier scénario. L'analyse préliminaire des résultats mène à certaines observations de contre-performance qu'il n'a pas été possible d'analyser en profondeur vue le problème d'exécution récurrente des scénarios à deux modèles et du manque de temps. Ces résultats montrent globalement que les scénarios contenant deux modèles du

même type prennent plus du double du temps à s'exécuter que les scénarios contenant un seul modèle. Dans ce tableau, les données sont tirées d'un échantillon de dix exécutions de chaque scénario.

Modèle	**Moyenne** (sec)	**Variance** (sec)
AircraftLauncher	0.046	0.001
2 AircraftLauncher	0.185	0.005
Aircraft3DOF	0.088	0.001
2 Aircraft3DOF	0.192	0.002
Missile	0.376	0.001
2 Missiles	N/A	N/A

TAB. 11.2 – Temps d'exécution comparatifs venant du scénario exécuté sur l'ordinateur à deux processeurs pour une simulation de 2 s .

Il est à noter que, dans le tableau 11.2, la simulation du scénario avec deux missiles ne s'est jamais complétée correctement, donc aucune donnée n'y apparaît. L'exécution de ces scénarios n'a cependant pas permis de déterminer avec exactitude la puissance disponible du simulateur parallèle et du gain de temps entre une simulation séquentielle et parallèle puisque aucun scénario possédant deux modèles n'a pu être correctement simulé avec une reproductibilité fiable sur l'ordinateur possédant deux processeurs. Pour contourner les problèmes rencontrés d'accès mémoires simultanés, il aurait fallu soit développer un prototype vide d'héritage des modèles ou soit bloquer les modèles en synchronisant les accès mémoires. La première solution était peu intéressante, car elle n'aurait pas vraiment permis de valider les travaux de parallélisation en exécutant des modèles parfaitement adaptés et aurait demandé considérablement de temps pour reconstruire les librairies adéquatement. La deuxième solution était plus intéressante, car plus rapide et permettait aussi d'exécuter de vrais modèles confrontant l'environnement parallèle à la réalité de la simulation. La deuxième solution a été choisie, mais n'a pas été terminée.

Ces données préliminaires révèlent aussi principalement qu'aucun gain de temps n'est présentement réalisé avec une plateforme multiprocesseurs et montre même que l'exécution de deux modèles semblables est plus longue que deux fois le temps d'exécution d'un seul modèle. Un autre fait extrêmement surprenant est que même si le modèle `AircraftLauncher` s'exécute deux fois plus rapidement que le modèle `Aircraft3DOF`, les deux scénarios contenant deux modèles ont à peu près le même temps d'exécution. Ces résultats sont donc très surprenants et *a priori* décevants, mais il ne faut pas oublier que les scénarios doubles ne s'exécutent correctement qu'environ 10 à 20 % du temps.

Pour connaître les causes de ces résultats, il faudra pousser plus loin l'étude de

l'exécution des fils et de leur répartition sur les processeurs par le système d'exploitation pour pouvoir analyser correctement ce qui se passe durant la simulation. La première possibilité expliquant ce résultat est l'exécution des différents fils sur le même processeur par le système d'exploitation Windows^{TM}. Ce système d'exploitation distribue lui-même les fils d'exécution sur ses différents processeurs et il est très possible que sans une optimisation manuelle, l'environnement de simulation parallèle ne dévoile pas toute sa puissance de calcul.

11.2 Travaux futurs et suivi

Pour pouvoir faire un suivi sur le design de l'architecture parallèle et distribuée, les développeurs de KARMA auront à implémenter les modules qui sont incomplets, à modifier l'architecture de KARMA pour la rendre compatible aux accès de données simultanés et à analyser les résultats pour optimiser au mieux le parallélisme dans le simulateur. Pour ce faire, un résumé des points principaux à aborder est présenté. Ce résumé des travaux futurs n'est pas exhaustif et seulement certains points peuvent être choisis et implémentés selon les besoins de l'outil de simulation KARMA.

Pour que l'architecture de KARMA tire profit d'un formalisme de simulation discrète plus standard et rigoureux, l'implantation du formalisme DEVS décrit à la section 7.1.3 est indispensable. Ce changement pourra profiter aussi bien à la simulation séquentielle qu'à la simulation parallèle. Ce module contient l'adapteur des modèles DEVS discuté à la section 7.1.1 et les algorithmes de simulation conservatifs DEVS présentés à la section 7.1.5.

