THÈSE

présentée à

l'Université de Nice – Sophia Antipolis
U.F.R. Sciences

pour obtenir le titre de

DOCTEUR

mention : Informatique

PROSIT
Un environnement pour la programmation de simulations à événements discrets

par Günther SIEGEL

Soutenue le 29 Septembre 1997 devant la commission composée de :

MM.	Nhan	LE THANH	Président
	Jean	BEZIVIN	Rapporteur
	Bernard	DION	Rapporteur
	Giorgio	TONELLA	Rapporteur
	Denis	CAROMEL	Examinateur
	Alain	JEAN-MARIE	Examinateur
	Philippe	MUSSI	Directeur

Remerciements

Je tiens tout d'abord à remercier M. Nhan Le Thanh professeur à l'université de Nice–Sophia Antipolis d'avoir accepté d'être le président de mon jury malgré un emploi du temps très chargé.

Je tiens également à remercier Jean Bezivin, Bernard Dion et Giorgio Tonella pour l'intérêt qu'ils ont porté à mes recherches. Leurs remarques pertinentes m'ont permis de compléter et d'éclaircir de nombreux points importants de ce manuscrit. Je leur exprime ma reconnaissance pour leur disponibilité aussi bien lors des discussions formelles et informelles, que pour la lecture de la thèse et la rédaction du rapport.

Un grand merci à Alain Jean-Marie et Denis Caromel pour avoir accepté de faire partie du jury.

Mes remerciements les plus vifs vont tout particulièrement à Philippe Mussi qui a dirigé mes recherches. Sa disponibilité sans faille, ses nombreux conseils et surtout la confiance qu'il m'a toujours témoignée, m'ont permis de mener à bien cette thèse. Tout au long de ces trois années passées à l'INRIA, il m'a apporté un soutien constant.

Je tiens aussi à remercier mes collègues et amis des équipes Mistral et Sloop de l'INRIA, pour leur accueil chaleureux et la bonne ambiance de travail. Merci à Bruno Gaujal pour m'avoir supporté 3 ans dans son bureau. Ses remarques et commentaires sur mon travail ont toujours été des plus enrichissants. Merci aussi à Alain Jean-Marie pour ses collaborations fructueuses, sa grande disponiblité, et surtout pour avoir été un des premiers utilisateurs de PROSIT. Merci à Nathalie Furmento pour les nombreuses modifications apportées à SCHOONER. Et enfin merci à Ephie pour avoir toujours une solution aux problèmes administratifs et pour sa bonne humeur constante.

Je n'oublie pas mes parents et ma famille dont le soutien inconditionnel, malgré la distance qui nous sépare et mes trop rares séjours, m'a permis de mener à terme ces années d'études et de recherches.

Pour finir, je remercie Vanessa pour son affection et pour avoir toujours été à mes côtés. Sans sa patience et ses sacrifices, ce travail n'aurait pu aboutir. Je lui promet d'être enfin disponible et de passer mes week-ends à la maison.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

TABLE DES EXEMPLES

1. INTRODUCTION

1.1 Problématique

L'étude des systèmes complexes qui nous entourent ainsi que leur conception et leur dimensionnement font de plus en plus souvent appel aux techniques de la simulation par ordinateur. Dans le cadre des systèmes à événements discrets, ces techniques consistent à transcrire le modèle en un programme qui spécifie les actions associées aux événements. L'ordinateur permet ensuite, à moindre coût, de déterminer l'évolution temporelle du système. Mais malgré une certaine diffusion de ces techniques, la simulation par ordinateur tient souvent plus d'un *art* que d'un outil de base pour ingénieur. De nombreux facteurs expliquent cette situation.

Tout d'abord, la réalisation d'une simulation nécessite très souvent le codage de l'ensemble des composantes voire parfois celui du noyau de simulation lui-même. La réutilisation de code et de modèles est dans ce domaine réduite. Ce manque de réutilisation est d'autant plus pénalisant que la complexité des simulations tend à croître. Cette croissance provient à la fois de la taille et de la complexité intrinsèque des modèles, du niveau de détail désiré et des échelles de temps multiples. Il est donc essentiel que ce domaine important de la recherche et de l'industrie bénéficie des avancées récentes en génie logiciel et plus particulièrement en programmation objet.

Un second facteur important est celui de l'évolution très rapide des besoins des utilisateurs auxquels peu d'environnements répondent. De nos jours, la modélisation est hiérarchique et hétérogène ce qui permet de traiter les grands systèmes en utilisant le paradigme de modélisation le mieux adapté aux différentes composantes. La production industrielle à moindre coût et la standardisation amènent à utiliser des éléments déjà existants d'où la possibilité de tirer profit de bibliothèques de sous-modèles. De même, la simulation n'est plus vue comme un outil universel mais doit être intégrée aux ateliers de conception et de développement, et le simulateur couplé aux outils préexistants.

Un dernier point est le besoin de performances. Or les environnements de simulation disponibles sont très souvent peu performants et rarement optimisables par le programmeur. Dans le cadre de la simulation de très grands modèles, il faut faire appel aux techniques de la simulation distribuée. Mais peu d'environnements proposent un modèle de programmation et d'exécution supportant à la fois la simulation séquentielle et la simulation distribuée.

1.2 Grandes orientations

Le travail de recherche que nous présentons porte sur l'étude et la conception d'un environnement pour la programmation de simulations à événements discrets. Le contenu de cette thèse et plus particulièrement les résultats obtenus intéresseront en premier lieu les programmeurs d'environnements de simulation. Mais les utilisateurs finaux ainsi que les modélisateurs y trouveront aussi des éléments portant sur la modélisation de systèmes à événements discrets, la conception et l'exploitation de simulateurs. L'objectif principal de nos travaux consiste à fournir un *framework*, aussi appelé *canevas*, permettant le codage puis la simulation de toute classe de systèmes à événements discrets. Le framework, en plus des critères classiques de génie logiciel (robustesse, modularité, réutilisabilité, etc.), permettra de réaliser aussi bien des simulations séquentielles, que répliquées ou distribuées. Le parallélisme sera intégré de manière transparente et naturelle à l'environnement. Le passage d'une simulation séquentielle à une simulation distribuée se fera sans modification de la modélisation et avec le minimum de modification du code. Pour cela, nous faisons appel au paradigme objet et nous cherchons à intégrer le moteur de simulation au sein des mécanismes supportant l'approche objet.

Le premier aspect de notre contribution consiste en la définition d'un modèle de programmation. Ce modèle est basé sur la notion d'objets actifs interagissant dans le temps de simulation par le biais de demandes de service. Ce modèle possède une extension distribuée qui introduit la migration des objets actifs et diverses abstractions permettant au programmeur de spécifier et de contrôler la répartition spatiale de son modèle et sa synchronisation. Nous avons aussi cherché à ce que le modèle supporte naturellement et simplement la construction de simulations hiérarchiques et hétérogènes.

La mise en œuvre du modèle, nous a amené à utiliser les techniques réflexives (un système est *réflexif* s'il contient une structure appelée *auto-représentation* qui décrit la connaissance qu'il a de lui-même [Maes, 1987]). La seconde partie de notre travail porte donc sur l'utilisation et la mise en œuvre de la réflexivité dans le cadre de la simulation à événements discrets. Nous montrons l'intérêt de cette technique et comment, associée au concept d'objet actif, elle permet le développement et la réutilisation de classes de comportement. Un autre aspect important de nos propositions porte sur l'utilisation de ces deux techniques afin d'intégrer les modèles de programmation séquentielle et distribuée au sein d'une même plate-forme de développement.

Cette recherche s'est déroulé sous la direction de Philippe Mussi successivement au sein de l'équipe Mistral puis de l'équipe Sloop [1] de l'INRIA. L'objectif central du projet Sloop est de développer des méthodes et outils permettant l'utilisation efficace de machines multiprocesseurs pour la simulation de systèmes à événements discrets. La réalisation de cet objectif implique la poursuite de recherches de haut niveau dans les domaines des langages à objets parallèles, de la simulation et de l'algorithmique parallèle et distribuée. Nos travaux se situent donc au cœur des objectifs du projet.

1.3 Organisation du mémoire

Ce mémoire est structuré en un chapitre introductif, trois grandes parties et un chapitre présentant nos conclusions et perspectives.

1. Sloop est un projet commun avec le CNRS et l'Université de Nice-Sophia Antipolis (laboratoire I3S).

Partie 1

Cette première partie présente le cadre de nos recherches et l'état de l'art. Après avoir introduit au chapitre 2 les bases de la simulation à événements discrets, nous présentons au chapitre 3 un panorama des techniques de distribution des simulations. Nous insistons dans ce chapitre sur la distribution spatiale des modèles et sur les algorithmes de conservation de causalité associés. Le chapitre 4 est une synthèse des relations unissant la simulation, la programmation objet et la programmation répartie. Ces thèmes constituent les trois domaines d'étude de nos recherches. Le chapitre 5 détaille et motive les principaux objectifs de nos travaux.

Partie 2

La seconde partie décrit nos propositions en terme de modèles de conception, de programmation et d'exécution pour un environnement de simulation à événements discrets. Nous présentons au chapitre 6 la partie séquentielle de nos travaux et au chapitre 7 la composante distribuée. Le chapitre 8 applique le modèle présenté aux chapitres 6 et 7 en traitant les éléments propres à la réalisation d'une simulation. Ce chapitre propose aussi quelques aides à la construction de simulateurs. Le chapitre 9 complète les chapitres précédents en montrant comment le modèle supporte le développement de simulations modulaires et réutilisables, et comment il permet le développement de simulations mixtes et hiérarchiques. Le dernier chapitre de cette partie, le chapitre 10, est une étude comparative du modèle de programmation et de l'architecture Prosit avec d'autres paradigmes de programmation et certains environnements de simulation.

Partie 3

Cette dernière partie donne les grandes lignes de la mise en œuvre actuelle de l'environnement Prosit. Le chapitre 11 présente l'implémentation du framework séquentiel et le chapitre 12 celle du framework distribué. Nous présentons au chapitre 13 les librairies et outils développés autour du framework. Le chapitre 14 présente une évaluation qualitative et quantitative du prototype. L'étude quantitative comprend une évaluation des mécanismes de base puis l'évaluation des versions séquentielle et distribuée de la librairie files d'attente. Ce chapitre contient aussi un bilan et diverses techniques d'optimisation des simulations. Le dernier chapitre, le chapitre 15, présente deux applications significatives en cours de développement avec l'environnement Prosit. Ces travaux ont permis des collaborations très fructueuses qui ont contribué à la maturation de l'environnement Prosit. Les premiers résultats sont très positifs et montrent l'apport de l'environnement Prosit tant sur le plan de la puissance d'expression que sur celui de la facilité d'utilisation. Ils constituent une validation concrète de notre approche.

Première partie

CADRE DE NOS RECHERCHES ET ÉTAT DE L'ART

2. LA SIMULATION À ÉVÉNEMENTS DISCRETS

2.1 Définition d'un système à événements discrets

Dans le cadre de cette thèse, nous nous intéressons à l'étude des systèmes dynamiques discrets (par opposition aux systèmes dynamiques continus). Un système est dit discret, si les variables d'état le caractérisant sont discrètes. L'espace d'état du système est alors dénombrable ou fini, et les changements d'états se produisent d'une manière discrète dans le temps à des instants $(t_i)_{i \in \mathbb{N}}$. Un changement d'état du système est appelé un événement et a une durée nulle. La trajectoire du système est constante par partie, chaque niveau correspond à un état et la longueur d'un segment représente le temps de séjour dans cet état (voir Fig. 2.1).

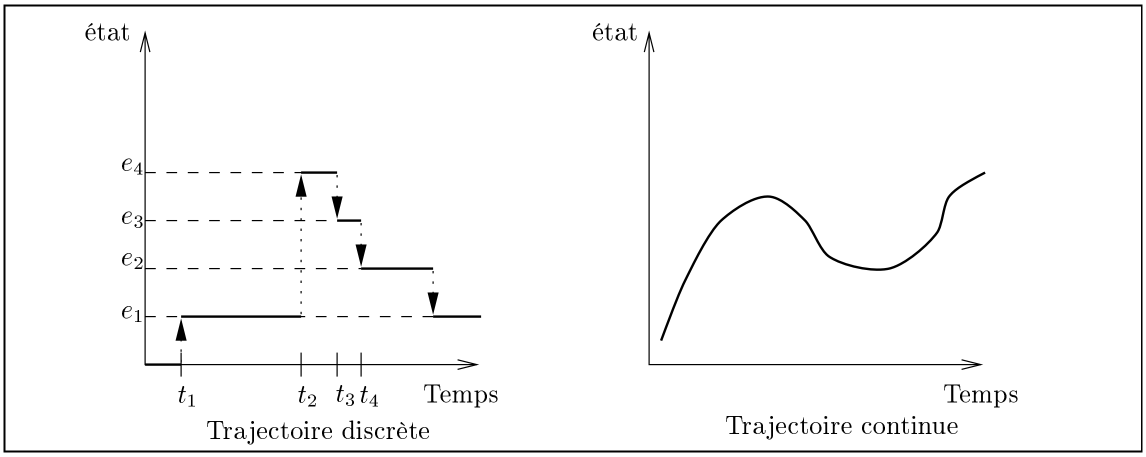


FIG. 2.1 – *Exemple de trajectoires*

L'étude de la dynamique d'un système, en vue par exemple de l'évaluation de ses performances, conduit à la notion de modèle et de processus de modélisation. Le modèle est une représentation de la réalité dans un autre formalisme (mathématique, logique, etc.) afin de pouvoir mieux étudier celle-ci. Le processus de modélisation consiste à exprimer, dans le formalisme retenu, les relations entre les paramètres et les variables d'état du système étudié.

De nombreuses techniques ont été mises au point pour l'étude temporelle des systèmes à événements discrets, nous pouvons par exemple citer : la logique temporelle, l'algèbre min-max, la théorie des files d'attente, la simulation, etc. Parmi ces approches, la simulation par ordinateur est une méthode d'expérimentation largement répandue dans plusieurs domaines de l'activité scientifique et technologique.

2.2 Simulation par ordinateur des systèmes à événements discrets

Dès l'origine, l'ordinateur s'est avéré être un outil de grande valeur pour la réalisation de simulations. En effet, une fois le modèle transcrit sous la forme d'un programme, l'ordinateur permet d'étudier l'évolution du système dans le temps. Cette étude se fait sans risque pour l'expérimentateur, à un coût réduit et lui permet de tester aisément ses hypothèses. De même, lors de l'étude de systèmes stochastiques, il est possible d'effectuer rapidement plusieurs simulations en faisant varier les paramètres. La simulation par ordinateur est donc une technique puissante. Elle est aussi irremplaçable lorsque l'étude analytique ou numérique échoue en raison de l'incapacité, soit de mettre le système en équations, soit de résoudre ces équations, dans le contexte actuel. Dans de très nombreux cas, l'approche consistant à simplifier le modèle jusqu'à obtenir une solution analytique n'est pas réaliste, car le modèle est alors trop éloigné du système réel.

Une fois la phase de modélisation effectuée, l'étape suivante consiste à construire le programme de simulation. Nous considérons dans cette thèse que la spécification algorithmique du comportement du système est toujours possible, et nous nous intéressons à la mise en œuvre du simulateur. Pour cela, trois aspects doivent être pris en compte : (1) la notion d'état, (2) la représentation et la gestion du temps, et enfin (3) l'évolution dynamique du modèle.

2.2.1 Représentation de l'état

L'état considéré est l'état physique du système, et sa définition est centrale à toute bonne modélisation. L'information de nature historique contenue dans l'état courant, doit permettre de déterminer l'ensemble des comportements futurs possibles. Le *noyau de simulation* gère aussi d'autres informations qui ne font pas partie du modèle mais permettent son étude. Ce sont essentiellement les données liées aux indicateurs de performances.

2.2.2 Gestion du temps

La nature dynamique du modèle nécessite la mise en place d'un mécanisme de représentation et de gestion du temps. Classiquement, le temps est représenté par une variable appelée l'*horloge de simulation*. Le noyau se charge de faire progresser cette horloge en maintenant un *échéancier* des actions futures non encore simulées. Deux techniques ont été proposées pour la progression de cette horloge : *progression par événements* et *progression à pas fixes*.

Dans la progression par événements, le noyau récupère de l'échéancier la date de la prochaine action à effectuer, avance l'horloge de simulation à cette date, et effectue ensuite les traitements correspondant à l'événement. La date de simulation progresse ici par sauts, par pas variables, en fonction des événements à traiter.

Dans la progression à pas fixes (aussi appelé *par horloge* ou encore *synchrone*), l'horloge de simulation progresse par pas constants δt. A l'étape i, de date $i\delta t$, le noyau sélectionne les événements qui auraient dû être simulés durant l'intervalle $](i-1)\delta t, i\delta t]$, considère qu'ils ont lieu à la date $i\delta t$ et les simule.

2.2.3 Gestion de la dynamique du modèle

L'implémentation directe de la notion d'événements conduit à structurer l'évolution du modèle par ces mêmes événements, on parle alors de *simulation orientée événements*. Dans cette approche, le programmeur doit identifier les événements caractéristiques et écrire pour chacun d'eux une routine réalisant les transformations associées des variables d'état. La simulation évolue dans le temps en exécutant les routines rattachées aux événements par dates croissantes. L'exécution d'une routine pouvant créer de nouveaux événements ou en détruire.

Une seconde approche consiste non plus à avoir une vision événementielle, en fragmentant le code en routines, mais à considérer le comportement des entités en interaction. On a alors une approche *orientée processus*, et un événement est vu au niveau logique comme "*l'activité*" d'un processus. Le code décrit l'évolution des entités à mesure qu'elles progressent dans le système, sous la forme de séquences ordonnées d'événements séparés par le passage du temps.

Signalons enfin une autre technique de simulation, appelée *simulation équationnelle*, qui est issue entre autres des travaux de Greenberg *et al.* [Greenberg *et al.*, 1990], de Cohen *et al.* [Cohen *et al.*, 1985] et de Baccelli *et al.* [Baccelli *et al.*, 1992]. Dans cette approche, l'évolution du modèle est décrite au moyen d'équations de récurrence. Ces équations permettent de calculer les dates auxquelles les événements ont lieu dans le système. Cette approche est différente des précédentes sur plusieurs points. Tout d'abord, il n'y a plus besoin d'échéancier. Ensuite, elle permet d'étudier la dynamique du réseau en utilisant simplement certaines opérations matricielles, apportant ainsi un gain très important en vitesse d'exécution (voir par exemple [Baccelli et Canales, 1993]).

2.3 La simulation séquentielle

2.3.1 Techniques de mise en œuvre

Dans le cadre séquentiel, la progression à pas fixes est rarement utilisée. Les deux principaux inconvénients de cette technique sont les erreurs introduites par le fait de simuler les événements à la fin de l'intervalle, et le fait de considérer les événements d'un intervalle comme simultanés. Pour pallier à ces inconvénients, il faut alors prendre δt très petit, ce qui se fait au détriment des performances. En effet, l'opération de recherche des événements à simuler est alors beaucoup plus fréquente et on risque d'avoir plus souvent des intervalles vides, sources de traitements inutiles. D'un autre côté, la simulation par horloge permet de réaliser très facilement les primitives du type "*attendre jusqu'à une condition*" (par exemple : attendre jusqu'à ce qu'un serveur soit inactif durant 10 unités de temps).

La progression par événements ne possède pas ces inconvénients. Le noyau se charge de faire progresser l'horloge en exécutant les actions dans l'ordre d'occurrence des événements. Le principe de causalité (le futur ne peut influencer le passé) est donc garanti.

La simulation orientée événement est probablement la plus facile à mettre en œuvre. Le noyau doit fournir une structure de données représentant un événement et diverses primitives de gestion de l'échéancier (insertion/retrait d'événements, etc.). Un événement est classiquement représenté par une date d'occurrence et un pointeur sur la fonction à exécuter. Lorsqu'un événement arrive en tête de l'échéancier, le noyau avance l'horloge de simulation à la date de l'événement et exécute la fonction associée.