Pour que l'architecture de simulation de KARMA puisse produire des simulations parallèles utilisant les modèles déjà développés, l'accès simultané aux données devra être permis pour plus d'un fil. Ce problème évoqué dans la section 10.2 rend nécessaire l'ajout de mutex ou de sémaphores pour que les simulations parallèles soient possibles sur un ordinateur multiprocesseurs. L'utilisation des outils de synchronisation du cadre d'application ACE permettra de garder la flexibilité et la portabilité souhaitée dans l'outil de simulation KARMA.

Après avoir réglé le problème des accès simultanés aux données, une analyse de l'exécution des fils sur la plateforme multiprocesseurs devra être effectuée pour vérifier le niveau de parallélisme obtenu avec les algorithmes DEVS, présenté aux sections 7.1.3 et 7.1.5, et l'outil de simulation parallèle développé durant ce projet. Cette analyse permettra peut-être de déterminer qu'une disposition différente des modèles à l'intérieur

des fils serait plus optimale que la première.

En dernier lieu, le développement des capacités distribuées du simulateur pourra être accompli. Le design de ce module est décrit dans les sections 7.1.5 et 8.3. L'implantation de ce module revient principalement à établir la communication distribuée des diverses entités dans la simulation. Cependant, ce module est celui qui est le moins développé et demandera donc beaucoup d'efforts de la part des développeurs pour être complété.

Un développement itératif des capacités du simulateur est conseillé pour réduire au minimum les risques de complications de la programmation et permettre de vérifier à plus long terme, avec plusieurs scénarios, la stabilité de l'exécution de la simulation parallèle.

Voici, en terminant, la liste suggérée des tâches à effectuer :

- Implantation d'une sécurité pour les traitements multifils dans l'architecture de KARMA.
- Analyse de l'exécution des fils sur la plateforme multiprocesseurs.
- Implantation du formalisme DEVS.
- Développement du simulateur distribué.

Annexe A

Détails des scénarios utilisés

Cette annexe donne des détails plus spécifiques concernant les scénarios utilisés lors des tests sur les deux plates-formes. Par souci de simplicité, seul le scénario contenant le plus de modèles et de sous parties, celui qui a été testé et validé, sera présenté. Ce scénario comporte les trois modèles de base : le Missile, le Aircraft3DOF et le Aircraft-Waypoint. En général, les deux avions possèdent la même structure de composition dans toutes les simulations fonctionnelles.

- AircraftWaypoint & Aircraft3DOF
  - Sensor
  - ExpendableDispensor
  - Signature
  - ThreatSystem
  - Joystick
  - MAWS
  - ThreadSensor 1 à 4
  - Dircm
  - Structure
- Missile
  - Sensor
  - Seeker
  - Guidance
  - Autopilot
  - Propulsion
  - Control
  - Airframe

- Signature
- StructureMissile

Annexe B

Details sur la non-causalité de l'architecture KARMA

Le tableau B.1 montre les différences numériques qu'il est possible de constater pour les variables d'état entre une simulation séquentielle avec l'architecture de KARMA et le simulateur parallèle sur une plateforme monoprocesseur. Le tableau ne montre que les résultats de quatre variables d'état que l'architecture calcule pour le modèle du missile. Ce modèle est plus représentatif des différences numériques que les avions dans le scénario utilisé puisqu'il est le seul à utiliser les données d'un autre modèle pour calculer ses variables d'état. Les données sont tirées d'un scénario d'une durée de 7 s ayant une fin d'engagement, c'est-à-dire une situation qui permet de déterminer si l'engagement est terminé. Dans le cas d'un scénario avec le missile, la fin d'engagement survient si le missile touche sa cible. Cette situation est représentée par un concept représentant la distance finale (la plus courte) entre le missile et sa cible (*miss distance*). Pour une simulation de 7 s avec le scénario utilisé ; la distance finale du missile est de 4.472174m et la cible est considérée comme détruite.

Dans les deux tests effectués, la valeur finale de la variable *miss distance* est demeurée la même, ce qui confirme la validité globale de la simulation parallèle en comparaison avec la simulation séquentielle de l'architecture KARMA. Les données ont cependant subi certaines divergences qui représentent pour la majorité une variation en dessous de 1% de leur valeur réelle. Même si la différence numérique des données n'a pas d'effet sur l'issue de la simulation dans ce scénario, il est impératif que le problème de causalité soit réglé et que le simulateur parallèle offre une reproductibilité parfaite par rapport à l'architecture séquentielle de KARMA.