Pour la simulation orientée processus, le noyau de synchronisation doit fournir une structure de données et une structure de contrôle pour la représentation des processus. Comme chaque processus évolue de manière concurrente dans le temps simulé, il ne peut plus être décrit par un sous-programme. On fait alors appel aux mécanismes de co-routines (voir [Knuth, 1973] p. 190–196) ou de processus légers [Birrel, 1989] qui leur assurent une exécution quasi-parallèle. Les interactions entre processus se font en général par le biais de sémaphores et de moniteurs. Le noyau fournit diverses primitives de gestion : création/destruction de processus, suspension pour une durée, etc.

2.3.2 Limitations de la simulation séquentielle

Les ressources informatiques requises pour l'exécution d'une simulation n'ont cessé de croître à mesure que se complexifiaient les modèles. Les principales limitations d'une étude séquentielle proviennent donc des temps prohibitifs d'exécution et de la grande taille des modèles. L'approche la plus utilisée consiste alors à faire appel aux techniques de l'informatique répartie, en utilisant plusieurs ordinateurs.

3. LA SIMULATION DISTRIBUÉE DE SYSTÈMES À ÉVÉNEMENTS DISCRETS

3.1 Techniques de distribution

La distribution d'une simulation peut se faire à plusieurs niveaux et selon différents critères. Nous présentons dans cette section six techniques possibles. Une septième consisterait à regrouper au sein d'un simulateur deux ou plusieurs de ces techniques. Nous ne nous intéressons pas à la parallélisation de simulations qui fait appel aux techniques de parallélisation automatique et aux calculateurs vectoriels. Cette approche est rarement utilisée, car elle demande une organisation particulière du code et ne donne de bons résultats, en terme d'accélération, que pour certaines classes de modèles (voir par exemple [Browne et Chandak, 1983]). Les principales caractéristiques des techniques de distribution sont résumées à la table 3.1.

Distribution de l'application Dans cette approche, on exécute sur chaque ordinateur une instance de la simulation avec des paramètres différents, on parle alors de *simulation répliquée parallèle*. Du point de vue du développement (programmation), c'est l'alternative la plus simple, puisqu'elle correspond à des exécutions séquentielles réalisées en parallèle et sans interactions. D'un autre côté, cette approche n'est utilisable que lorsque le programme de simulation n'est pas trop volumineux pour la machine, et quand les temps d'exécution d'une simulation sont acceptables. L'approche n'est pas non plus appropriée dans le cas où les résultats d'une simulation doivent être utilisés comme paramètre pour la suivante. La simulation répliquée est souvent utilisée dans les problèmes d'optimisation et pour améliorer la qualité des résultats statistiques (voir notamment les travaux d'Heidelberger [Heidelberger, 1986, Heidelberger, 1988]).

Distribution fonctionnelle La distribution fonctionnelle consiste à organiser le noyau de simulation en sous-tâches et à distribuer leurs exécutions sur différentes machines (voir par exemple [Sheppard *et al.*, 1985], [Comfort et Gopal, 1988], [Righter et Walrand, 1989]). Il est par exemple possible de déporter la gestion des statistiques, l'animation ou encore la génération des nombres aléatoires. Dans la mesure où il y a relativement peu de sous-tâches, le nombre de machines utilisables, et donc l'accélération possible, se trouve limité.

Distribution des événements Les deux approches précédentes n'exploitent pas le parallélisme intrinsèque au modèle. En effet, la réalité modélisée est souvent de nature parallèle (dans une "grande surface", il y a en parallèle des clients qui entrent, des clients qui choisissent des articles, etc.), et de nombreux événements peuvent donc être traités simultanément. Ce parallélisme peut être exploité en distribuant l'exécution des événements (voir par exemple [Jones *et al.*, 1989], [Righter et Walrand, 1989]). Le noyau de simulation, situé sur le processeur maître, sélectionne les événements à exécuter (ce sont généralement les événements simultanés) et les distribue aux processeurs esclaves. Cette approche est appropriée pour les machines multiprocesseurs à mémoire partagée. L'exécution concurrente n'est permise que s'il n'y a aucun risque de violer la causalité. Pour cela, le noyau de simulation doit posséder des informations supplémentaires sur la structure des événements. Cette connaissance supplémentaire est difficile à généraliser et à automatiser. De plus, une accélération n'est envisageable que si la granularité des événements est forte, c'est-à-dire s'il faut beaucoup de temps de calcul pour simuler chaque événement.

Distribution temporelle Supposons que l'on veuille simuler le système sur l'intervalle de temps $[t_0, T]$, et que l'on dispose de N processeurs. La parallélisation sur le temps consiste à diviser l'intervalle de simulation en N sous-intervalles de taille $\frac{T-t_0}{N}$ et à assigner la simulation du modèle pour chacun des intervalles de temps à un processeur. Le processeur p_i simule le système complet sur l'intervalle $[t_0 + (i-1)\frac{T-t_0}{N}, t_0 + i\frac{T-t_0}{N}]$. Les conditions initiales de chaque processeur sont à préciser. Quand la simulation de chaque intervalle est finie, on compare l'état du i-ième processeur aux conditions initiales du $i+1$-ième (pour toute valeur i dans l'ensemble des processeurs). S'il y a égalité, on arrête la simulation; dans le cas contraire, on recalcule les conditions initiales et on relance la simulation. Plusieurs auteurs ont étudié l'approche temporelle et ont proposé des algorithmes de simulation :

- Chandy et Sherman [Chandy et Sherman, 1989] conçoivent la simulation comme le remplissage d'un rectangle *espace–temps* par les événements. Ils proposent de découper "au mieux" cet espace, et de simuler chaque portion par un processeur. L'exécution se ramène alors à la recherche d'un point fixe par la technique itérative de relaxation.
- Greenberg *et al.* [Greenberg *et al.*, 1990] ainsi que Baccelli et Canales [Baccelli et Canales, 1993] utilisent des équations de récurrence pour décrire l'évolution du système. Chaque processeur déterminant la date d'occurrence des événements pour une tranche de temps.

Coopération de simulateurs Une autre approche consiste à faire coopérer des simulateurs séquentiels existants. Cette technique a connu un essor récent sous l'impulsion des militaires américains désireux de faire coopérer des simulateurs dédiés (simulateurs de véhicules terrestres, d'avions, etc.) afin d'obtenir la simulation complète d'une zone de combat. De nouveaux protocoles ont vu le jour sous le terme *Distributed Interactive Simulation* (DIS), la mise en œuvre est la suivante :

- à l'initialisation, les sous-simulateurs chargent la base de données décrivant le terrain et échangent les informations sur les entités simulées (caractéristiques générales, représentation graphique, position et déplacement initial, etc.);
- durant l'exécution, chaque sous-simulateur simule les entités dont il est responsable. Pour les interactions avec les autres entités, il y a approximation de la position (algorithme de *dead reckoning* [Lin et Schab, 1994]) et communication par messages. Chaque sous-simulateur calcule aussi la position de ses entités par le même algorithme d'approximation, dès qu'il y a une différence entre la position réelle et la position estimée,

le simulateur diffuse, aux autre simulateurs, les nouvelles caractéristiques (position et déplacement). Le trafic est ainsi limité.

Le protocole DIS ne spécifiant aucun protocole de synchronisation, le bon fonctionnement nécessite des interactions quasiment temps réel entre les sous-simulateurs. Il est alors possible de substituer un sous-simulateur par une interface homme-machine et d'effectuer ainsi de l'entraînement interactif. Les protocoles de communication et les algorithmes d'approximation sont en cours de standardisation [IEEE, 1993]. Signalons les nombreux travaux actuellement en cours afin de spécifier le successeur de DIS qui est appelé *High Level Architecture* (voir les nombreux articles paru dans [DIS, 1996]). L'objectif principal de HLA est de structurer l'interconnexion entre les simulateurs afin de réduire les communications et de permettre le regroupement d'un plus grand nombre de simulateurs géographiquement éloignés.

Distribution spatiale Une autre possibilité consiste à organiser le modèle en sous-modèles. A chaque sous-modèle est associé un *processus logique* responsable de sa simulation, les processus logiques sont alors exécutés par des *processus physiques* sur les machines (voir par exemple [Fujimoto, 1990]). Chacun des processus logiques évolue de manière concurrente, les communications se faisant par envoi de messages portant une date de simulation. Au sein d'un processus logique, les événements sont traités en séquences, les points d'interaction devant être synchronisés. La distribution spatiale exploite le parallélisme intrinsèque au modèle. Une seconde source de parallélisme consiste à traiter en parallèle, par des processus logiques différents, des événements programmés à des dates de simulation différentes à condition qu'ils ne soient pas liés par une relation de causalité.

La principale difficulté lors d'une implémentation consistera à réduire les temps d'exécution tout en garantissant la causalité. Nous présentons dans la section suivante plusieurs algorithmes de maintien de la causalité.

3.2 Conservation de la causalité : étude et classification des protocoles

Il existe une littérature abondante traitant des protocoles de synchronisation (une présentation peut être trouvée dans [Ferscha, 1995a, Fujimoto, 1990, Fujimoto et Nicol, 1992]), il est malheureusement difficile de s'y retrouver, d'autant plus que certaines propositions sont des variantes ou des combinaisons d'autres protocoles. Ces protocoles sont généralement présentés, pour des raisons historiques, de manière dichotomique : d'un côté les protocoles conservatifs et de l'autre les protocoles optimistes. Nous considérons comme Reynolds [Reynolds, 1988] que cette vision est trop schématique et nous avons retenu deux des critères de sa taxonomie pour cette présentation (cf. Fig. 3.1) :

1. *Agressivité* – Consiste à traiter des messages à partir d'une information partielle. Les actions entreprises risquent donc de devoir être défaites.

2. *Risque* – Consiste à émettre des messages provenant d'un calcul qui risque d'être défait. Il y a donc risque de propager l'erreur.

Critère de distribution	Granularité	Caractéristiques
Application	Forte (simulateur)	Mise en œuvre aisée. Quand applicable, l'accélération est de n sur n processeurs.
Fonction	Moyenne et fixe (tâche)	Parallélisme limité au nombre de tâches. Accélération si la granularité des tâches est importante par rapport aux communications.
Evénement	Faible (événement)	Nécessite des informations sur la nature des événements et est adapté aux machines à mémoire partagée. Accélération si de nombreux événements à forte granularité peuvent être exécutés simultanément.
Temps	Forte (simulateur)	Techniques de mise en œuvre spécifiques (composante mathématique importante). L'accélération potentielle n'est pas bornée. Peut être couplé à une approche spatiale.
Simulateurs spécialisés	Forte (simulateur)	Permet de réutiliser des simulateurs existants au sein d'une simulation de plus haut niveau. L'accélération n'est pas l'objectif principal.
Spatiale	Moyenne et variable (sous-système)	Approche la plus répandue. Bon compromis entre difficulté de mise en œuvre/accélération/ généralité.

Tab. 3.1 – *Récapitulatif des critères de distribution*

La conception d'un algorithme consiste alors à sélectionner le degré de risque et d'agressivité qui sera appliqué.

3.2.1 Protocoles avec risque et agressivité nuls

Dans cette approche, les messages et les événements sont traités selon l'ordre strict d'occurrence dans le temps de simulation. Nous présentons deux classes d'algorithmes entrant dans ce cadre, les approches dirigées par le temps et les approches conservatrices.

Approches dirigées par le temps

Les approches dirigées par le temps sont une mise en œuvre distribuée de la technique de progression à pas fixes du temps présentée à la section 2.2.2. On distingue les algorithmes synchrones et ceux asynchrones. Dans le cas synchrone, tous les sous-systèmes sont à la même date de simulation et progressent à la même vitesse. Un sous-système peut passer à la simulation des événements de l'intervalle suivant quand l'ensemble des sous-systèmes a simulé les événements de l'intervalle courant et quand les messages en transit ont été traités. La simulation est généralement organisée en une succession de deux phases : (1) sélection des événements pouvant être simulés et simulation, (2) communication et synchronisation globale. Dans le cas asynchrone, un sous-système peut passer à la simulation de l'intervalle suivant, dès que ses prédécesseurs ont simulé l'intervalle courant. La synchronisation se fait alors de proche en proche par envoi de messages. Afin d'éviter le traitement des intervalles vides, il est aussi possible de ne re-synchroniser qu'à la date de la prochaine interaction entre sous-systèmes.

De nombreux algorithmes ont été proposés pour la gestion de l'horloge (centralisée, distribuée, hiérarchique, etc.) et de la synchronisation (anneau virtuel, primitive système, etc.).

Approches asynchrones conservatrices

Historiquement, la première méthode asynchrone fut une méthode conservatrice, elle fut développée indépendamment par Chandy et Misra [Chandy et Misra, 1979], et par Bryant [Bryant, 1977]. La synchronisation repose sur le principe suivant: si chaque sous-système respecte localement le principe de causalité, alors ce dernier sera respecté globalement dans toute la simulation.

Respecter le principe de causalité, oblige le sous-système à déterminer s'il est en mesure de simuler un événement. Il pourra simuler un événement s'il est certain de ne pas avoir ensuite à en simuler un de date antérieure. S'il n'en n'est pas certain, il devra se mettre en attente. Afin de progresser, on associe un *canal* à chaque sous-système susceptible de lui envoyer un message. Les messages devront circuler par ordre chronologique sur ces canaux. A chaque canal est associé une file FIFO, ainsi qu'une horloge indiquant la date du premier message à traiter, ou celle du dernier message traité si le canal est vide. Le sous-système pourra simuler un événement si la date de ce dernier est inférieure à celle des horloges des canaux.

Mais dans ce protocole, le fonctionnement des sous-systèmes peut être source *d'interblocage*. Notons que cette situation d'interblocage ne correspond pas à un *interblocage du modèle*, mais résulte de l'algorithme de synchronisation utilisé. Il faut alors soit éviter l'interblocage, soit le détecter et le résoudre.

Eviter l'interblocage : les messages *nuls* Afin d'éviter les situations d'interblocage, les sous-systèmes envoient sur les canaux provisoirement inutilisés des *messages nuls*. Les messages nuls reçus par un sous-système lui permettent d'incrémenter son horloge et donc de simuler d'éventuels événements en attente. Malheureusement, cette solution ne suffit pas à prévenir l'interblocage dans le cas où il y a un cycle dans le réseau. Pour résoudre ce problème, on couple l'émission de messages nuls avec un mécanisme de *prédiction* (appelé *lookahead*). Pour élaborer les messages nuls, les sous-systèmes font une prédiction sur la date du prochain message émis en fonction de leur état courant. A la réception d'un message nul, si le sous-système n'a pas d'événements à simuler, il émet sur ses canaux de sortie des messages nuls avec une date incrémentée de la prédiction. La prédiction est une caractéristique intrinsèque du modèle simulé et ne peut être déterminée automatiquement par le simulateur, c'est donc au modélisateur de l'introduire dans sa programmation.

Les performances de cette approche sont directement liées à la capacité d'avoir une bonne prédiction (voir [Fujimoto, 1989, Loucks et Preiss, 1990]). Lors de la mise en œuvre, Lin *et al.* [Lin *et al.*, 1990] proposent de distinguer les systèmes possédant ou non un lookahead explicite (pour chaque cycle, au moins un processus logique possède une valeur définie au préalable non nulle de lookahead) et implicite (la valeur du lookahead provient du comportement de l'entité et peut être extraite à partir de son état courant). Les systèmes possédant un lookahead explicite peuvent être simulés par l'algorithme classique de Chandy/Misra et celui de Bryant. Nicol étudie dans [Nicol, 1988] les files d'attente FCFS qui sont des systèmes avec lookahead implicite. Il propose de pré-calculer les temps de service et le routage et d'exploiter cette information pour le lookahead. Pour les systèmes ne possédant aucun lookahead, l'approche classique consiste à utiliser les algorithmes de détection/résolution de l'interblocage ou les algorithmes synchrones. Lin *et al.* [Lin *et al.*, 1990] proposent une autre approche qui consiste à agréger les sous-systèmes fortement connexes constituant le cycle sans lookahead.

Les principales optimisations des protocoles à base de messages nuls consistent à améliorer la prédiction et à diminuer le nombre de messages nuls. La prédiction peut être améliorée en exploitant le lookahead implicite. Le nombre de messages nuls peut être réduit par exemple, en ne les envoyant qu'à la demande ([Bain et Scott, 1988, Misra, 1986]). Une connaissance de la topologie peut aussi éviter aux processus logiques de se mettre inutilement en attente. Par exemple, Cai et Turner proposent avec leur algorithme *carrier null message* [Cai et Turner, 1990] d'enrichir les messages nuls avec des informations de nature topologique.

Détection/résolution de l'interblocage Chandy et Misra ont aussi développé [Chandy et Misra, 1981, Misra, 1986] d'autres techniques alternatives qui ne nécessitent pas l'utilisation de messages nuls. Ils ont proposé des mécanismes séparés pour la détection des interblocages et d'autres mécanismes pour les briser. Le principe de base consiste à aviser le sous-système possédant l'événement avec la date la plus petite de toute la simulation qu'il peut le traiter. Pour cela, les algorithmes de détection de la terminaison de programmes distribués [Mattern, 1987], détection d'interblocage en environnement distribué [Chandy *et al.*, 1983] peuvent être utilisés.

Dans la plupart des algorithmes, l'interblocage n'est détecté que quand l'ensemble des sous-systèmes est atteint. Liu et Tropper [Liu et Tropper, 1990] proposent une optimisation qui consiste à détecter et à résoudre l'interblocage localement avant qu'il ne se propage.

Autres protocoles Une autre technique possible consiste à combiner la synchronisation dirigée par le temps et la synchronisation dirigée par les événements. Lubachevsky propose dans [Lubachevsky, 1988] un protocole (*bounded lag*) qui exploite l'écart de temps séparant des événements causalement liés afin de désynchroniser les sous-systèmes et de simuler le maximum d'événements de manière concurrente. La connaissance de cet écart de temps est difficile à généraliser et repose souvent sur l'existence d'un temps (logique) de propagation des événements entre sous-systèmes et de périodes durant lesquelles un sous-système n'émettra aucun message (périodes *opaques* dans la terminologie de l'auteur). Sokol *et al.* proposent une approche similaire (*Moving Time Window* [Briscoe *et al.*, 1988, Sokol et Stucky, 1990]) mais qui ne garantit pas le respect de la causalité si la fenêtre est trop grande. Les auteurs proposent un compromis entre vitesse d'exécution et correction ; différentes techniques permettant d'améliorer la correction sont aussi avancées (essentiellement une classification des événements modifiant ou non l'entité, retournant ou non une valeur, etc.).

3.2.2 Protocoles avec risque et agressivité non bornés

Ces algorithmes sont à l'opposé des précédents, la synchronisation y est entièrement relâchée et tout message peut être émis ou traité. On parle alors d'algorithmes optimistes, car ils considèrent, a priori, que la cohérence sera maintenue, une erreur de causalité détectée devant alors être corrigée. La méthode Time Warp est très certainement la méthode optimiste la plus connue.

Time Warp

Dans cette méthode développée par Jefferson [Jefferson, 1985], une erreur de causalité est détectée quand un sous-système reçoit un message avec une date antérieure à la date de simulation locale (ce type de message est appelé *straggler*). Le sous-système doit alors effectuer un retour dans le temps (*rollback*) en revenant à la date de simulation précédant celle du message. Il doit restaurer son état, et pour chaque message émis, envoyer un *anti-message* l'annulant. La réception d'un anti-message peut amener un sous-système à effectuer un rollback si le message correspondant a déjà été traité. Chaque sous-système doit donc stocker les modifications des variables d'états ainsi que les messages émis. Pour pouvoir récupérer de la mémoire (*fossil collection*) et effectuer les entrées/sorties, la notion de *temps virtuel global* (*Global Virtual Time* -GVT-) a été introduite. Elle représente une borne inférieure sur les dates de retour en arrière possibles ; toutes les actions effectuées à un temps simulé inférieur au GVT ne seront jamais défaites. Un algorithme distribué permet de faire progresser ce GVT à partir des horloges locales (des algorithmes de calcul du GVT peuvent être trouvés dans [Baldwin *et al.*, 1991, Bauer et Sporrer, 1992, Bellenot, 1990, Concepcion et Kelly, 1991, Lin et Lazowska, 1990, Mattern, 1993]). Notons que certaines optimisations du Time Warp introduisent des variantes dans la définition du GVT.

De nombreux algorithmes ont été proposés afin d'améliorer les performances du Time Warp. Il est possible d'agir sur trois points : (1) contrôle de la progression du temps, (2) gestion des rollback et des anti-messages, et (3) gestion de la mémoire. Les techniques de contrôle de la progression du temps rentrent dans le cadre du contrôle du risque et de l'agressivité et seront donc présentés dans la section suivante. Nous présentons dans la suite les deux autres approches.

Gestion des rollback et des anti-messages Dans le Time Warp classique, un sous-système détectant une incohérence temporelle émet directement les anti-messages, on parle alors d'annulation agressive (*agressive cancellation*). Gafni [Gafni, 1988] propose une approche paresseuse (*lazy cancellation*) où le sous-système n'émet pas directement les anti-messages, il restaure le contexte puis simule l'événement. Si certains messages précédemment émis sont identiques à ceux qu'il va émettre, il est inutile de les annuler. Cette approche permet de réduire les communications et de diminuer les rollback inutiles mais elle permet à des calculs erronés de se propager. Cette méthode ne doit donc être utilisée que si la probabilité de générer les mêmes messages est très grande. Reiher *et al.* comparent expérimentalement dans [Reiher *et al.*, 1990] ces deux protocoles. Les performances sont très proches, moins de 10% d'écart, et dépendent fortement du système simulé. Madisetti *et al.* ont proposé une solution permettant d'éviter que des calculs erronés ne se propagent. Dans leur algorithme (*wolf call* [Madisetti *et* *al.*, 1988]), le sous-système ayant détecté une erreur de causalité émet un message particulier qui interrompt toute la simulation. Ce n'est qu'après avoir restauré globalement les contextes, que la simulation pourra reprendre. Une autre approche paresseuse (appelé *lazy re-evaluation*, *jump forward* ou encore *lazy rollback*) est proposée par West dans [West, 1998] et consiste à comparer, après un rollback, le nouvel état à l'état erroné de même date. S'ils sont identiques et si aucun nouveau message n'est arrivé, le sous-système revient à l'état futur. Cette approche est plus coûteuse en temps et plus complexe à implémenter en raison de la comparaison des états.

Certains protocoles optimistes ont la caractéristique très importante de ne pas être nécessairement bornés par le chemin critique [1] et donc de permettre des accélérations super-linéaires. Jefferson et Reiher ont démontré dans [Jefferson et Reiher, 1991] que les protocoles *lazy cancellation* et *lazy rollback* ont cette propriété.

Gestion de la mémoire Une gestion efficace de la mémoire contribue aux bonnes performances et est indispensable pour rendre le protocole fonctionnel dans un environnement réel à mémoire finie.

Dans le Time Warp classique, l'ensemble des données du sous-système est sauvegardé périodiquement. De la mémoire peut être récupérée en effectuant plus souvent le calcul du GVT, ce qui ralentit sensiblement l'exécution. L'optimisation la plus simple consiste alors à augmenter l'écart de temps entre deux sauvegardes (*periodic checkpointing, interleaved state saving* [Lin et Lazowska, 1989, Lin *et al.*, 1993]). Mais cette technique augmente le coût d'un rollback, car il faut alors revenir plus loin dans le passé. La principale difficulté consiste donc à déterminer statiquement ou dynamiquement la valeur de l'intervalle séparant deux sauvegardes (une présentation ainsi qu'une comparaison des principaux algorithmes existants peuvent être trouvées dans [Fleischmann et Wilsey, 1995]). Une autre approche consiste à diminuer l'espace mémoire d'une sauvegarde en ne sauvegardant que les modifications [Bauer et Sporrer, 1993]. C'est une approche intéressante pour les processus ayant une grande image mémoire, et des changements peu fréquents et localisés. Elle augmente par contre la complexité des algorithmes, car il faut être en mesure de reconstruire l'état global à l'aide des sauvegardes partielles. Palaniswamy et Wilsey comparent analytiquement dans [Palaniswamy et Wilsey, 1993] les techniques de sauvegarde périodique et incrémentielle. Et enfin, il est aussi possible d'accélérer l'ensemble des opérations de gestion mémoire en faisant appel à des composants matériels spécifiques [Fujimoto et Jand, 87, Fujimoto *et al.*, 1992].

Malgré ces diverses optimisations, il est toujours possible qu'un sous-système épuise l'ensemble de sa mémoire disponible. Dans la première solution proposé par Jefferson [Jefferson, 1985], un sous-système désirant récupérer de la mémoire retourne à l'émetteur le message non traité portant la plus grande date de simulation (*message sendback*). Lin [Lin, 1992] propose que le sous-système libère de la mémoire en effectuant un *rollback artificiel*. Ces deux techniques nécessitent au minimum un rollback. Preiss et Loucks [Preiss et Loucks, 1995] proposent eux que certaines sauvegardes d'états soient détruites afin de récupérer de la mémoire (*prune-back*). Il n'y a alors pas de rollback immédiat, mais un rollback ultérieur sera éventuellement plus coûteux. Gafni [Gafni, 1988] généralise cette approche, le sous-système sélectionne un

1. Le chemin critique est calculé à partir du temps réel qu'il faut pour simuler chaque événement et des contraintes d'antériorité et de causalité. Le chemin critique est une borne inférieure sur le temps de simulation du modèle pour tout protocole conservatif.

objet mémoire à détruire (celui ayant la date maximale) : si c'est un message d'entrée il est retourné à l'émetteur, si c'est un message de sortie un anti-message est émis, si c'est un état il y a rollback. Dans le cas d'une mémoire partagée, Jefferson [Jefferson, 1990] propose un algorithme proche de celui de Gafni (*cancelback*), la mémoire récupérée est celle du message globalement le plus récent.

Lin a introduit dans [Lin, 1992] la notion d'optimalité d'un mécanisme de gestion mémoire. Un algorithme de gestion mémoire est optimal si l'exécution des simulations ont une complexité spatiale $O(M_s)$, où M_s est la quantité de mémoire utilisée par la simulation séquentielle du modèle. Dans l'article, l'auteur évalue selon ce critère les algorithmes *fossil collection*, *message sendback*, *cancel back* et *artificial rollback*.

3.2.3 Protocoles contrôlant le risque et l'agressivité

Dans les protocoles avec risque et agressivité nuls, le parallélisme du système n'est pas entièrement exploité. En effet, si la simulation d'un événement risque de violer la causalité, on attend. Dans les protocoles ne bornant ni le risque ni l'agressivité, le parallélisme du modèle est alors très bien exploité, mais les coûts associés aux mécanismes de restauration de la causalité en cas de violation font souvent perdre le bénéfice de la distribution. Une approche intéressante consisterait à varier, statiquement ou dynamiquement, le degré de risque et d'agressivité. Toujours pour des raisons historiques, la plupart des propositions ont porté soit sur la relaxation des contraintes d'attente du protocole conservatif, soit sur l'introduction d'une forme de contrôle au sein de l'algorithme optimiste.

La relaxation des contraintes d'attente du protocole conservatif consiste à simuler des événements de manière spéculative. Le degré d'agressivité n'est pas borné, mais le degré de risque est nul. Mehl et Horst proposent avec leur algorithme (*Speculative Simulation* [Mehl et Horst, 1991]), d'utiliser les périodes inactives des processus logiques afin de simuler des événements futurs. Lors de la simulation, ces événements modifient une copie des variables et les messages en sortie sont conservés. Steinman propose un algorithme similaire (*Breathing Time Buckets* [Steinman, 1992]) où les événements à risque sont quand même exécutés mais sans émission de messages. En cas de rollback, ce dernier sera local.

Le contrôle de l'algorithme optimiste est en général introduit par le biais d'un mécanisme de *fenêtre* permettant de ralentir les sous-systèmes les plus rapides. C'est le cas des algorithmes proposés par Briner dans [Briner, 1991] (*bounding time window*) et par Matsumoto et Taki dans [Matsumoto et Taki, 1993] (*adaptive time ceiling*). La fenêtre permet de contrôler l'agressivité, Steinman propose lui de limiter le risque (*breathing Time Warp* [Steinman, 1993]), les sous-systèmes fonctionnent en Time Warp jusqu'à un certain point (notion de distance par rapport au GVT), puis à partir de là les messages ne sont plus émis en sortie mais stockés, évitant ainsi les rollback en cascade. Lubachevsky *et al.* proposent une extension de leur algorithme conservatif *bounded lag* appelé *filtered rollback* [Lubachevsky *et al.*, 1989] dans lequel la prédiction sur le temps séparant les événements liés par une relation de causalité est utilisée afin de limiter les rollback.

Les protocoles à fenêtre permettent de limiter l'optimisme du Time Warp, une approche plus sophistiquée consiste à faire varier dynamiquement la synchronisation entre un mode conservatif et un mode optimiste. C'est ce que propose Ball et Hoyt dans [Ball et Hoyt, 1990] avec l'algorithme *adaptive Time Warp control algorithm*. Les auteurs proposent de suspendre temporairement les sous-systèmes ayant récemment trop souvent effectué de rollback. La durée de suspension (*blocking window*) est basée sur une évaluation du temps d'attente de messages en mode conservatif et du temps perdu par un rollback. Fersha propose dans [Ferscha, 1995b] (*probabilistic direct optimism control* -PADOC-) de maintenir dans chaque sous-système un historique des dates des messages arrivant et d'effectuer une prévision sur la date du prochain message. Cette prévision est utilisée pour mettre le sous-système en attente. L'auteur propose divers algorithmes de prévision ; un compromis devant être fait entre le nombre d'échantillons conservé, le coût de l'algorithme de prévision et la qualité de cette dernière.

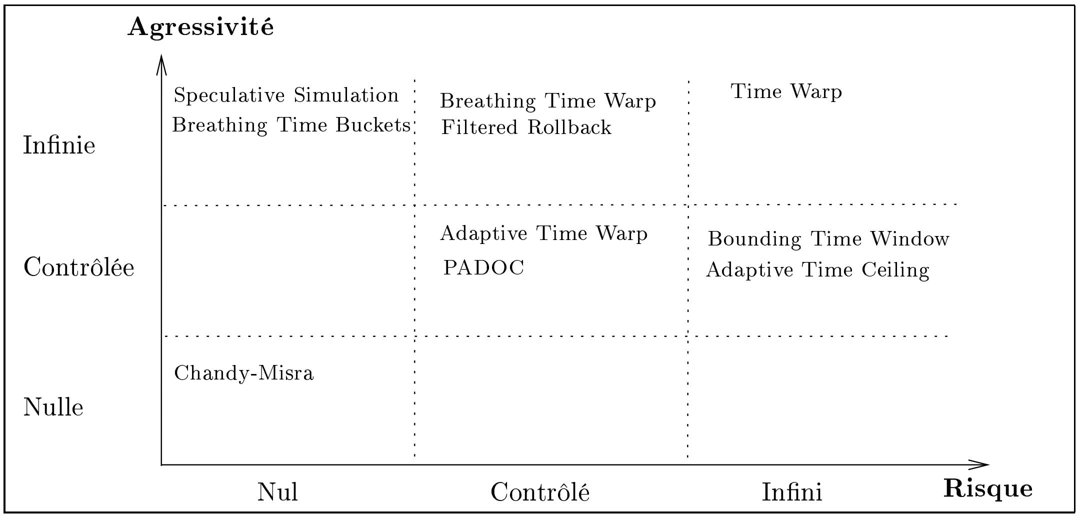


FIG. 3.1 – *Classification des protocoles de synchronisation*

3.3 Discussion

Les bonnes performances d'une simulation distribuée dépendent de la nature de l'application. Pour une distribution spatiale, l'application doit contenir du parallélisme intrinsèque et se prêter à un découpage en composantes. Le nombre de composantes dépendra de la granularité et de la dispersion des événements. La nature des interactions (topologie, dispersion, etc.) déterminera la classe d'algorithmes à utiliser. Pour un système fortement synchrone, on utilisera naturellement les approches dirigées par le temps, les approches asynchrones s'appliqueront aux autres classes de systèmes. Une fois le protocole sélectionné, sa mise en œuvre devra aussi contenir du parallélisme. Nous discutons dans les trois sections suivantes des mérites et des faiblesses des algorithmes asynchrones présentés précédemment.

L'architecture de la machine parallèle est aussi à considérer. Les principaux facteurs intervenant sont : la mémoire (partagée ou pas, quantité par processeur, etc.), les processeurs (nombre, hétérogénéité, etc.), le support de communication (débit, latence, réseau d'interconnexion, etc.) et enfin la disponibilité de primitives systèmes spécifiques très efficaces (synchronisation et communication globales, etc.).

Devant la multitude de protocoles existants, il est difficile de sélectionner celui qui donnera les meilleures performances. Cette difficulté se trouve accentuée par la grande sensibilité de l'accélération au modèle et au paramétrage de ce dernier. Un environnement de simulation doit donc proposer plusieurs protocoles ainsi que des outils d'aide à la sélection.

3.3.1 Simulation dirigée par le temps et simulation dirigée par les événements

La simulation distribuée dirigée par le temps possède les mêmes faiblesses que dans le cadre séquentiel (voir la section 2.3.1). Par contre, c'est la technique de conservation de la causalité la plus aisée à mettre en œuvre (pas de problèmes d'interblocage, de débordement de mémoire, etc.). La mise au point ainsi que la validation des simulations sont aussi plus faciles. D'autre part, elle permet de simuler des systèmes de topologie dynamique (ce qui est par exemple impossible avec les approches conservatrices) et est naturellement adaptée à la simulation de systèmes synchrones (voir par exemple [Yu *et al.*, 1989]). Cette approche est par contre particulièrement sensible à la répartition de charge entre processeurs. Les versions synchrones peuvent aussi être peu performantes lors de l'utilisation de plusieurs dizaines de machines faiblement couplées.

L'approche dirigée par les événements est caractérisé, dans le cadre distribué, par son non-déterminisme. Deux exécutions peuvent avoir des comportements différents en raison des variations de la charge des processeurs et de l'occupation de la bande passante.

3.3.2 Les deux extrêmes : simulation conservatrice et simulation optimiste

Pour les systèmes à topologie dynamique et ceux ayant des interactions irrégulières, l'approche optimiste semble être plus efficace que les approches conservatrices. Ces dernières sont adaptées à la simulation de systèmes physiques constitués d'un ensemble d'entités à forte granularité et faiblement couplées. Les approches conservatrices peuvent aussi exploiter certaines spécificités structurelles du modèle.

D'une manière empirique, il semble que les algorithmes optimistes soient plus performants que ceux conservatifs mais consomment plus de mémoire (il y a un compromis espace/temps à faire). La comparaison analytique est particulièrement difficile en raison du nombre très important de facteurs à prendre en compte. Pour ce qui est d'une comparaison expérimentale, il faudrait préalablement réaliser une même implémentation. Mais le problème des optimisations d'un protocole inexistantes dans l'autre (par exemple la gestion de la mémoire en Time Warp) se posera alors. Nous présentons donc dans le tableau 3.2 une comparaison qualitative de ces deux familles (cette comparaison est tirée de [Ferscha, 1995a]).

3.3.3 Contrôle du risque et de l'agressivité

Les techniques de synchronisation mixtes sont très certainement les plus intéressantes et constituent un domaine de recherche actif. Les techniques auto-adaptatives sont les plus ambitieuses et libèrent le modélisateur de la difficile tâche de sélection du mode de synchronisation. Notons néanmoins que ces techniques sont plus difficiles à mettre en œuvre notamment en présence de systèmes dont le mode de fonctionnement change souvent.

	Conservatrice	Optimiste
Parallélisme	Le parallélisme du modèle n'est pas entièrement exploité.	Le parallélisme du modèle est entièrement exploité.
Prédiction	Nécessaire pour le fonctionnement du protocole, vital pour les performances.	Non nécessaire, mais peut être utilisé pour certaines optimisations.
Comportement	Performant si l'utilisation des canaux est homogène. La dispersion des événements dans l'espace et dans le temps ne pose pas de problème.	Performant si l'avancement du temps est homogène entre les processus physiques. La dispersion des événements dans l'espace et dans le temps peut dégrader les performances.
GVT	Implicite, découle de l'algorithme.	Doit être calculé explicitement de manière très efficace. Un calcul centralisé peut entraîner un goulot d'étranglement, le calcul distribué est coûteux en communication.
Etats		Meilleures performances quand l'espace des états ainsi que l'occupation mémoire d'un état sont réduits. Mécanisme de sauvegarde/restauration des états, complexe et vital pour les performances.
Mémoire	Consommation mémoire conservatrice (de par l'algorithme).	Consommation mémoire agressive. Coût de la sauvegarde des états. La gestion mémoire nécessite un calcul du GVT fréquent et efficace. Algorithme de gestion mémoire complexe.
Communication et topologie	Communications FIFO. Séparation des canaux d'entrées. Topologie statique.	Communications quelconques. Topologie dynamique.
Implémentation	Directe. Structure de contrôle relativement simple.	Difficile et complexe à mettre au point. Structure de contrôle complexe. Flot d'exécution à gérer. De nombreux facteurs peuvent faire l'objet d'améliorations.
Sensibilité	Grande sensibilité au paramétrage du modèle.	Grande sensibilité à la répartition de la charge.

Tab. 3.2 – *Comparaison des approches conservatrice et optimiste*

4. PROGRAMMATION OBJET ET SIMULATION

Nous réalisons dans ce chapitre une synthèse des relations unissant les trois domaines d'étude de notre recherche : la simulation, la programmation objet et la programmation répartie.

4.1 La programmation objet

La taille des programmes et les problèmes de maintenance ont mis en évidence la nécessité de promouvoir l'abstraction des données, la modularité et la réutilisabilité du code. La programmation et la conception objet constituent à l'heure actuelle la solution la plus souple et la plus prometteuse pour ces différents besoins.

4.1.1 Le paradigme objet

D'une manière concrète, l'objet est une entité regroupant les données (composante statique) et les procédures (composante dynamique) de traitement (notion *d'encapsulation*). Les détails d'implémentation sont cachés, le monde extérieur n'a accès aux données que par l'intermédiaire d'un ensemble d'opérations constituant l'interface. La seule manière de communiquer avec un objet est d'invoquer une des procédures de son interface. C'est l'objet qui est responsable de la manière dont l'opération est effectuée. L'objet est donc défini par son comportement et non plus par sa structure (*abstraction de données*). Le programmeur n'a pas à se soucier de la représentation physique des entités manipulées et peut donc raisonner en termes d'abstractions.

La programmation objet est dite dirigée par les données : pour traiter une application, le programmeur commence par définir les types d'objets appropriés (*les classes*), avec leurs opérations spécifiques, chaque entité manipulée dans le programme est un représentant (*une instance*) d'un de ces types. Simula [Lamprecht, 1981] et Smalltalk [Goldberg et Robson, 1983] sont les premiers langages de classes [1]. Simula est le premier véritable langage de classes, mais c'est Smalltalk qui possède une approche entièrement uniforme : la seule entité est l'objet et la seule structure de contrôle est l'envoi de message, qui permet de communiquer avec les objets.