Exécution séquentielle avec l'architecture de KARMA

Variable d'état	X	Y	Z
Orientation	0.604312	0.004242	-13.447579
Acceleration Vector	-16.460963	8.517332	-2.978419
Velocity Vector	50.626904	351.100065	0.913612
World Location	3570.064652	1561.995763	-999.514808

Exécution séquentielle du simulateur parallèle

Variable d'état	X	Y	Z
Orientation	0.604840	0.004035	-13.438137
Acceleration Vector	-16.453277	8.454706	-3.081656
Velocity Vector	50.416866	351.362945	1,045972
World Location	3563.283259	1559.415794	-999.409024

TAB. B.1 – Problème de divergence numérique des données avec le missile

Bibliographie

[BBM99] Lokesh Bajaj, Rajive Bagrodia, and Richard Meyer. Case study : Parallelizing a sequential simulation model. *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 29 – 36, 1999.

[BL03] R. Brochu and R. Lestage. Three-degree-of-freedom (dof) missile trajectory simulation model and comparative study with a high fidelity 6dof model. Technical report, Defence R&D Canada - Valcartier, Technical Memorandum TM-2003-056, 2003.

[BPGH+99] Zeigler Bernard P., Ball George, Cho Hyup, Lee J.S., and Sarjoughian Hessam. Implementation of the devs formalism over the hla/rti : Problems and solutions. *SIW*, 1999.

[bR] RakNet by Rakkarsoft. http ://www.rakkarsoft.com/.

[Bry77] R.E. Bryant. Simulation of packet communication architecture computer systems. *MIT-LCS-TR-188, Massachusetts Institute of Technology*, 1977.

[CM79] K. M. Chandy and J. Misra. Distributed simulation : A case study in design and verification of distributes programs. *IEEE Transactions on Software Engineering*, SE-5(5) :440–452, 1979.

[Cor97] Georgia Tech Research Corporation. Fdk- federated simulations development kit, 1997.

[CW98] Aaron Cohen and Mike Woodring. *Win32 Multithreaded Programming.* O'REILLY, 1998.

[Fis95] Paul A. Fishwick. *Simulation Model Design and Execution : Building Digital Worlds.* Prentice Hall, 1995.

[Fuj97] R.M. Fujimoto. Zero lookahead and repeatability in the high level architecture. *Proceedings of the 1997 Spring Simulation Interoperability Workshop*, 1997.

[Fuj00] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* John Wiley & Sons, 2000.

[Gey99] David W. Geyer. The use of multiple threads in an object-oriented real-time simulation. *American Institute of Aeronautics and Astronautics*, (AIAA-99-4338), 1999.

[GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Groa] Distributed Object Computing Group. http ://www.cs.wustl.edu/˜schmidt /ace.html.

[Grob] Distributed Object Computing Group. http ://www.cs.wustl.edu/˜schmidt /tao.html.

[Groc] Object Management Group. http ://www.uml.org/.

[Gro04] Object Management Group. Model driven architecture web site, retrieved from http ://www.omg.org/mda/, 2004.

[HGJ+04] N. Harrison, B. Gilbert, A. Jeffrey, M. Lauzon, and R. Lestage. *Adaptative and Modular M&S Configuration for Increased reusability.* I/ITSEC, Orlando, FL, USA, 2004.

[HGL+02] N. Harrison, B. Gilbert, M. Lauzon, A. Jeffrey, C. Lalancette, R. Lestage, and A. Morin. *A M&S Process to Achieve reusability and Interoperability.* Proceedings of the NATO RTO M&S Conference, RTO-MP-094, 11.1-11.18, 2002.

[HJS03] Stephen D. Huston, James CE Jonhson, and Umar Syyid. *The ACE Programmer's Guide.* Addison Wesley, 2003.

[Imp] MPICH Implementation. http ://www-unix.mcs.anl.gov/mpi/mpich1/.

[JD00] Kevin G. Jones and Samir R. Das. Parallel execution of a sequential network simulator. *Proceedings of the 2000 Winter Simulation Conference*, 2000.