Ces principes permettent à la fois une programmation rigoureuse et puissante, tout en restant souple et agréable :

- *abstraction de données* : inutile de connaître la représentation physique d'un objet, c'est à lui de décider comment effectuer l'opération demandée. La *surcharge* des méthodes est

1. Pour une présentation exhaustive des langages de classes et des autres familles de langages objets, nous conseillons la lecture du livre de Colnet *et al.* [Colnet *et al.*, 1989].

possible, des objets de classes différentes répondent chacun à leur manière aux mêmes messages ;

- *modularité et flexibilité* : le programme est structuré en objets dont il est facile de modifier l'implémentation, à condition que l'interface reste la même ;
- *réutilisabilité* : un objet étant défini par son comportement, il est facile d'inclure sa classe dans une bibliothèque. Le programmeur pourra utiliser cette classe pour construire des objets de même type ou pour construire des objets plus spécifiques. La spécialisation est possible grâce au mécanisme d'*héritage* qui permet de partager et de réutiliser des propriétés entre classes. Les connaissances les plus générales sont mises en commun dans des classes qui sont ensuite spécialisées par définitions de *sous-classes* successives, contenant des connaissances de plus en plus spécifiques. La spécialisation peut être réalisée par *enrichissement* (la sous-classe est dotée de nouveaux attributs et/ou de nouvelles méthodes), ou par *substitution* (une nouvelle définition est donnée à une méthode héritée) ;
- *lisibilité et compréhension* : découle de l'encapsulation, de la surcharge et de la modularité.

4.1.2 La modélisation et la conception objet

Un des apports majeurs de l'objet est la possibilité de travailler et de penser en terme de domaine d'application durant la plus grande partie du cycle de vie du logiciel. Les méthodes objets couvrent les phases d'analyse, de conception, de programmation et de modélisation d'applications. Meyer [Meyer, 1988] fut probablement un des premiers à proposer à la fois une méthodologie de conception et de programmation objet, et le langage de programmation associé (Eiffel). Depuis, le nombre de méthodes n'a cessé de croître. Certaines s'appliquent à un domaine particulier (par exemple la méthode *Real Time Object-Oriented Modeling*) ou ont au contraire une vocation généraliste (par exemple les méthodes proposées par Booch [Booch, 1991], par Rumbauch [Rumbauch, 1996], etc.), elles peuvent couvrir soit une partie, soit l'ensemble du cycle de vie d'un logiciel et ce à l'aide d'un ou de plusieurs formalismes. Après cette phase d'effervescence, seules quelques méthodes restent largement utilisées et l'on assiste actuellement au rapprochement voire à la fusion et à la conception de méta-méthodes (c'est le cas du formalisme *Unified Modeling Language* [Fowler et Scott, 1997] qui regroupe les propositions de Booch, de Jacobson et celles de Rumbauch).

Un autre grand intérêt du paradigme objet est la réutilisabilité et nous pouvons en distinguer trois formes : la réutilisation de bibliothèques de classes, la réutilisation d'un ensemble de classes constituant une infrastructure logicielle et enfin la réutilisation de classes spécialisées pour la résolution d'un problème de conception dans un contexte précis.

Réutilisation de bibliothèques de classes Cette première forme de réutilisation consiste à fournir des bibliothèques accompagnées d'une description des classes et des méthodes. C'est une forme de réutilisation très simple qui est dans son ensemble orientée traitement (librairie graphique, librairie de structures de données, etc.). La réutilisation de classes constitue l'évolution naturelle de la programmation procédurale par API.

Réutilisation d'infrastructures Dans un second temps vient la réutilisation d'un ensemble d'objets constituant une structure, une architecture, et qui collaborent afin d'assumer un ensemble de responsibilités. Un tel ensemble de classes est appelé un *framework* (ou *canevas*). Les frameworks sont utilisés pour construire des familles d'applications qui partagent une architecture et une implémentation. Utiliser un framework consiste à sélectionner les traitements clés a personnaliser et à fournir le code associé. Le framework utilise le code quand nécessaire et l'utilisateur n'a pas à se soucier de la structure, de l'ordre d'exécution et des appels aux bibliothèques d'API de niveau système. Il est aussi possible de personnaliser un framework en remplaçant certaines de ses classes. Pour cela, il suffit de faire appel aux techniques classiques de l'approche objet comme l'héritage multiple et la surcharge de méthodes. Une application construite à l'aide d'un framework est plus facile à maintenir et offre une interface plus cohérente. D'un autre côté, le framework spécifiant l'architecture restreint le degré de liberté du programmeur. C'est pourquoi il est important de sélectionner un framework le plus flexible et extensible possible. Les frameworks sont orientés domaines d'applications (programmation client/serveur, interface base de données, etc.). Parmi les avantages des frameworks, nous avons :

- *Une infrastructure préfabriquée.* Les frameworks réduisent les temps de codage et de mise au point en fournissant des sous-systèmes déjà opérationnels.
- *Guide d'architecture.* Les frameworks fournissent une architecture pré-cablée, masquant ainsi la complexité du logiciel en fournissant un plus haut niveau d'abstraction.
- *Applications moins monolithiques.* Les frameworks encouragent à écrire des fragments limités d'applications qui se connectent au framework approprié.
- *Les bases d'une industrie de composants logiciels.* Les frameworks bien conçus permettent à d'autres sociétes informatiques de développer et de commercialiser des parties de code ou des composants entiers.
- *Une maintenance réduite.* La réduction du code utilisateur entraine une réduction directe des coûts de maintenance. De même, grâce à l'héritage, la correction d'un bogue ou une optimisation au niveau du framework profite directement à l'utilisateur.

Réutilisation de *solutions* à un problème de conception Une dernière forme de réutilisation, plus récente, consiste à spécifier un ensemble de classes spécialisées pour la résolution d'un problème de conception dans un contexte précis. Un tel ensemble de classes s'appelle un *design pattern* [Gamma *et al.*, 1995] (aussi appelé *patron de conception*), chaque design pattern est caractérisé par un nom, le problème confronté, la solution apportée et ses conséquences. Les design patterns sont proches des frameworks, mais se distinguent par :

- les design patterns sont des éléments d'architecture plus petits que les frameworks. Un framework peut contenir plusieurs design patterns, le contraire est impossible ;
- les design patterns sont plus abstraits que les frameworks ;
- les design patterns sont moins spécialisés que les frameworks. Un framework est spécifique à un domaine d'application, le problème résolu par le design pattern peut se retrouver dans plusieurs domaines.

Les design patterns sont orientés problèmes (assurer qu'une classe ne possède qu'une instance accessible de tout le code, codage d'un itérateur, etc.). Nous présentons à la table 4.1 un bref résumé des caractéristiques qui distinguent les frameworks de la programmation procédurale par API et des bibliothèques de classes orientées objet.

Caractéristique	Frameworks	Librairies de classes	Lib. de procédures (API)
Modèle de programmation	Le framework constitue l'application et gère le flot de contrôle.	Le programmeur doit coder le flot de contrôle et l'interface entre les classes.	Le programmeur doit coder le flot de contrôle et l'enchaînement des appels aux procédures.
Structure du programme	Coopération de frameworks.	Application monolithique utilisant des librairies.	Application monolithique utilisant des librairies.
Spécialisation	Le framework utilise le code du programmeur. On peut spécialiser certaines composantes.	On dérive de nouvelles classes.	Il faut écrire du code et utiliser d'autres API.
Granularité du contrôle	Moyenne. On ne peut spécialiser que certaines composantes.	Importante. On peut dériver toutes les classes.	Importante. On peut tout recoder.
Abstraction des services	Elevée.	Faible. Abstraction de l'API, mais possède son niveau de complexité. Il faut déterminer les méthodes à appeler et dans quel ordre.	Très faible. On manipule directement l'API. Il faut déterminer les fonctions à appeler et dans quel ordre.
Code à écrire à l'utilisation	Très peu.	Modéré.	Beaucoup.
Coût de la maintenance	Faible.	Moyen.	Elevé.
Réduction de la complexité	Oui. Le framework guide l'architecture.	Non. Il faut rechercher les classes et intégrer plusieurs librairies.	Non. Il faut développer toute l'application, comprendre et faire coopérer les API.
Temps pour développer une application	Faible.	Moyen. Dépend de la réutilisabilité des classes.	Important.
Réutilisation de code	Très élevée.	Elevée. Les classes sont réutilisées.	Moyenne. Certaines fonctions sont réutilisables.
Le modèle est-il bien connu ?	Non.	Moyennement.	Oui.

Tab. 4.1 – *Comparaison des frameworks, librairies de classes et librairies de procédures (Traduit de [Orfali* et al.*, 1996]).*

4.2 La programmation objet parallèle

D'une façon générale, le modèle objet est bien adapté à la programmation parallèle, car des processus fonctionnant de manière concurrente peuvent être facilement modélisés par des objets communiquant par envois de messages. Là encore, il n'est pas étonnant que le premier langage objet, Simula, ait été conçu pour gérer et synchroniser des co-routines. Caromel dans [Caromel, 1991] recense deux apports majeurs de la programmation objet :

- l'unification objet/processus, qui permet de fusionner les notions de *module*, de *type* et de *tâche*;
- l'unification appel de routine/communication entre processus, qui simplifie la programmation et permet de réutiliser directement, par le biais de l'héritage et du polymorphisme, les routines d'une classe séquentielle afin de programmer un processus.

4.2.1 Intégration de la concurrence dans le modèle objet

Papathomas a exploré dans [Papathomas, 1990] l'intégration du parallélisme dans le modèle objet (cf. Fig. 4.1). Les approches rencontrées varient considérablement, l'objet pouvant être vu comme un processus, comme une donnée passive ou encore comme l'encapsulation de plusieurs processus et des données associées. L'auteur propose de distinguer les approches orthogonales de celles non orthogonales. Dans le premier groupe, les objets sont sans rapport avec la concurrence, la gestion de cette dernière est entièrement à la charge du programmeur. Dans les langages non orthogonaux, les objets sont associés à la gestion de la concurrence, par exemple en protégeant leurs données d'accès simultanés. L'existence de cette protection, pour l'ensemble des objets ou seulement pour certains objets, permet d'introduire une nouvelle dichotomie : les approches uniformes (il n'existe qu'une seule catégorie d'objets) et les approches non uniformes (il faut distinguer les objets protégeant leurs données des autres). Un dernier critère de classification consiste à distinguer les langages pour lesquels l'exécution concurrente s'exprime naturellement par l'interaction des objets (approche intégrée) et ceux pour lesquels un autre concept est introduit (par exemple celui de processus ou d'activité).
Un autre domaine d'investigation parmi les langages non orthogonaux est la structure des objets actifs, c'est-à-dire le nombre de processus pouvant être simultanément actifs au sein de l'objet ainsi que leurs modes de création et de synchronisation. Liskov *et al.* [Liskov *et al.*, 1986] distinguent les langages pour lesquels un nombre fixe de processus peuvent s'exécuter de manière concurrente au sein d'un objet (structure *statique*) de ceux ou ce nombre est variable (structure *dynamique*). Wegner [Wegner, 1987], lui s'intéresse à l'ordonnancement des processus au sein d'un objet et il distingue les objets sérialisant l'exécution des méthodes (objets *séquentiels*), ceux entrelaçant les exécutions (objets *quasi-parallèles*) et ceux permettant une exécution concurrente (objets *concurrents*).

Mais le paradigme objet n'a pas résolu l'ensemble des problèmes liés au développement des systèmes parallèles et la recherche dans ce domaine est très active. De cette importante activité découlent de nombreux modèles de programmation et une pléthore de langages prototypes. Le nombre d'extensions parallèles de langages séquentiels est également important, lors d'une étude [Furmento *et al.*, 1995], nous avons par exemple recensé plus de 22 extensions du langage C++. Une telle variété ne facilite pas la tâche du programmeur et nuit à la diffusion des programmes. Si aucun langage pour la programmation de systèmes distribués ne s'est imposé dans l'industrie, on peut néanmoins noter les efforts récents de standardisation d'architectures logicielles (CORBA dans le monde Unix et OLE chez Microsoft) et l'émergence de nouveaux langages orientés "*internet*" (par exemple : Java [Flanagan, 1996], Phantom [Courtney, 1995] et Python [Rossum, 1994]).

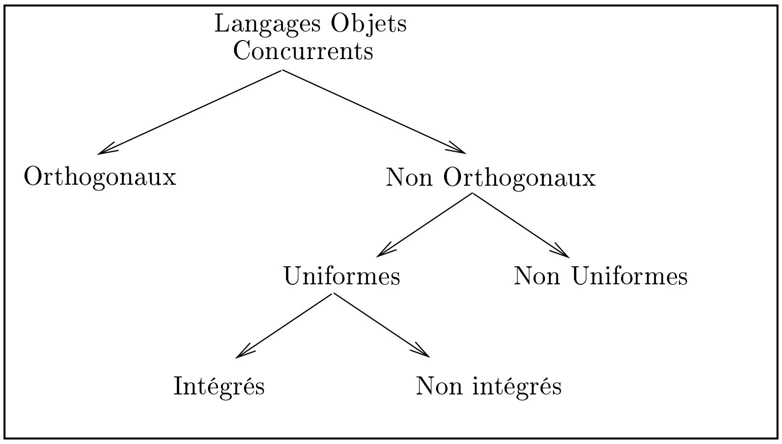


Fig. 4.1 – *Classification des langages objets concurrents ([Papathomas, 1990])*

4.3 La programmation objet dans le cadre de la simulation

La simulation et la programmation objet ont une longue histoire commune. Le premier langage objet, Simula-67, est à l'origine un langage de simulation [Dahl et Nygaard, 1966]. Plus récemment, Stroustrup conçut les principaux mécanismes du langage C++, alors qu'il travaillait sur un simulateur de programmes distribués [Stroustrup, 1994]. Dans la suite de cette section, nous présentons les principaux apports de la technologie objet au domaine de la simulation à événements discrets. Nous en présentons ensuite les limitations majeures et nous terminons sur quelques axes de recherche.

4.3.1 Apports de la technologie objet

Les avantages de la modélisation et de la programmation objet, dans le cadre de la simulation, ont été étudiés par de nombreux auteurs ([Beaumariage et Mize, 1990, Bezivin, 1987, Delcambre et Pollacia, 1994, Egdorf et Painter, 1994, Gavrilov et Vinogradov, 1995, Hill, 1994]) et la conférence annuelle *Object Oriented Simulation Conference* est entièrement dédiée à ce thème.

La conception objet et la modélisation d'un système physique sont des processus très similaires, tout deux partent du principe que l'étude d'un système doit être basée sur ses composantes. Les deux décomposent le problème en un ensemble d'entités autonomes, communiquant par messages. Chaque entité est responsable de l'accomplissement de certaines actions qui dépendent de son état et de son comportement. Comme le fait remarquer Meyer "*Pour modéliser le monde réel en vue de le simuler, quoi de mieux que de décrire les objets à simuler*" ([Meyer, 1988] p. 78). L'utilisation d'un langage objet pour la programmation de simulations est alors très naturelle dans la mesure où il y a correspondance entre objets du monde réel, objets modélisés et simulés et objets du langage de programmation. Le programmeur de simulations bénéficiera aussi des nombreuses propriétés de ces langages :

– *Maintenance et mise-à-jour aisées.* Un projet de simulation évolue généralement dans le temps car les besoins et le système simulé peuvent changer. Le changement peut aussi provenir des objectifs du projet qui évoluent. Une bonne conception et réalisation objets faciliteront cette évolution.

- *Abstraction.* La correspondance évoquée précédemment entre objets du monde réel, objets modélisés et simulés et objets du langage de programmation est un premier facteur d'abstraction. Un second facteur provient du mécanisme d'encapsulation apporté par l'approche objet. Il est alors possible de simuler des modèles de plus en plus grand et de plus en plus complexe.
- *Réutilisabilité.* Elle permet de réutiliser les anciennes simulations afin de les faire évoluer, ou encore de réutiliser des composantes dans de nouvelles simulations.

Mais il existe des différences importantes entre la programmation objet et la simulation. Trois concepts caractérisent ces différences : les entités, les événements et le temps de simulation.

Dans une simulation, certaines entités sont dynamiques, et s'exécutent de manière indépendante et concurrente. L'approche objet n'intègre pas nécessairement le concept d'entité active ou d'objet actif. Notons que ce concept existait déjà dans Simula, via le mécanisme de co-routine, justement pour les besoins de la simulation. Ce mécanisme a ensuite disparu des langages descendants de Simula pour être réintroduit récemment dans Java.

Un événement représente un changement d'état d'une ou de plusieurs entités du système simulé et se produit à une date précise. Un événement peut être interne à une entité ou faire intervenir plusieurs entités en les synchronisant ou en transmettant de l'information. Dans la programmation objet, les instances interagissent par échange de messages. Cet échange prend généralement la forme d'un *appel synchrone de méthode* dans lequel l'appelant reste bloqué jusqu'à ce que la méthode appelée ait terminé son exécution.

Les entités du programme de simulation évoluent dans le temps de simulation. Ce temps logique est défini par le modélisateur et synchronise l'ensemble des actions. A une même date, plusieurs actions peuvent avoir lieu. La programmation objet n'intègre pas directement les notions de temps et de durée qui doivent être introduites via des structures et des traitements appropriés. De plus, la nature synchrone des appels sérialise les traitements complexifiant la mise en œuvre des actions concurrentes.

4.3.2 Limitations actuelles

Il est important de souligner que la programmation objet, si elle accélère les phases de conception, de codage et de mise au point, peut éventuellement apporter un surcoût à l'exécution. Ce surcoût peut provenir soit du modèle objet, qui favorise par exemple une profondeur très importante d'emboîtement lors des appels de méthodes, soit de son implémentation. Doyle [Doyle, 1990] compare quantitativement sept langages objets, il ressort que Smalltalk est le plus lent principalement à cause de son approche homogène (toute entité est un objet, communication uniquement par message) et de son mécanisme de recherche dynamique des méthodes appelées. Beaumariage *et al.* [Beaumariage *et al.*, 1991] signalent qu'une simulation codée dans un langage objet peut être jusqu'à six fois plus lente que si elle avait été codée avec un langage procédural. Selon eux, un surcoût important provient de la gestion mémoire associée à la multiplication de petits objets à durée de vie très brève.

Il est aussi important de noter que l'utilisation d'un langage de haut niveau, comme c'est le cas avec un langage objet non dédié, peut souvent complexifier la tâche de vérification du programme de simulation.

4.3.3 Axes de recherche

Un premier thème de recherche particulièrement actif est celui qui consiste à augmenter à la fois la puissance d'expression et le degré d'abstraction des langages de simulation. Ce double objectif passe par une meilleure intégration des deux domaines (programmation objet et simulation à événements discrets). L'augmentation de l'expressivité est motivée par les nouveaux besoins en simulation comme la simulation mixte, la simulation hétérogène, etc.

Si dès l'origine, la programmation de simulateurs séquentiels a fait appel à la technologie objet, la mise en œuvre de simulateurs distribués a, elle, très rarement bénéficié des avancées de la programmation objet répartie. Cette faible exploitation des résultats de la recherche peut, en partie, s'expliquer par le relatif cloisonnement des communautés et par le fait que la plupart des langages parallèles restaient des prototypes de laboratoire. Mais la croissance des besoins de simulation de systèmes de plus en plus grands et de plus en plus complexes, les limites de capacité et de vitesse des outils classiques de simulation, et enfin une certaine maturité et standardisation des techniques de programmation parallèle ont pour conséquence de rapprocher l'ensemble des communautés (cf. Fig. 4.2).

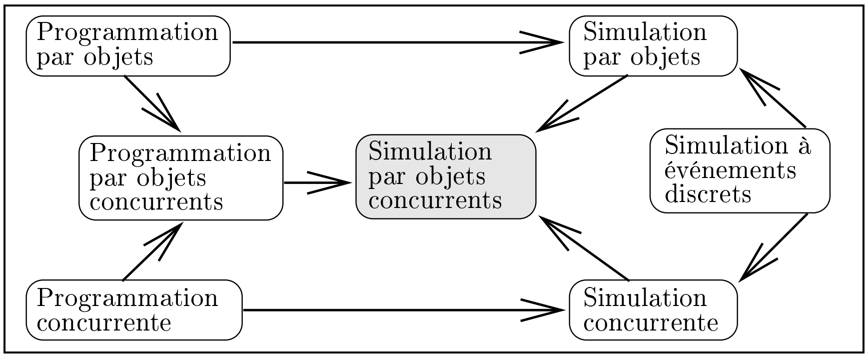


FIG. 4.2 – *Intégration des concepts (tiré de [Bezivin, 1992])*

5. OBJECTIFS DÉTAILLÉS DE NOS TRAVAUX

Nous détaillons et nous motivons dans ce chapitre, les objectifs que nous nous sommes fixés. Ce chapitre peut être vu comme le cahier de charges de l'environnement de simulation Prosit développé dans le cadre de notre thèse.

5.1 Caractéristiques générales de l'environnement de simulation

Le thème central de notre recherche est la conception et le développement d'un environnement de programmation pour la simulation à événements discrets. Nous ne désirons pas fournir un outil capable de lire un modèle en entrée et de fournir des résultats en sortie, mais plutôt une boîte à outils composée de librairies permettant de construire un programme de simulation. Ce programme intégrera le modèle, et son exécution correspondra à la simulation de ce dernier. Une des orientations principales concerne donc la volonté d'utiliser le même langage pour le codage du simulateur et la description des modèles. On se refuse ainsi à définir un nouveau langage pour la définition des modèles qui serait ensuite compilé et/ou interprété lors de la simulation. L'objectif étant ici de fournir à l'utilisateur toute la puissance du langage de programmation pour le codage de ses modèles, et une entière liberté d'interface avec du code ou des outils préexistants.

Le succès de l'environnement ne sera possible que s'il apporte une grande flexibilité au niveau de la modélisation et du codage. Toute classe de systèmes à événements discrets doit pouvoir être simulée. Cette flexibilité et ce pouvoir d'expression ne devront pas se faire au détriment de la capacité de concevoir, à terme, une méthodologie pour l'ensemble du cycle de vie d'une simulation. L'environnement devra aussi comprendre divers outils de mise au point des simulations et d'analyse des résultats.

La troisième grande orientation est notre volonté de réaliser aussi bien des simulations séquentielles, que répliquées ou distribuées. Le parallélisme devra être intégré de manière transparente et naturelle à l'environnement. Le passage d'une simulation séquentielle à une simulation distribuée devra se faire sans modification de la modélisation et avec le minimum de modification du code.

Nous considérons le paradigme objet comme le meilleur candidat pour atteindre ces trois objectifs et nous allons chercher à intégrer le moteur de simulation au sein des mécanismes supportant l'approche objet. Cette intégration devrait permettre de masquer au mieux les mécanismes de simulation et de distribution à l'utilisateur final.

5.2 Fonctionnalités d'utilisation

L'environnement comprendra donc un ensemble de librairies constituant le noyau de synchronisation et des classes de base pour la simulation. Autour du noyau s'articuleront divers outils d'analyse et de suivi. Il devra être possible de développer, au-dessus des librairies de base, des librairies de modélisation (file d'attente, réseau de Petri, etc.) et des librairies d'application plus spécialisées (réseau de communication, transport, etc.).

En plus des critères classiques de génie logiciel (robustesse, modularité, etc.) que devra remplir l'environnement, nous attachons une importance particulière à trois points : (1) la capacité à construire des simulations de manière modulaire et hiérarchique, (2) la possibilité d'intégrer au sein d'un même programme plusieurs techniques de simulation, et enfin (3) la spécification d'une méthodologie pour la conception et d'une architecture pour le codage des simulations.

5.2.1 Simulation modulaire et hiérarchique

Nous venons de voir qu'il y aura une utilisation à deux niveaux avec une approche bibliothèque. D'un côté, l'utilisateur final construira sa simulation en instanciant des classes de bibliothèques ou en utilisant des interfaces de plus haut niveau, et de l'autre, le programmeur mettra au point des classes spécialisées et optimisées. L'utilisation de bibliothèques, en encourageant la réutilisation, facilite le codage et permet une architecture modulaire. L'approche modulaire est particulièrement importante lors de l'étude de systèmes complexes, car elle permet entre autres :

- *une utilisation boîte noire.* Chaque module réalise une tâche précise et possède une interface bien définie. La spécialisation des modules facilite leur optimisation ;
- *une validation aisée.* Chaque module peut être testé et validé indépendamment, la validation du programme complet est alors plus facile ;
- *la réalisation de systèmes ouverts.* Un programme de simulation modulaire peut être étendu, modifié et adapté afin de simuler des systèmes plus complexes.

Deux formes de réutilisation doivent être supportées : la réutilisation de classes et la réutilisation de modèles. La réutilisation de classes est la forme la plus simple et consiste à réutiliser de manière individuelle les entités. La réutilisation de modèles est l'étape suivante, car elle consiste à réutiliser un ensemble d'entités reliées par des interactions éventuellement complexes. Dans la réutilisation de classes, la difficulté consiste à faire coopérer les instances d'une classe avec d'autres instances. Dans la réutilisation de modèles, la difficulté est d'un ordre supérieur, car il faut à la fois faire coopérer un ensemble d'instances avec d'autres instances, et intégrer plusieurs schémas de contrôle et d'exécution (échelle des temps, mode de synchronisation, classes et priorités des clients, etc.).

La réutilisation de modèles est un aspect particulièrement important, car elle permet de construire les programmes de simulation de manière hiérarchique. Une construction hiérarchique apporte :

- *une facilité de conception.* La plupart des systèmes réels sont de nature hiérarchique, la modélisation et le codage sont alors naturels et aisés ;
- *la "scalabilité".* Il est possible de construire des simulations de plus haut niveau à partir de composantes. Ces composantes pouvant être des composantes de base fréquemment utilisées, des composantes spécifiques particulièrement optimisées, ou encore un système complexe ;
- *l'identification et la classification.* A un niveau donné de la hiérarchie, il est possible d'isoler le comportement spécifique d'un composant (par exemple markovien, forme produit, linéaire, etc.) ce qui facilite le codage et permet diverses optimisations.

5.2.2 Simulation mixte

Un autre objectif important de l'environnement sera de permettre le développement de simulateurs mixtes. Par simulation mixte, nous entendons l'intégration au sein d'un même programme de simulation de plusieurs techniques de gestion du temps et de contrôle de la dynamique du modèle (par exemple : simulation discrète/continue, simulation à événements/équationnelle, etc.). Une simulation mixte peut aussi consister à faire interagir au sein d'une même simulation des entités modélisées selon des formalismes différents (par exemple par des files d'attente et des réseaux de Petri). La réalisation de simulations hétérogènes répond à un double objectif : faciliter le codage en utilisant le paradigme le mieux adapté à une composante, et réduire les temps d'exécution.

5.2.3 Méthodologie de conception et architecture de codage

Nous avons présenté à la section 4.1.2 la notion de framework qui constitue, en terme de réutilisabilité et de facilité de codage, l'approche logicielle la plus adaptée à nos objectifs. L'environnement Prosit se présentera donc sous la forme d'un framework dédié à la simulation à événements discrets. Les principaux avantages de cette approche sont :

- une architecture prédéfinie permettant de construire très rapidement des systèmes opérationnels. Le framework réduit les phases de codage, de débogage et de mise au point ;
- un cadre permettant à la fois le développement d'une industrie de composants logiciels et la possibilité d'évaluer et de comparer les propositions issues des travaux de recherches. Une fois le framework entièrement défini, l'intégration de nouvelles composantes est aisée ;
- une maintenance réduite. Le framework fournit la plus grande partie de l'architecture ; le code utilisateur est réduit et localisé, et donc plus facile à tester. Grâce à l'héritage, l'ajout d'une nouvelle fonctionnalité au framework est immédiatement accessible à toutes les classes.

Mais une utilisation optimale du framework nécessite la mise en place d'une aide méthodologique à la conception et à la programmation des simulations. Afin de s'adapter aux catégories d'utilisateurs, trois niveaux seront nécessaires : (1) utilisateurs de librairies spécialisées, (2) programmeurs de librairies et enfin (3) programmeurs d'extensions au framework.

Parallèlement aux composantes spécialisées, nous envisageons de fournir différents *design patterns* résolvant un problème de modélisation et de simulation spécifique, mais se retrouvant dans plusieurs domaines. Nous pourrions avoir un pattern pour l'asservissement de n entités, un pattern pour la détection de propriétes globales, etc.

5.3 Support d'exécution et implémentation

Afin de favoriser l'utilisation et la diffusion de l'environnement, les simulations distribuées auront comme support d'exécution un réseau de stations de travail ou d'ordinateurs personnels. Le parc de machines sera éventuellement hétérogène et faiblement couplé. Nous nous plaçons ici dans la tendance actuelle du parallélisme virtuel qui consiste à préférer un réseau de machines, encore appelé *machine virtuelle* ou *super-calculateur virtuel*, à un super-calculateur beaucoup plus coûteux. L'hétérogénéité de l'environnement demande une implémentation aisément portable. Portabilité qui ne devra pas être obtenue au détriment des performances.

5.4 Aspects novateurs

Les aspects novateurs de nos objectifs comportent deux axes :

- **Structure**. Nous cherchons à développer une boîte à outils permettant la simulation de toute classe de systèmes à événements discrets. Cet environnement possédera plusieurs niveaux d'utilisation et chaque composante pourra être spécialisée ou redéfinie. La réutilisabilité est pour nous un élément fondamental. Le modèle de programmation devra posséder une grande puissance d'expression tout en permettant un codage simple et aisé. C'est pourquoi l'abstraction des mécanismes de simulation constitue un thème important de nos recherches. Cette abstraction passe, selons nous, par l'intégration du moteur de simulation au sein des mécanismes objets.
- **Fonctionnalités**. L'environnement intègrera des fonctionnalités avancées comme la distribution, le hiérarchisation et l'hétérogénéité des modèles, et ce, sans remettre en question les objectifs précédents. Un élément important de nos recherche consistera à définir un modèle de conception et de programmation qui supportera de manière naturelle et sans rupture ces fonctionnalités.

Nous présentons au chapitre 10 une comparaison détaillée de notre modèle de programmation avec d'autres paradigmes de programmation et certains environnements de simulation.

5.5 Résumé des objectifs

Le thème central de notre recherche est la conception et le développement d'un environnement pour la programmation de simulations à événements discrets. Cette recherche comporte deux objectifs principaux :

1. l'abstraction des mécanismes liés à la simulation à événements discrets ;
2. l'abstraction des mécanismes liés a la distribution d'une simulation. Nous désirons être en mesure de réaliser des simulations séquentielles, répliquées et distribuées sans modification du modèle et avec le minimum de modification du code.

Afin d'atteindre ce haut niveau d'abstraction, nous avons retenu l'utilisation du paradigme objet et nous allons chercher à intégrer le moteur de simulation au sein des mécanismes supportant l'approche objet.

Au niveau de l'utilisation, nous avons comme objectifs :

– Mise en place d'une architecture de codage sous forme de framework et proposition d'une méthodologie de conception.
– Programmation à trois niveaux : (1) utilisateurs de librairies spécialisées, (2) programmeurs de librairies et enfin (3) programmeurs d'extensions au framework.
– Simulation modulaire et hiérarchique.
– Simulation mixte.

Au niveau de l'implémentation et du support d'exécution :

– Performance des simulations.
– Portabilité de l'environnement et des programmes utilisateurs.
– Utilisation de machines hétérogènes.

Deuxième partie

PROSIT : UN MODÈLE OBJET POUR LA SIMULATION DES SYSTÈMES À ÉVÉNEMENTS DISCRETS

6. SIMULATION À BASE D'OBJETS ACTIFS

Ce premier chapitre de présentation du modèle, introduit et motive les grandes orientations fondamentales adoptées. Après avoir décrit le modèle de programmation, nous abordons le concept d'objet actif et passons en revue ses caractéristiques et ses modes d'interactions. Nous terminons le chapitre par l'étude des objets multi-actifs.

6.1 Modèle de programmation

Présentée au chapitre 2, la gestion du temps et de la dynamique du modèle est centrale à la mise en œuvre d'une simulation, et soulève une première question :

Quelle est l'architecture des modèles de simulation ?

6.1.1 Architecture des modèles de simulation

L'architecture des modèles précise la structure et l'organisation des différentes composantes ainsi que leur mode d'interaction. Elle constitue, pour le programmeur de simulations, à la fois un ensemble de classes à réutiliser, et des règles et spécifications à respecter. Trois parties du système modélisé doivent être représentées :

- le modèle de fonctionnement qui spécifie comment opèrent chacun des objets constituant le système ;
- le modèle d'interaction qui spécifie comment interagissent les différents objets et quel est le cheminement des entités circulant dans le système ;
- le modèle de charge qui spécifie l'ensemble des informations statistiques permettant de caractériser le comportement des entités.

Ces trois parties comportent des données et des traitements différents, qu'il n'est pas toujours facile d'isoler et de transcrire sous forme algorithmique. Néanmoins, nous avons vu précédemment que ces difficultés pouvaient être partiellement résorbées par la modélisation et la programmation objet. L'apport majeur étant la similitude entre les objets physiques, les objets modélisés et les objets du langage de programmation. Nous conservons donc cette représentation et nous avons :

Toute entité simulée est représentée par un objet instance d'une classe.

Une fois retenue cette approche, qui est naturelle dans le cadre de la programmation objet, nous devons déterminer le mode d'expression de la dynamique du modèle. Nous avons retenu l'approche orientée processus en faisant appel au concept d'entité active. La notion d'entité active et l'unification *entité simulée/objet du langage de programmation*, nous ammène naturellement à introduire les objets actifs. Ils doivent être vus comme des entités autonomes ayant une activité propre, utilisant des objets passifs, et interagissant avec d'autres objets actifs. Nous reviendrons en détail à la section 6.2 sur les caractéristiques des objets actifs.

Nous avons fait ces choix pour les raisons suivantes :

– l'approche orientée processus nous semble plus naturelle, notamment pour les utilisateurs non spécialistes. Elle est mieux adaptée aux problèmes posés en termes d'activités se déroulant en parallèle, et permet généralement de rester à un niveau logique de description du système simulé ;

– l'organisation d'une simulation comme un ensemble d'objets actifs coopérants est aux niveaux conceptuel et physique similaire à la réalité étudiée ;

– les actions d'une entité, nous parlerons du comportement, sont regroupées. La classe contient à la fois la spécification (composante statique) et le comportement (composante dynamique) de l'entité modélisée. Le code est dans son ensemble plus lisible et plus facile à maintenir.

En conclusion, nous avons :

Une simulation correspond à l'exécution d'un ensemble d'objets actifs.

6.1.2 Gestion du temps

Nous venons de voir que les entités modélisées seraient représentées par des objets actifs. Le comportement et les interactions de ces derniers correspondent aux actions du système physique et doivent se dérouler dans le temps de simulation. Nous devons donc définir :

Comment est géré le temps de simulation ?

Le temps de simulation synchronise l'ensemble des objets actifs et sa valeur est une grandeur logique définie par le modélisateur. Nous désirons une gestion automatique, transparente et naturelle du temps. Ces trois critères sont particulièrement importants, car ils apportent un haut niveau d'abstraction et permettent un codage aisé ainsi qu'une réduction du risque d'erreur.

Pour atteindre ces objectifs, nous associons à chaque objet actif une *horloge* indiquant sa date de simulation courante. La progression des horloges ainsi que le maintien de la cohérence temporelle entre les objets actifs sont automatiquement gérés par le système. Le maintien de la causalité consiste à ordonnancer les objets actifs en fonction de leur date. Le temps est représenté par la classe `Time` et les horloges par la classe `Simulation_Clock` (voir l'exemple de code 6.1). Ces deux classes définissent l'ensemble des primitives de manipulation nécessaires. Une entité peut consulter sa date courante ou attendre un laps de temps. Nous avons :

> Les objets actifs évoluent de manière concurrente dans le temps simulé. Ils disposent des primitives suivantes :
>
> – `Time get_time()`, qui retourne la date courante de simulation ;
>
> – `wait(Time)`, qui permet d'attendre un laps de temps.

Exemple de code 6.1 La classe Simulation_Clock (partielle)

```
class Simulation_Clock {
private:
   Time value;
public:
   void reset(); // remise a zero
   void set(Time t); //  maj de l'heure
   void forward(Time t); // avancer l'heure
   void backward(Time t); // reculer l'heure
   Time get(); // heure courante
   // Operateurs de comparaison
   bool operator==(const Simulation_Clock &other_clock);
   bool operator>(const Simulation_Clock &other_clock);
   ...
}; // end Simulation_Clock
```

6.2 Caractéristiques des objets actifs

Dans la section précédente, nous avons présenté le modèle de programmation. La partie centrale de ce modèle est la notion d'objet actif que nous détaillons ici.

6.2.1 Définition et création d'un objet actif

La définition d'un objet actif se fait au niveau de sa classe par le biais du mécanisme d'héritage. La notion d'objet actif est représentée par une classe particulière : la classe `Simulation_Obj`. Nous appellerons classe active une classe qui hérite directement ou indirectement de la classe `Simulation_Obj`. Une classe active décrit les caractéristiques et le comportement des entités instances.

> Un objet actif est l'instance d'une classe qui hérite directement ou indirectement de la classe `Simulation_Obj`.

Associer le concept d'objet actif à une classe et utiliser le mécanisme d'héritage présentent de nombreux avantages. On pourra par exemple doter les objets actifs de comportements par défaut ; on pourra aussi définir des classes de comportements. Nous présenterons ces possibilités ultérieurement.

Un objet actif est créé via le mécanisme classique d'instanciation. Dès sa création, l'objet est pris en charge par le système.

6.2.2 Comportement d'un objet actif

Nous avons indiqué le mécanisme de définition d'une classe active, nous devons maintenant indiquer :

Comment spécifier le comportement d'un objet actif?

Le comportement d'un objet actif regroupe les actions effectuées par l'entité au cours de sa vie. Pour le codage des comportements, le programmeur doit disposer de toute la puissance du langage de programmation. Une classe d'objets actifs doit aussi pouvoir hériter du comportement défini dans une classe ancêtre, ou encore redéfinir un comportement spécifique. C'est pourquoi :

Le comportement d'un objet actif est spécifié dans la routine `behave()` de sa classe.

A sa création, l'objet actif est pris en charge par le système mais n'exécute pas encore le code associé à son comportement. L'activation de l'objet se fait au cours de la simulation à la date spécifiée par le programmeur. La date d'activation est indiquée à la création de l'objet sous la forme d'un paramètre au constructeur.

Durant sa vie, un objet actif peut se mettre en attente passive (primitive `sleep()`). Il sera alors bloqué en attente de réactivation par un autre objet actif. Le mécanisme d'attente passive est particulièrement important et permet à un objet, n'ayant provisoirement plus rien à faire, de se mettre en attente afin de ne pas consommer de ressources (essentiellement CPU).

L'exemple de code 6.2 propose une version simplifiée de la programmation du problème des philosophes.

6.2.3 Terminaison et destruction d'un objet actif

Un objet actif est considéré comme terminé quand son comportement se termine. Deux formes de terminaison sont possibles, la terminaison normale ou encore implicite, qui correspond à la fin de la routine `behave()` et la terminaison prématurée, ou encore explicite, qui doit être demandée par l'objet à l'aide de la primitive `terminate()`. Une fois terminé, l'objet peut être détruit au sens du *système*, i.e., libération de la mémoire, appel d'un destructeur, etc.

Exemple de code 6.2 Un exemple de classe active : une version des philosophes

```
class Philosophe : public Simulation_Obj {
private:
  void behave() {
    for (;;) {
      if(get_time()>1000) terminate();
      if (fourchettes_gauche_et_droite_libres()) {
        prendre_fourchettes();
        wait(random()); // Mange
        rendre_fourchettes();
        wait(random()); // Pense
        }
      else
        wait(random()); // Attente, risque de famine !
      // endif
    }
  }
  ...
}; // end Philosophe
```

6.2.4 Interactions entre objets actifs

Nous venons de détailler le comportement des objets actifs, nous devons maintenant spécifier :

De quelle manière interagissent les objets actifs?