[JJ03] Nutaro James Joseph. *Parallel Discrete Event Simulation With Application To Continuous Systems.* PhD thesis, The Universtiy Of Arizona, 2003.

[Lan] The Simula Programming Language. en.wikipedia.org/wiki/simula.

[Lub89] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple loop networks. *Comm. ACM*, 32(1) :111–131, 1989.

[Mac] Parallel Virtual Machine. http ://www.csm.ornl.gov/pvm/.

[Mad99] M. M. Madden. An overview of lasrs++. *AIAA Panel on Real-Time Software Architecture*, 1999.

[Mar99] Dave Marshall. Ipc : Sockets, retrieved from www.cs.cf.ac.uk/dave/ c/node28.html, Mai 1999.

[SSB01] Douglas Schimdt, Michael Stal, and Frank Buschman. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Network Objects*, volume 2. John Wiley and Sons, 2001.

[Sta] MPI Standard. http ://www-unix.mcs.anl.gov/mpi/.

[MD96] Defense Modeling and Simulation Office (DMSO)n. High level architecture, 1996.

[Mis86] J. Misra. Distributed discrete event simulation. *ACM Computing Surveys*, pages 39–65, 1986.

[Nic91] D. Nicols. Performance bounds on parallel self initiated discrete event simulations. *ACM Transaction on Modeling and Computer Simulation*, pages 24–50, 1991.

[OMG95] OMG. Common object request broker archictecture, July 1995.

[Pag98] E.H. Page. Zero lookahead in a distributed time-stepped simulation. *The MITRE Corporation*, 1998.

[PoAP] Branch Publications, American Institute of Aeronautics, and Astronautics Papers. http ://simulators.larc.nasa.gov/aiaa/aiaapapers.html.

[RMMB04] Stephen V. Rice, Ana Marjanski, Harry M. Markowitz, and Stephen M. Bailey. Object-oriented simscript. *Proceedings of the 37th Annual Simulation Symposium (ANSS'04)*, 2004.

[SH02] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming : Mastering Complexity Using ACE and Patterns*, volume 1. Addison Wesley, 2002.

[SH03] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming : Systematic Reuse with ACE and Frameworks*, volume 2. Addison Wesley, 2003.

[SHP97] Douglas C. Schmidt, Timothy H. Harrison, and Nat Pryce. Thread-specific storage for c/c++, an object behavioral pattern for accessing per-thread state efficiently. *4th annual Pattern Languages of Programming conference*, 1997.

[SKN00] Prasad Sushil K. and Junankar Nikhil. Parallelizing a sequential logic simulator using an optimistic framework based on a global parallel heap event queue : An experience and performance report. *Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pages 111 – 118, 2000.

[SRK01] Paul C. Sugden, Melissa A. Rau, and P. Sean Kennedy. Platform-independance and scheduling in a multi-threaded real-time simulation. *American Institute of Aeronautics and Astronatics*, (AIAA-2001-4244), 2001.

[SSB01] Douglas Schimdt, Michael Stal, and Frank Buschman. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Network Objects*, volume 2. John Wiley and Sons, 2001.

[Sta] MPI Standard. http ://www-unix.mcs.anl.gov/mpi/.

[Sta01] Ingolf Stahl. Gpss- 40 years of development. *Proceedings of the 2001 Winter Simulation Conference*, 2001.

[Sysa] Linux Operating System. www.linux.org.

[Sysb] Unix®System. www.unix.org.

[Syy] Umar Syyid. *A Tutorial Introduction to the ADAPTIVE Communication Environment (ACE).* HUGHES NETWORK SYSTEMS (HNS).

[Tec] OPNET Technologies. http ://www.opnet.com.

[TF93] Jya-Jang Tsai and Richard M. Fujimoto. Automatic parallelization of discrete event simulation program. *Proceedings of the 1993 Winter Simulation Conference*, 1993.

[WFR01] Hao Wu, Richard M. Fujimoto, and George Riley. Experiences parralelizing a commercial network simulator. *Proceedings of the 2001 Winter Simulation Conference (WSC'01)*, 2001.

[YXBP03] Kwan Cho Young, Hu Xiaolin, and Zeigler Bernard P. The rtdevs/corba environment for simulation-based design of distributed real-time systems. *Simulation*, 79 :197–210, Issue 4, April 2003.

[ZPK00] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation Second Edition.* Academic Press, 2000.