Avant de répondre à cette question, nous devons recenser et préciser en quoi consistent ces interactions. Nous pouvons distinguer quatre formes d'interactions :

- *la création d'objets actifs.* Durant la simulation, un objet actif peut dynamiquement en créer d'autres. C'est par exemple ce qui se passe lors de la simulation d'un écosystème ;
- *la destruction d'objets actifs.* De manière symétrique à la création, un objet actif peut en détruire d'autres. La destruction peut correspondre à un comportement du système physique (simulation d'un champ de bataille) ou à un artefact de programmation (un objet source a généré tous ses clients et, est donc inutile) ;
- *la demande de service.* Lors d'une demande de service, un objet actif client demande à un objet actif serveur d'effectuer une action. La notion de service introduit celles d'identifiant et de code de service. Le code de service est à distinguer du comportement de l'objet serveur. Le comportement du serveur détermine quand et comment servir les clients, le code du service spécifie les actions à effectuer au cours du service ;
- *la synchronisation.* La synchronisation est le mécanisme par lequel deux ou plusieurs objets actifs peuvent coordonner leur exécution.

La création et la destruction dynamiques d'objets actifs sont souvent indispensables et d'une manière générale facilitent la programmation. La création d'un objet actif se fait par le biais du mécanisme standard d'instanciation et a déjà été présenté. La destruction peut être vue comme une demande du service destruction. De manière similaire, la synchronisation peut se faire par échange de demandes de service ou encore à l'aide d'un objet actif de synchronisation. Nous pouvons alors considérer que les objets actifs interagissent uniquement par le biais de demandes de service. Cette uniformisation des interactions simplifie le modèle et facilite sa mise en œuvre et son utilisation. Nous avons alors :

Les objets actifs interagissent par le biais de demandes de service.

Une demande de service intervient dans le temps de simulation, à chaque demande est associée une date. Nous distinguons les demandes bloquantes de celles non bloquantes. Lors d'une demande bloquante, le demandeur ne peut poursuivre tant que le service n'est pas terminé, dans le cas non bloquant, il peut poursuivre, dès réception de la demande par l'objet serveur. Seuls les services bloquants pourront retourner un résultat. Dans notre modèle de programmation, nous considérons que tout système discret peut être modélisé par un ensemble d'entités interagissant par des demandes, bloquantes ou non, de service.

Nous avons vu précédemment que le comportement d'un objet actif était spécifié dans la routine `behave()` de sa classe. De manière similaire, le code des services sera spécifié dans les autres routines de la classe. Ces routines seront appelées *routines services*. Comme pour le comportement, il sera alors possible d'hériter et de redéfinir des routines services. Le codage des services comme fonction membre amène naturellement à unifier les demandes de service aux appels de méthodes. Ce qui nous donne :

Le code des services est spécifié comme fonction membre de la classe. Une demande de service s'exprime comme un appel de méthode.

L'unification des demandes de service aux appels de méthodes et la gestion automatique du temps ont pour conséquence que les interactions entre objets actifs ont toujours lieu à la date courante. Une demande de service prend un temps nul, c'est le déroulement du service qui peut éventuellement avoir une durée. Il est donc impossible d'émettre une demande de service, même non bloquante, pour une date future. De même, à la fin d'une demande bloquante, le client poursuit son exécution à la date de l'objet serveur en fin de service. Ce mode de fonctionnement est conforme aux systèmes physiques, le modèle de programmation est donc plus facile à assimiler et à utiliser.

Nous donnons à l'exemple de code 6.3, la déclaration d'une classe active simulant un guichet de banque.

6.2.5 Résumé du cycle de vie d'un objet actif

Nous pouvons maintenant passer en revue le cycle de vie complet d'un objet actif (voir Fig. 6.1). Le système associe un statut à chacun des états possibles. A sa création, l'objet a le statut *initialisé*, il passe au statut *actif* dès sa prise en charge par le système. Lors d'une

Exemple de code 6.3 Un exemple de classe active : un guichet de banque

```
class Guichet: public Simulation_Obj {
private:
   void behave(); // comportement
public:
   // declaration des services
   Compte *ouvrir_compte(Information *);
   void fermer_compte(Compte *);
   Position *demander_position(Compte *);
}; // end Guichet
```

requête bloquante, il a le statut *bloqué* et le statut *endormi* lors d'une mise en attente passive. Les statuts de terminaison sont *suicidé* et *terminé* qui correspondent respectivement à une terminaison volontaire et à une terminaison normale.

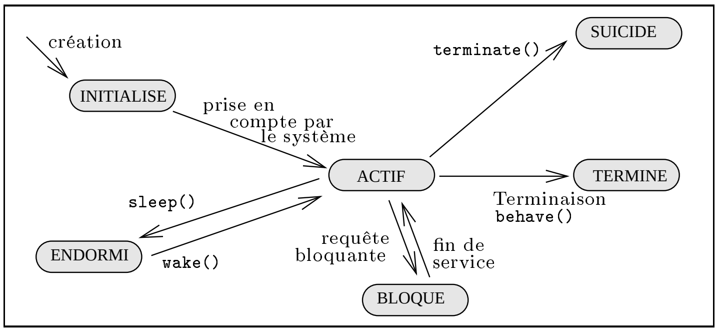


Fig. 6.1 – *Cycle de vie d'un objet actif*

6.3 Mise en œuvre des interactions

Nous nous intéressons de plus près dans cette section à la mise en œuvre des demandes de service. Nous avons décidé précédemment d'unifier les demandes de service aux appels de méthode, mais la notion de service en simulation possède des caractéristiques spécifiques :

– une demande est caractérisée par un double niveau de paramétrage :

   1. niveau modèle, ce sont les paramètres qui seront directement utilisés dans le code du service (ex : nombre d'articles, taille d'un paquet de communication, etc.) ;
   2. niveau contrôle, ce sont les paramètres qui seront utilisés pour le déroulement du service (ex : priorité, temps maximal de service, etc.) ;

– nous avons introduit dans notre modèle, la possibilité d'effectuer des demandes de service de manière bloquante ou non bloquante. Ces deux caractéristiques sont rarement simultanément présentent dans un langage objet où l'échange prend généralement la forme d'un appel de méthode bloquant ;

- une demande peut avoir différents statuts de terminaison : service entièrement, partiellement ou pas du tout exécuté. Ce statut, lors d'une requête bloquante, doit être transmis au demandeur. Le statut de terminaison est différent d'un éventuel résultat du service. Cette différenciation est importante dans le cas ou le service a été partiellement ou pas du tout exécuté, car le résultat est alors indéterminé. Nous parlerons donc de retour de demande de service plutôt que de fin de service ;
- l'objet serveur peut recevoir simultanément plusieurs demandes de service, il peut les servir immédiatement de manière séquentielle ou parallèle, ou encore les stocker en vue d'un traitement ultérieur.

La manipulation explicite des demandes de service (stockage, comparaison, etc.) nécessite de les représenter sous forme d'objets. Une fois les demandes de service représentées par un objet requête et stockées dans une structure accessible, il sera possible de développer des classes de comportement implémentant diverses politiques de service (FIFO, LIFO, priorité, etc.). Le mécanisme à mettre en place doit comporter les éléments suivants :

1. une abstraction des requêtes et de leurs services,
2. une liste des requêtes accessible par le programmeur,
3. une abstraction des routines.

Un tel mécanisme n'est pas propre à la simulation et correspond à la notion de *réification des messages* dans les langages objets. Ferber décrit dans [Ferber, 1989] ce mécanisme qu'il nomme *reification of communication* et qui est une forme de réflexivité [1] d'un langage. Lors de la réification des communications, l'appel d'une méthode est transformé en la création et l'émission d'un objet message. L'objet message est *livré* à l'objet serveur via une primitive de réception qui elle n'est pas réifiée afin d'éviter une réification infinie.

Nous allons exploiter ce mécanisme en enrichissant les objets requêtes avec les paramètres propres à la simulation. De manière symétrique, nous allons aussi représenter la terminaison d'une méthode par un objet réponse. Ce dernier contiendra des informations sur la date d'émission, le statut de terminaison du service et l'éventuel résultat.

Les interactions entre objets actifs sont réalisées par un mécanisme de réification. A une demande de service correspond l'émission d'un objet requête. Le retour d'une demande bloquante correspond à l'émission d'un objet réponse.

Avant d'aborder la mise en œuvre de la réification, nous devons nous intéresser aux problèmes posés par la sémantique particulière des demandes de service. Nous devons déterminer

1. La réflexivité (voir [Colnet *et al.*, 1989] p. 495) peut se définir d'une manière générale comme la description du comportement d'un système qui entretient un rapport de cause à effet avec lui-même. Plus précisément, un système réflexif possède une structure, appelée auto-représentation, qui décrit la connaissance qu'il a de lui-même. Un tel système peut alors non seulement répondre à des questions qui le concernent, mais également s'auto-modifier. Il a toujours une représentation précise de lui-même et son comportement est en accord avec cette représentation.

comment exprimer la nature bloquante et non bloquante des demandes de service, et aussi comment introduire syntaxiquement le double niveau de paramétrage des requêtes. Nous devons enfin sélectionner le mode de traitement du statut de retour d'une demande de service.

6.3.1 Prise en compte de la sémantique d'une demande de service

Nature bloquante et non bloquante des demandes de service

La nature bloquante ou non bloquante d'une demande de service peut être déterminée soit par l'appelant soit par l'objet actif serveur. Nous pouvons aussi envisager une sélection statique ou dynamique du mode d'interaction. Notons que ces choix ne se posent que pour les demandes de service sans valeur de retour. En effet, dans le cas contraire, l'appelant attend toujours la réponse (cf §6.2.4).

Nous avons retenu une sélection par l'appelant du mode bloquant ou non-bloquant d'une demande de service. Ce choix est motivée par les raisons suivantes :

- le protocole d'interactions est simplifié et peut être optimisé. Lors d'une sélection par l'objet serveur, un mécanisme de *rendez-vous* et d'analyse de la requête doit être mis en place. Durant cette phase, le client est en attente, même lors d'une demande non bloquante ;
- cette approche correspond bien au modèle d'objet actif autonome sélectionnant le mode de demande du service en fonction de son contexte, de son comportement, etc.

Nous avons aussi retenu une sélection dynamique du mode d'interaction. Nous considérons que cette approche offre une plus grande souplesse de programmation notamment lors de la réutilisation de classes.

Double niveau de paramétrage

Le double niveau de paramétrage d'une requête a pour objectif de dissocier la composante contrôle du service, qui fait partie du comportement de l'objet, du service lui même qui peut être programmé par l'utilisateur final. Cette séparation devrait faciliter la mise en œuvre ainsi que la maintenance du code, et surtout permettre la réutilisation de classes de comportements. Le double niveau de paramétrage peut être pris en compte de différentes manières. Nous passons en revue trois approches en insistant sur les avantages et les inconvénients de chacune.

Paramètre supplémentaire La première solution possible est celle qui consiste à regrouper les paramètres de contrôle dans un objet, et de passer celui-ci comme dernier (ou premier) paramètre à l'appel de la routine service.

```
Control_Parameters *ctrl = new Control_Parameters;
ctrl->set_priority(1);
ctrl->set_sync();
...
caisse->payer(15,ctrl);
```

C'est une approche couramment utilisée dans la programmation distribuée. Son principal inconvénient vient du fait que l'objet contrôle se situe au même niveau que les paramètres du service alors qu'il n'est pas exploité dans le code du service mais par le comportement de l'objet.

Attribut de l'objet actif La seconde possibilité est une variante de la précédente et consiste à mettre en place un attribut contrôle qui est automatiquement transféré par le mécanisme de réification.

```
Obj_Ctrl->set_priority(1);
Obj_Ctrl->set_sync();
...
caisse->payer(15);
```

Cette approche permet de configurer en une seule fois plusieurs demandes de service, mais a comme inconvénient majeur de séparer la configuration, de l'appel lui-même. Cette non-atomicité peut poser des problèmes de lisibilité et de robustesse de code.

Extension syntaxique Cette dernière méthode a pour objectif de pallier aux faiblesses de la précédente. Elle repose sur une extension de la syntaxe d'appel C++. Un appel de service est toujours préfixé du mot clé `sync` ou `async`, indiquant la nature bloquante ou non de l'appel. Suite à l'appel de méthode, viennent les paramètres de contrôle sous une forme qui est proche du codage des constructeurs.

```
sync caisse->payer(15) :priority(1);
...
sync porte->passer();
```

Le principal inconvénient de cette approche est l'extension de la syntaxe C++ et le besoin d'un pré-processeur ou d'un compilateur spécifique. En introduisant une extension spécifique à la simulation, nous perdons une partie de l'abstraction.

Statut de terminaison d'un service

En cas de non-terminaison d'un service, la valeur de retour peut être prise en compte de différentes manières :

- une valeur spécifique peut être retournée, comme par exemple `-1` ou `null`. Cette solution, très utilisée, est contraignante et pas suffisamment générale. En effet, tout service doit alors avoir une valeur de retour qui doit toujours être de même nature (par exemple entier ou pointeur). De plus, à moins de distinguer plusieurs valeurs de retour, il est impossible de connaître la cause de non-terminaison ;
- les différents cas de non-terminaison peuvent être différenciés par la mise en place d'une variable contenant le statut de terminaison de la dernière demande de service. Cette technique est par exemple utilisée sous Unix lors des appels systèmes (variable `errno`). Cette solution n'est pas très robuste, car si l'utilisateur ne teste pas systématiquement le statut, il risque d'accéder à un résultat non initialisé ;

– une dernière approche consiste à faire appel au mécanisme d'exception C++. Lors d'une non-terminaison d'un service, une exception est levée dans le contexte du demandeur. L'utilisation des exceptions est intéressante car elle garantit que le programmeur a bien prévu le code de traitement lié à la non-terminaison du service.

Approche retenue pour Prosit

Nous avons retenu pour Prosit l'extension syntaxique afin de résoudre le problème du double niveau de paramétrage. Pour celui de la prise en compte du statut de terminaison d'un service, nous avons adopté l'utilisation des exceptions. Nous considérons que ces deux mécanismes offrent un bon niveau d'abstraction et favorisent un code robuste. Nous illustrons à l'exemple 6.4 une demande de service avec prise en compte du statut de terminaison.

Exemple de code 6.4 Demande de service avec prise en compte du code de retour

```
try {
   sync station1->service(param) :priority(20);
}
catch (Service_Exception &exception) {
// traitant de non-terminaison
}
```

6.3.2 Mise en œuvre de la réification

Nous présentons maintenant le fonctionnement du mécanisme de réification adopté. Ce mécanisme comprend quatre étapes qui correspondent à quatre points de réification (cf. Fig. 6.2). Chacun de ces points peut être surchargé par le programmeur.

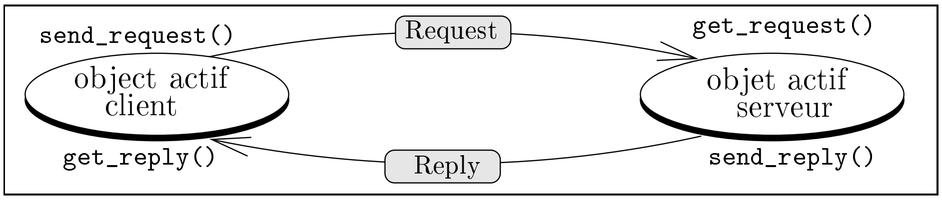


Fig. 6.2 – *Mécanisme de réification*

Emission de la requête Une fois l'appel de service réifié en objet requête, la fonction `send_request()` du client est appelée avec la requête en paramètre. Cette fonction complète l'initialisation de la requête et la transmet. La requête est une instance de la classe `Request` (voir l'exemple de code 6.5). Cette classe hérite de la classe `Temporal_Obj` et possède de ce fait une horloge qui indique la date d'émission. L'attribut `mid` représente la fonction à exécuter (une instance de la classe `MID` représente un identifiant de fonction membre), `params` contient la liste des paramètres effectifs. La requête possède aussi un pointeur sur les objets actifs source et destinataire. La routine `transfer()` permet de transférer la requête.

Exemple de code 6.5 La classe Request (partielle)

```
class Request : public Temporal_Obj {
private:
    Params *params;
    MID mid;
    Request_Id id;
    Simulation_Obj *source;
    Simulation_Obj *destinataire;
public:
    virtual void transfer();
    virtual bool is_synchronous();
}; // end Request
```

Réception de la requête La requête est automatiquement livrée au serveur via la primitive `get_request()`. Cette fonction est non réifiée et consiste à placer l'objet requête dans la liste des requêtes non encore traitées. Cette structure est accessible au programmeur et possède les opérations classiques de manipulation de liste. C'est au comportement de l'objet de sélectionner et de servir la requête. La classe `Simulation_Obj` fournit des primitives de base pour l'accès aux requêtes (requête la plus ancienne, la plus récente, etc.) ; les librairies pourront fournir diverses politiques de service pré-programmées ainsi que des primitives de plus haut niveau (requête la plus prioritaire, requête portant sur un service spécifique, etc.). Nous détaillons la programmation du contrôle à la section 6.3.3.

Emission de la réponse De manière symétrique à la demande de service, une fois la requête servie, il y a création de l'objet réponse et appel de la primitive `send_reply()` du serveur. Cette primitive se charge d'émettre l'objet réponse. La réponse est une instance de la classe `Reply` (voir l'exemple de code 6.6). Elle contient un pointeur sur le résultat éventuel ainsi qu'un pointeur sur l'exception à lever en cas de non-terminaison. L'attribut `id` contient l'identifiant de la requête initiale. La réponse possède aussi un pointeur sur les objets actifs source et destinataire. La routine `transfer()` permet de transférer la réponse.

Exemple de code 6.6 La classe Reply (partielle)

```
class Reply : public Temporal_Obj {
private:
    Result *result;
    Exception *error;
    Request_Id id;
    Simulation_Obj *source;
    Simulation_Obj *destinataire;
public:
    virtual void transfer();
}; // end Reply
```

Réception de la réponse La réponse est livrée à l'objet client via la primitive `get_reply()`. Cette fonction est non réifiée et consiste à traiter la réactivation de l'objet.

Exemple

Nous donnons ici un exemple de classe active avec surcharge des points de réification. La surcharge permettant de tracer les interactions de l'objet actif.

Exemple de code 6.7 Surcharge des points de réification

```
class Monitoring : public Simulation_Obj {
private:
  void send_request(Request *rq){
    cout << "emission d'une requete par " << *this << " a " << get_time() << endl;
    rq->print();
    Simulation_Obj::send_request(rq);
  }
  void send_reply(Reply *rp){...};
public:
  void get_request(Request *rq){...};
  void get_reply(Reply *rp){...};
}; // end Monitoring
```

6.3.3 Programmation du contrôle

Le contrôle de l'objet actif est explicite et centralisé dans la routine `behave()`. Le programmeur a, à sa charge, les actions propres de l'objet (par exemple une fonction a exécuter périodiquement) et le service des requêtes. Le service comprend la sélection de la requête à servir et l'exécution de la routine associée. La centralisation du contrôle permet d'hériter et de redéfinir le comportement. Le codage de cette fonction doit être particulièrement soigné afin qu'une petite modification du comportement, lors de l'héritage, ne nécessite pas le re-codage complet du *behave()*[2].

La sélection de la requête se fait en parcourant la liste des requêtes. Parmi les critères de sélection, nous avons les paramètres de contrôle (priorité, etc.), les paramètres de service, le service demandé ainsi que les attributs tels que la date de demande, l'émetteur, etc. L'exécution du service, associé à la requête, se fait en utilisant la primitive `direct_exec_request()` qui prend en paramètre la requête. Afin de faciliter le codage de ces différentes tâches, la classe `Simulation_Obj` définit diverses primitives de base. Des fonctionnalités de plus haut niveau pourront être specifiées par les bibliothèques de classes.

Nous donnons dans l'exemple de code 6.8, le comportement d'un objet servant de manière cyclique une requête et consommant une unité de temps. Quand il n'y a pas de requête à servir, l'objet se met en attente passive (il est automatiquement réactivé à l'arrivée d'une requête).

2. Ce problème d'interaction entre la définition du comportement d'un objet actif et la réutilisation par héritage est bien connu des chercheurs de la communauté des langages objets parallèles sous le nom *d'anomalie d'héritage* (voir [Matsuoka et Yonezawa, 1993]).

Exemple de code 6.8 Comportement d'un objet actif

```
void Server::behave() {
  for (;;) {
    while (request_waiting()) {
      Request *rq= request_sorted->first();
      direct_exec_request(rq);
      if (rq->is_synchronous()) send_reply_for(rq);
      delete rq;
      wait(1);
    }
    sleep();
  }
}
```

6.4 Introduction de la multi-activité

Il est très courant en modélisation de rencontrer des entités effectuant plusieurs actions de manière concurrente. Deux techniques de mise en œuvre sont alors envisageables :

- La première technique consiste à représenter l'entité par plusieurs objets actifs, chacun n'exécutant qu'une seule action à la fois. Le nombre d'objets actifs pouvant être fixe ou alors dynamique en fonction de la quantité de travail. Cette approche pose de nombreux problèmes de synchronisation et d'accès aux données communes. Elle n'est pas non plus naturelle dans la mesure où le programmeur doit alors spécifier plusieurs classes actives.
- La seconde technique consiste à coder de manière linéaire dans le comportement de l'objet l'entrelacement des différentes actions. Cette approche passe donc par l'extension du comportement de l'objet actif ce qui complexifie le code le rendant difficile à étendre et à maintenir.

Aucune de ces méthodes n'étant satisfaisante, nous nous posons la question suivante :

Les objets peuvent-ils être multi-actifs ?

Avant de répondre à cette question, nous pouvons faire les remarques suivantes :

- nous avons retenu l'approche orientée processus car elle permettait de ne plus fragmenter le code associé aux événements. Il serait intéressant de conserver cette approche lors de la programmation des routines services et pas uniquement pour les comportements ;
- le concept de multi-activité est une abstraction naturelle à introduire et très puissante.

Nous avons alors décidé d'introduire la notion de multi-activité dans notre modèle (cf. Fig. 6.3). Le comportement d'un objet actif peut créer et contrôler plusieurs activités annexes. Les activités de l'objet se déroulent de manière concurrente dans le temps simulé. Le contrôle des activités annexes, par le comportement de l'objet, permet de programmer une véritable politique de gestion des services.

Un objet actif peut créer et contrôler des activités annexes. Les activités de l'objet se déroulent de manière concurrente dans le temps simulé.

Signalons que l'utilisation d'activités annexes n'est pas obligatoire, le programmeur pouvant utiliser une des deux techniques présentées en début de section.

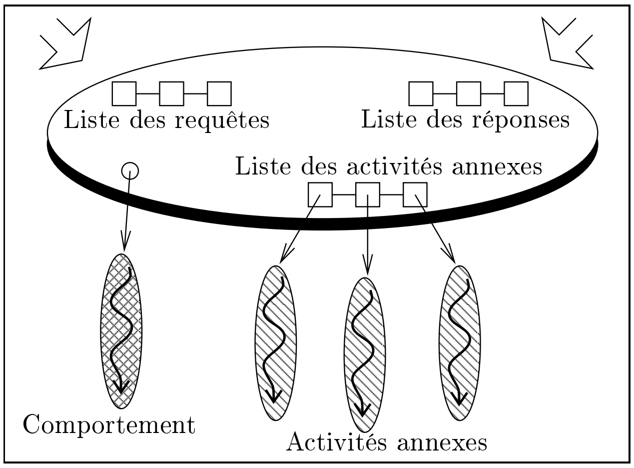


Fig. 6.3 – *Architecture d'un objet actif*

Le comportement d'un objet actif devant explicitement manipuler des activités, nous avons introduit une abstraction adéquate, l'activité, et divers mécanismes de contrôle.

6.4.1 Notion d'activité

Une activité correspond à l'exécution, par un objet actif, d'une de ses méthodes. Une activité se déroule dans le temps de simulation sous le contrôle du comportement. Contrairement à l'activation classique d'une méthode, le déroulement d'une activité n'est pas continu dans le temps réel, mais se déroule sous la forme *d'activation/suspension*. Une activité peut s'arrêter et être réactivée ultérieurement. Une activité s'exécute du point d'arrêt courant jusqu'à arriver à un nouveau point d'arrêt qui devient le point de départ de la prochaine activation (cf. Fig. 6.4). Une fonction service est donc constituée de sections de code algorithmique, qui s'exécutent entièrement, et de points d'arrêt. Comme points d'arrêt, nous avons :

– la consommation de temps,

– la mise en attente passive,

– l'appel bloquant d'un service.

Une activité se termine quand elle a fini d'exécuter le code de la méthode à laquelle elle est rattachée. Elle peut aussi se terminer explicitement en utilisant la primitive `terminate()`.

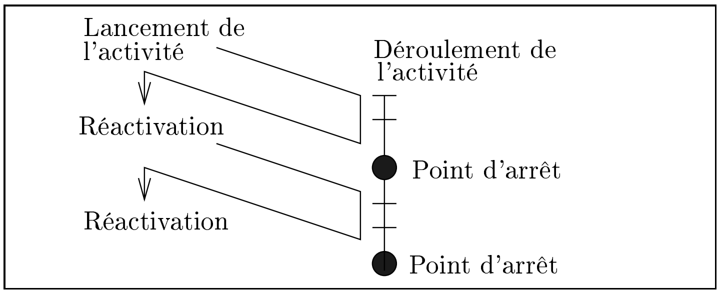


Fig. 6.4 – *Schémas d'exécution d'une activité*

Les activités sont manipulées sous forme d'objets instances de la classe `Activity`. Une activité est donc à la fois une entité du langage de programmation et une abstraction qui permet au comportement de manipuler et de contrôler l'exécution de ses méthodes actives. Au sein d'un objet actif, plusieurs activités peuvent exécuter la même méthode, le code de ces dernières doit donc être conçu de manière réentrante. L'existence de points d'arrêt au sein d'une méthode et la possibilité d'avoir des exécutions concurrentes de la même méthode, complexifie le modèle de programmation basé sur les assertions *pré–post* [Meyer, 1988].

6.4.2 Contrôle des activités

Le comportement de l'objet actif a, à sa charge, le contrôle des activités annexes. Ce contrôle comprend la création et la destruction des activités, et la gestion de leur activation.

Différents modes de création sont disponibles, mais le plus courant consiste à créer une activité afin de servir une requête. Pour cela, le programmeur dispose de la primitive `exec_request()` qui prend en paramètre une requête et retourne une activité initialisée et prête à être activée. A son activation, elle exécutera la fonction service associée à la requête. La destruction d'une activité consiste à ne plus la réactiver et à la détruire au sens du *système* (libération mémoire, appel du destructeur, etc.).

L'activation se fait à l'aide de la primitive `run()`, l'activité s'exécute alors du point d'arrêt courant jusqu'au point d'arrêt suivant. Selon le point d'arrêt rencontré, l'activité change de statut et le comportement de l'objet effectue une des actions suivantes :

– si l'activité s'est terminée, le comportement émet éventuellement l'objet réponse et détruit l'activité ;

– si l'activité a émis une requête bloquante, le comportement l'insère dans la liste des activités bloquées. Quand l'objet réponse arrivera, le comportement se chargera de réactiver l'activité ;

– si l'activité a demandé à consommer du temps, le comportement l'insère dans son échéancier local et se chargera de l'activer à la bonne date.

Dans le cadre de la multi-activité, le comportement de l'objet actif peut être vu comme un *ordonnanceur* d'activités. La progression de son horloge doit être cohérente avec celles des activités annexes.

Nous donnons à l'exemple de code suivant (6.9), le comportement d'un serveur multiple.

Exemple de code 6.9 Comportement d'un objet multi-actif

```
void Server::behave(){
  for (;;) {
    // creation des activites
    while(request_waiting() && idle_server())
      schedule(exec_request(request_sorted->first()));
    // activation des activites
    while (reactivation_needed()) {
      Activity *act = next_activity();
      act->run();
      re_schedule(act);
    }
    // point d'arret de l'objet actif
    if(activities_running())
      wait(compute_interval());
    else
      sleep();
  }
}
```

6.5 Résumé du modèle séquentiel

Cette dernière section résume les grandes orientations adoptées dans ce chapitre. Au niveau du modèle :

- Une simulation correspond à l'interaction, dans le temps de simulation, d'un ensemble d'objets actifs. Le temps de simulation est géré de manière automatique par le système.
- Les objets actifs interagissent par le biais de demandes de service. Une demande peut être bloquante ou non-bloquante, et possède un niveau de paramètrage modèle et un niveau contrôle.
- Le contrôle de l'objet actif est explicite et centralisé. Le programmeur a, à sa charge, la programmation du comportement. Le comportement de l'objet comprend les actions propres, la gestion des requêtes et des services, ainsi que les interactions avec les autres objets actifs.
- Un objet actif peut créer et contrôler des activités annexes. Les activités annexes se déroulent de manière concurrente dans le temps de simulation et permettent d'exécuter les services.
- Nous avons mis en œuvre un modèle objet (cf. §4.2.1) non uniforme (nous distinguons des objets actifs et des objets passifs) au sein duquel l'exécution concurrente s'exprime de manière non intégrée (utilisation de la classe `Activity`). La structure des objets actifs est dynamique (un nombre variable de processus peuvent être simultanément actif au sein d'un objet) et quasi-parallèle (les processus intra-objet entrelacent leur exécution).

Au niveau de la mise en œuvre :

- Une classe d'objets actifs doit hériter de la classe `Simulation_Obj` et spécifier dans la routine `behave()` le comportement des instances.
- Une demande de service s'exprime à l'aide d'une syntaxe proche de l'appel de méthode.
- Un service est une méthode de l'objet serveur.
- Les interactions font appel à un mécanisme de réification. Une demande de service est automatiquement réifiée en un objet requête. De manière symétrique, la fin d'un service bloquant est aussi réifiée en un objet réponse. Chaque objet actif possède une liste des requêtes et une liste des réponses accessibles au programmeur.
- Chaque classe d'objets actifs possède quatre points de réification qui peuvent être redéfinis.

7. EXTENSION DISTRIBUÉE DU MODÈLE DE PROGRAMMATION

Nous poursuivons dans ce chapitre la présentation du modèle de programmation en abordant la composante distribuée de nos travaux.

Un de nos objectifs principaux, consiste à permettre le passage d'une simulation séquentielle à une simulation distribuée sans modification de la modélisation et avec le minimum de modification du code. C'est pourquoi nous avons conçu le modèle distribué comme une extension du modèle séquentiel. L'extension distribuée introduit un minimum de nouveaux concepts qui s'intègrent de manière naturelle au cadre déjà construit. Nous considérons aussi qu'une distribution à la fois optimale, automatique et transparente des objets actifs sur l'ensemble des machines n'est actuellement pas réaliste. C'est pourquoi nous fournissons à l'utilisateur les abstractions lui permettant de spécifier et de contrôler la répartition spatiale de son modèle. Ce contrôle explicite reste relativement simple et les mécanismes introduits permettront dans un second temps de développer diverses politiques de placement et de partage de charge.

Comme dans la version séquentielle, une simulation distribuée correspond à l'exécution d'un ensemble d'objets actifs interagissant dans le temps. Mais dans le cadre distribué, les objets sont répartis sur un ensemble de machines éventuellement hétérogènes. Le modèle de programmation couvre les points suivants :

- vision structurée et abstraite du parc de machines,
- mécanisme de placement des objets actifs et primitives de localisation,
- migration des objets actifs,
- synchronisation temporelle des objets actifs.

Nous détaillons dans les sections suivantes chacun de ces thèmes.

7.1 Architecture distribuée

Nous nous plaçons dans le cadre général d'une architecture distribuée faiblement couplée (cf. Fig. 7.1). Nous considérons un parc de machines hétérogènes, reliées par un réseau de communication. Chaque machine possède sa propre mémoire (il n'y a donc pas de mémoire partagée) et est dotée d'un processeur. Les seules hypothèses sur le réseau sont la possibilité d'avoir des communications point-à-point et l'aspect fiable et FIFO des communications.

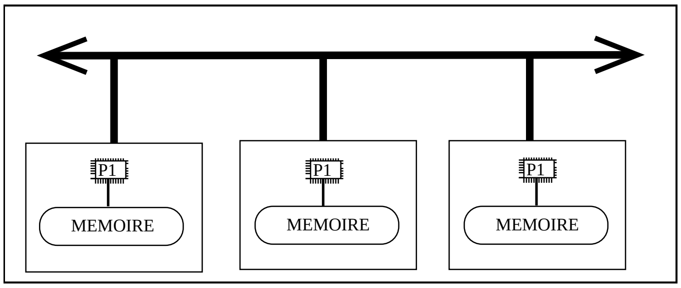


Fig. 7.1 – *Support d'exécution*

La distribution d'une simulation sur un ensemble de machines consiste à répartir les objets actifs composant le modèle. Pour des raisons de performance et de gestion de la synchronisation, nous désirons pouvoir indiquer que des objets actifs partagent le même contexte (mémoire, CPU, etc.). Le programmeur doit être en mesure d'étendre les *nœuds de calcul* utilisables, de créer des objets actifs sur ces nœuds, puis de les manipuler de manière uniforme. Pour cela, nous avons introduit deux composantes : le *computer* et le *cluster* (cf. Fig. 7.2).

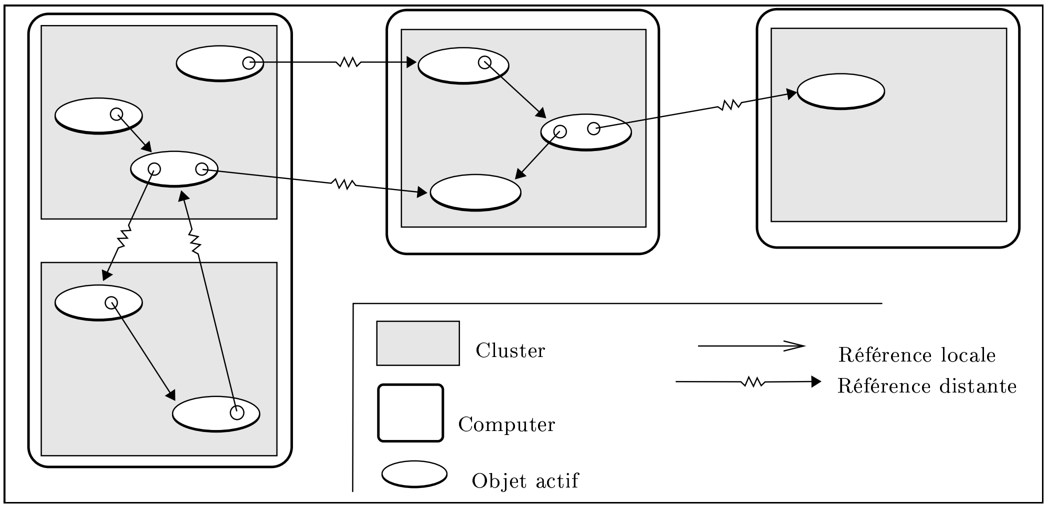


Fig. 7.2 – *Architecture d'une simulation distribuée*

7.1.1 Notion de computer

Un computer représente une machine logique du super-calculateur virtuel. Un computer (voir l'exemple de code 7.1) est caractérisé par un nom physique (nom réseau de la machine), un nom logique (nom symbolique utilisé par le programmeur) et son architecture (type de processeur et de système d'exploitation). De manière dynamique (voir l'exemple de code 7.3), le programmeur peut ajouter ou retirer des machines à sa configuration. Un fichier d'initialisation peut aussi être consulté au début de l'exécution afin de créer automatiquement les premiers computers.

Grâce à la désignation symbolique et au fichier d'initialisation, le nombre et la nature des machines physiques utilisées peuvent facilement être contrôlés et spécifiés au moment de l'exécution. Nous obtenons ainsi une grande souplesse d'adaptation et une bonne portabilité des simulations.

Exemple de code 7.1 La classe Computer (partielle)

```
class Computer {
protected:
   String symbolic_name; // nom symbolique
   String real_name; // nom reel
   Arch arch;// architecture
public: // primitives d'acces
   static Computer *any(); // n'importe quelle machine
   static Computer *best(); // la "meilleure" machine
   static Computer *current(); // la machine courante
}; // end Computer
```

7.1.2 Notion de cluster

Un cluster est l'entité abstraite que manipule le programmeur pour désigner un contexte (mémoire, CPU, etc.) sur un computer. Dynamiquement, le programmeur crée un cluster en instanciant la classe système `Cluster` (voir l'exemple de code 7.2). Le cluster est automatiquement placé sur la machine la "moins" chargée, mais le programmeur peut aussi indiquer la machine cible. Pour cela, il passe comme paramètre à l'allocateur, une instance de `Computer` ou encore le nom symbolique de la machine (voir l'exemple de code 7.3).

Exemple de code 7.2 La classe Cluster (partielle)

```
class Cluster {
protected:
   Cluster_Id *id; // identifiant du cluster
public: // primitives d'acces
   static Cluster *any(); // un cluster quelconque
   static Cluster *best(); // le cluster le moins chargé
   static Cluster *current(); // renvoi le cluster courant
   static List<Cluster>all(); //  tous les clusters
}; // end Cluster
```

Exemple de code 7.3 Utilisation de computers et de clusters

```
   Computer *comp1 = new Computer("m1","uranus.inria.fr");
   Computer *comp2 = new Computer("m2","neptune.inria.fr");
   ...
   Cluster *node1 = new Cluster; //  machine la moins chargée
   Cluster *node2 = new(Computer::any()) Cluster; // n'importe qu'elle machine
   Cluster *node3 = new("m1") Cluster; // la machine de nom symbolique m1
   Cluster *node4 = new(comp2) Cluster; // le computer comp2
```

7.2 Placement et localisation des objets actifs

A la création d'un objet actif, le programmeur précise le cluster cible (voir l'exemple de code 7.4). Si aucun cluster n'est précisé, l'objet est créé sur le cluster courant. En cours d'exécution, le programmeur est en mesure de déterminer si un objet est local ou distant, et dans le second cas de récupérer le cluster sur lequel il est placé.

Exemple de code 7.4 Création d'objets actifs

```
Station *st1, *st2, *st3;

st1 = new Station; // le cluster courant
st2 = new(Cluster::best()) Station; // le meilleur cluster
st3 = new(node2) Station; // le cluster node2
```

Un autre mécanisme important est celui de *binding* qui permet de localiser et d'obtenir une référence (un pointeur) sur un objet actif, quand on ne connaît que sa classe et, son nom ou son identifiant (voir l'exemple de code 7.5). Le binding permet d'obtenir cette référence sans passer par une entité tierce.

Exemple de code 7.5 Localisation d'un objet actif (mécanisme de binding)

```
Porte *entree=Porte::bind(id);
sync entree->passer();
```

7.3 Mécanisme de migration des objets actifs

La migration d'un objet actif consiste à quitter le cluster courant et à poursuivre l'exécution sur le cluster destination. La demande de migration est faite de manière explicite, dans le comportement de l'objet, à l'aide de la primitive `migrate()` (voir l'exemple de code 7.6). Cette opération se déroule dans le temps de simulation et a une durée nulle. L'objet poursuit son comportement, à la même date, en exécutant l'instruction suivant la demande de migration. La migration d'objets actifs est une opération très importante qui permet notamment de gérer efficacement les ressources limitées (mémoire, CPU, réseau), et de réduire les temps d'exécution en déplaçant les objets vers les données manipulées. Dans le cadre de la simulation, la migration permet aussi de réduire et de simplifier la synchronisation. En effet, une demande bloquante de service crée une relation bidirectionnelle entre l'objet client et l'objet serveur (transmission de l'objet requête puis transmission de l'objet réponse). Si ces deux objets sont placés sur des clusters différents, il y a aussi une relation bidirectionnelle entre les deux clusters. Par contre, si l'objet client migre vers l'objet serveur, la relation entre clusters est alors unidirectionnelle.

Exemple de code 7.6 Demande de migration

```
if(st1->is_remote()) migrate(st1->location());
sync st1->service(); // demande locale
```

7.4 Conservation de la causalité

La répartition des objets actifs sur les clusters correspond à une distribution spatiale du modèle. Nous avons présenté au chapitre 3 les principaux protocoles de conservation de la causalité associé à cette approche. Dans notre architecture, deux niveaux de synchronisation sont à distinguer :

- le niveau intra-cluster qui consiste à synchroniser les objets actifs placés sur un même cluster,
- le niveau inter-clusters qui consiste à synchroniser les clusters entre eux, et donc l'ensemble des objets actifs.

Ces deux niveaux peuvent utiliser le même protocole de synchronisation ou des protocoles différents. L'approche framework de PROSIT, permettra d'intégrer et de comparer différents protocoles. Avant de présenter l'intégration de la synchronisation dans le modèle PROSIT, nous discuterons des caractéristiques du modèle de programmation devant être prises en compte.

7.4.1 Spécificité du modèle de programmation PROSIT

Dans les sections précédentes, nous avons présenté et discuté des principaux mécanismes mis en œuvre pour le modèle de programmation PROSIT. Ces mécanismes apportent une grande puissance d'expression et un haut niveau d'abstraction mais ont des répercussions sur la synchronisation. Il est donc important de recenser ces caractéristiques et de bien comprendre leurs conséquences.

Simulation orientée processus Si la simulation orientée processus est plus naturelle au modélisateur, elle complexifie la notion d'état d'une entité, ainsi que les interactions entre elles.

Modèle objet actif Au cours de la simulation, des objets actifs peuvent être créés et détruits. Chaque objet est autonome et détermine le prochain service à demander en fonction de son code algorithmique et des références sur les autres objets actifs. La topologie, statique et fixée en début d'exécution, à la base des protocoles conservatifs n'est pas directement disponible. De même, dans le cadre d'une synchronisation optimiste, l'état d'un objet actif se révèle être à la fois complexe et volumineux. En effet, il comprend :

- les *données membres*, déclarées dans la définition de la classe (ainsi que dans les classes ancêtres) de l'objet actif ;
- les *données propres aux activités*, ce sont les données qui apparaissent dans le code des fonctions exécutées par l'activité (variables locales et paramètres). Ces données sont locales à l'activité ;
- le *contexte d'exécution de l'objet actif et des activités*, ce sont les structures internes au framework.

Il semble alors difficile de réaliser de manière efficace et transparente la sauvegarde et la restauration de l'état d'un objet actif.

Abstraction élevée et réutilisation de code Nous avons ici deux objectifs qui sont à la base de nos travaux. Pour les respecter, les mécanismes de synchronisation devront apporter un minimum de contraintes et demander peu de codage de la part du modélisateur.

Utilisation de clusters Le regroupement de plusieurs entités sur un cluster a pour objectif d'augmenter la granularité des sous-systèmes. Cette approche, si elle peut apporter de meilleures performances, complexifie l'implémentation de la synchronisation en introduisant deux niveaux de synchronisation : intra-cluster et inter-clusters.

7.4.2 Intégration des protocoles dans le framework Prosit

La synchronisation consiste soit à mettre en place un ordonnancement des objets actifs respectant le principe de causalité (approche conservatrice), soit à utiliser un mécanisme permettant de corriger les erreurs (approche optimiste). Ce choix se pose au niveau intra-cluster et au niveau inter-clusters.

Synchronisation intra-cluster

Au niveau du cluster, il est possible de conserver l'ordonnancement séquentiel fondé sur les horloges des objets actifs. Une autre approche consiste à mettre en place un ordonnancement cyclique ou un ordonnancement fondé sur un attribut priorité des objets.

Synchronisation inter-clusters

Au niveau inter-clusters, il est possible de mettre en place un ordonnancement global. Cet ordonnancement peut consister par exemple à utiliser les algorithmes synchrones dirigés par le temps. Cette approche sera réservée aux classes de modèles adaptées (cf. section 3.2.1). Notons que seul un ordonnancement global permet d'avoir une exécution distribuée déterministe.

Une autre possibilité consiste à faire appel aux algorithmes de conservation de causalité vus à la section 3.2. On peut y distinguer deux composantes : le contrôle et les informations nécessaires au contrôle. On peut remarquer que les informations nécessaires (prédiction, topologie, etc.) sont spécifiques aux entités modélisées et ne peuvent donc être fournies que par les objets actifs. Par contre, pour une synchronisation donnée, le contrôle est toujours le même quels que soient les objets actifs présents. Nous pouvons donc envisager la prise en compte du contrôle soit à un niveau supérieur aux objets actifs soit à leur niveau.

Gestion de la synchronisation au niveau de l'ordonnanceur Dans cette approche, l'ordonnanceur est responsable de la synchronisation des objets actifs présents sur son cluster. Comme dans le cadre séquentiel, il se charge d'ordonnancer les objets et de mettre à jour leur horloge. Mais ici, l'activation d'un objet ne se fera que si elle est valide (*i.e.* respecte la causalité) par rapport à la synchronisation en cours. L'ordonnanceur doit donc posséder tous les éléments nécessaires pour prendre cette décision. L'ordonnanceur gère aussi les messages de synchronisation (par exemple les messages nuls) et les données liées à la synchronisation (par exemple les canaux des protocoles conservateurs, les sauvegardes d'états des protocoles optimistes, etc.).

Afin d'assurer la synchronisation, des informations circulent à deux niveaux (cf. Fig. 7.3.a). Au premier niveau qui est interne au cluster, l'ordonnanceur récupère des objets actifs locaux, les données nécessaires à la gestion de la synchronisation. Ces informations peuvent être la valeur de la prédiction de l'objet, ou encore une sauvegarde de l'état courant. L'ordonnanceur peut aussi informer les objets actifs de l'évolution de la simulation, en leur demandant par exemple de revenir à une date passée (d'effectuer un *rollback*).
Au niveau supérieur qui est global, les ordonnanceurs des clusters s'échangent des informations conformément au protocole de synchronisation.

Les principaux avantages de cette approche sont les suivants :

- les objets actifs ne gèrent pas la synchronisation et sont donc plus faciles à coder. Les mécanismes liés aux informations à fournir à l'ordonnanceur pourront, dans de nombreux cas, être réalisés comme une spécialisation des classes séquentielles ;
- les objets actifs sont relativement indépendants du mode de synchronisation. En effet, les protocoles nécessitant, de la part des objets, les mêmes informations, peuvent être interchangés ;
- la gestion de la synchronisation à un niveau supérieur aux objets actifs devrait permettre certaines optimisations (par exemple une réduction du nombre de messages nuls, une simplification de la topologie, etc.). De plus, une nouvelle synchronisation, ou l'optimisation d'une synchronisation existante, se fera en développant un nouvel ordonnanceur ; il ne sera pas nécessaire de développer toute une nouvelle hiérarchie de classes de modélisation.

L'inconvénient majeur de cette approche vient du fait que l'ensemble des objets d'un cluster partagent le même mode de synchronisation, ce qui restreint les optimisations envisageables.

Gestion de la synchronisation au niveau des objets actifs La mise en œuvre de la synchronisation au niveau des objets actifs (cf. Fig. 7.3.b) consiste à coder un comportement spécifique intégrant la gestion du protocole sélectionné. Un objet actif se comporte alors comme un *processus logique* ce qui permet d'utiliser directement les algorithmes existant dans la littérature. Mais le comportement est éloigné du comportement séquentiel et devra être adapté à chaque variation du protocole (par exemple lors de l'utilisation des messages nuls automatiques ou à la demande). De plus, chaque objet actif devra posséder toutes les structures et données liées à la gestion de la synchronisation.

Approche retenue pour Prosit

Dans cette première version du framework, nous avons retenu au niveau du cluster l'ordonnancement temporel. Cette approche nous a paru la plus intéressante dans la mesure où

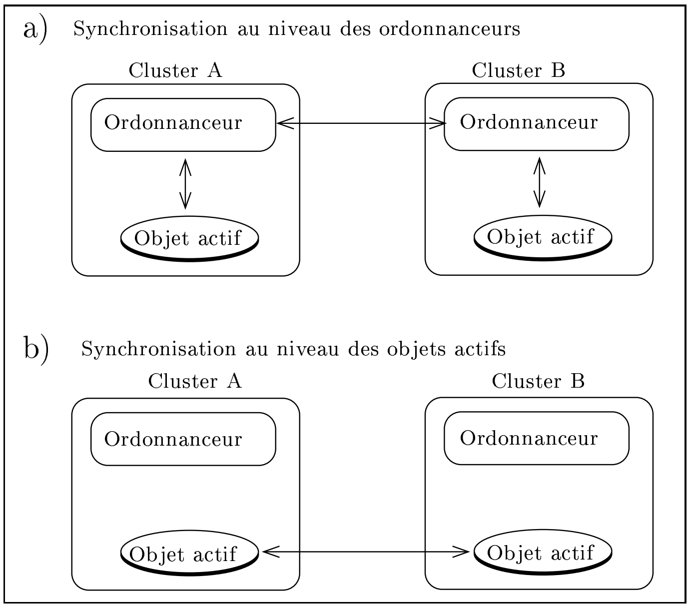


FIG. 7.3 – *Flux d'informations assurant la synchronisation inter-clusters*

elle permet d'obtenir le comportement d'une exécution séquentielle quand l'ensemble des objets actifs sont placés sur un même cluster. De plus, on réalise ainsi très simplement certains mécanismes de contrôle (contrôle de la progression des objets *rapides* dans la synchronisation optimiste, activation en priorité des objets actifs non bloqués dans la synchronisation conservatrice, etc.).

Au niveau inter-clusters, nous avons retenu comme premier domaine d'investigation, la gestion de la synchronisation au niveau de l'ordonnanceur. Nous avons fait ce choix à la fois pour des raisons pratiques (meilleure réutilisabilité du code séquentiel, compatibilité avec le modèle séquentiel, etc.) et pour des raisons de recherche. En effet, cette approche est à notre connaissance très peu étudiée (cf. l'étude des environnements existants présentés au chapitre 10). Nous avons vu précédemment que cette architecture nécessitait un protocole d'interaction entre l'ordonnanceur et les objets actifs. Pour cela, nous avons introduit une hiérarchie de classes définissant l'interface nécessaire. La figure 7.4 présente cette hiérarchie ainsi que les primitives définies.

7.5 Résumé du modèle distribué

Cette dernière section résume les grandes orientations adoptées dans ce chapitre. Au niveau de l'architecture :

- Le support d'exécution est organisé en un ensemble de clusters situés sur des computers éventuellement hétérogènes.
- A la création d'un objet actif, le programmeur spécifie le cluster cible.

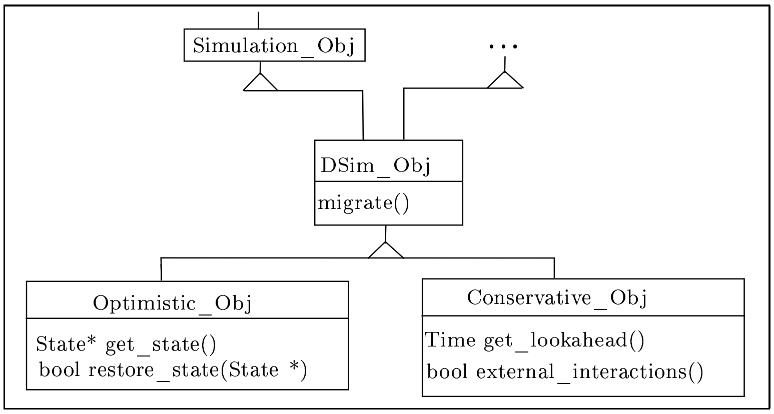


FIG. 7.4 – *Hiérarchie des classes distribuées*

Au niveau des nouvelles fonctionnalités des objets actifs :

- Les objets actifs sont manipulés de manière uniforme quelle que soit leur localisation.
- Le programmeur peut récupérer la localisation d'un objet actif. Il peut aussi obtenir une référence sur un objet actif à partir de sa classe et de son nom ou de son identifiant.
- Les objets actifs peuvent migrer entre clusters. La migration s'effectue dans le temps de simulation et a une durée nulle.

Au niveau de la synchronisation :

- La synchronisation est réalisée par les ordonnanceurs présents sur les clusters et comprend deux niveaux :
  1. au niveau du cluster, les objets actifs sont ordonnancés en fonction de leur date,
  2. entre clusters, les ordonnanceurs interagissent afin d'assurer la synchronisation globale.
- Les objets actifs participent à la synchronisation en fournissant des informations à l'ordonnanceur (valeur de prédiction, copie de l'état courant, etc.). Un objet actif doit aussi pouvoir réagir aux instructions de l'ordonnanceur (ex: effectuer un rollback). Ce protocole d'interactions est réalisé par la mise en œuvre d'une hiérarchie de classes.

8. MISE EN ŒUVRE D'UNE SIMULATION

Ce chapitre poursuit la description du modèle de programmation Prosit. Nous abordons ici les éléments propres à la réalisation d'une simulation et nous proposons quelques aides à la construction de simulateurs. Nous montrerons d'abord comment modéliser un système en termes d'objets actifs, puis comment coder le modèle obtenu. Pour la partie codage, nous détaillerons le codage des composantes puis la construction du modèle, et nous donnerons ensuite quelques indications sur les techniques de mise au point des simulations. Dans la dernière section, nous montrerons comment passer d'une simulation séquentielle à une simulation distribuée.

8.1 Modélisation par objets actifs

La réalisation de simulations est une tâche complexe, notamment pour des débutants, qui requiert la mise en place d'une méthodologie de développement. Une approche structurée et systématique est d'autant plus importante que l'utilisation d'un langage de haut niveau rend plus difficile la vérification et la validation de la simulation. L'objectif de cette section n'est pas de proposer une nouvelle méthodologie couvrant tout le cycle de vie d'une simulation, mais plutot de traiter du problème de la modélisation et de la conception dans le cadre de la programmation en Prosit.

La phase de modélisation comprend l'étude, l'analyse du système et la réalisation du modèle. La phase de conception consiste à passer du modèle à la spécification du programme de simulation. Ces deux étapes sont en théorie dissociables, mais une adéquation de la modélisation au processus de programmation Prosit rendra la conception et le codage plus naturels et donc plus efficaces en terme de temps, de qualité du code et de vitesse d'exécution. L'analyse du système physique doit donner lieu à une décomposition de ce dernier en composantes. Le nombre et la complexité des composantes dépendent de la dimension du système, de sa complexité intrinsèque et du niveau de détail désiré. A chaque composante sera associé un objet dans la simulation. La première tâche du programmeur consiste à déterminer pour chaque objet sa nature active ou passive. Ce choix est particulièrement important car il définit à la fois l'architecture logicielle et la dynamique de la simulation. Il n'existe pas de critères universels permettant de guider le programmeur et l'expérience joue ici un rôle très important. Il est aussi fréquent qu'une architecture possède un dual tout aussi valide. Nous illustrons cette dualité à la table 8.1 à partir d'un exemple de simulation de trafic routier. Nous pouvons néanmoins proposer les éléments d'aides suivants :

- une composante effectuant de manière autonome des actions, sera représentée par un objet actif ;

– une composante considérée comme une simple structure de données, manipulée par d'autres composantes pourra être représentée par un objet passif ;

– d'autres critères rentrent aussi en considération : un objet actif consomme plus de ressources (processeur, mémoire) qu'un objet passif, il est généralement plus facile de coder une simulation en terme d'objets actifs et les classes ont dans l'ensemble un plus grand potentiel de réutilisabilité.

Entités actives	Entités passives	Remarques
Infrastructures	Véhicules	Les segments de route interagissent par le biais de demandes de service ayant comme paramètre un véhicule
Véhicules	Infrastructures	Les véhicules sont actifs et se synchronisent à l'aide de structures de données passives représentant les segments de route
Véhicules Infrastructures		L'ensemble des entités sont actives. La traversée d'un segment par un véhicule correspond à une demande de service au segment

TAB. 8.1 – *Dualité actif/passif dans le cadre d'une simulation de trafic routier*

Une fois les composantes actives recensées, trois aspects doivent ensuite être pris en compte :

1. *Aspect statique et descriptif.* Le modélisateur doit décrire la structure statique des objets actifs : leur identité, leurs attributs, etc.
2. *Aspect dynamique.* L'aspect dynamique décrit le système physique en relation avec le temps, et le séquencement des opérations. L'analyse dynamique consiste à :
   (a) spécifier les interactions entre objets actifs. Pour chaque objet actif, il faut recenser les objets avec lesquels il interagit et détailler l'interaction : nom du service, mode bloquant ou non, paramètres (niveau modèle et contrôle), types et modes de retour pour les demandes bloquantes (valeur de retour, cas de rejet, etc.) ;
   (b) spécifier les propriétés dynamiques de chaque objet actif (mode de gestion des requêtes et des activités services) ;
   (c) préciser le comportement dynamique des objets. Cette étape permet de préciser comment, dans le temps de simulation, évolue et se comporte l'objet. C'est essentiellement une mise en œuvre algorithmique des propriétés recensées aux points (a) et (b). Cette étape consiste à spécifier la fonction `behave()` de chaque classe active.
3. *Aspect fonctionnel.* On précise le code des services réalisés par les objets par l'intermédiaire des méthodes.

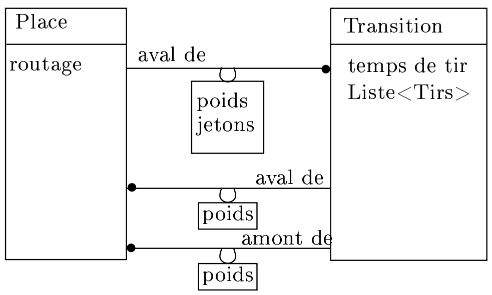
Nous pouvons remarquer que la méthodologie proposée est proche de certaines techniques d'analyse objet existantes. Seule la composante progression du temps de simulation est spécifique à notre domaine. Ainsi, la méthode OMT [Rumbauch, 1996] qui distingue lors d'une analyse : le modèle objet, le modèle dynamique et le modèle fonctionnel peut aussi être utilisée. La notation graphique d'OMT est aussi une forme de représentation très adaptée.
Nous illustrons, à la page suivante, les deux premiers aspects de la méthodologie dans le cadre de la simulation de réseaux de Petri.

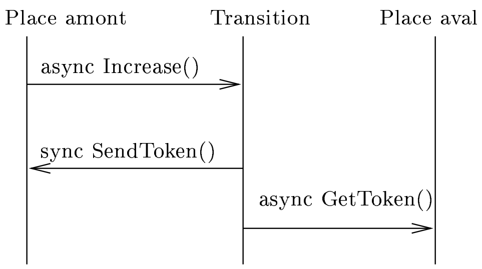
Modélisation par objets actifs d'un réseau de Petri

Les réseaux de Petri ont été introduits par C.A. Petri, dans les années 60-62, afin de modéliser les concepts d'actions asynchrones et concurrentes. Ils constituent un modèle graphique permettant de représenter les systèmes à événements discrets à l'aide de composantes élémentaires : les places, les transitions et les jetons. Le lecteur intéressé par plus d'informations sur la sémantique peut consulter [Murata, 1989].
Les objets actifs sont les places et les transitions. Les jetons sont passifs. Nous nous intéressons ici à la politique de routage cyclique et probabiliste.

1) *Aspect statique et descriptif*
- L'objet *place* a comme attribut : un identifiant alphanumérique, la liste des transitions en aval et pour chacune d'elles la liste des jetons qui lui sont réservés et le poids de l'arc, les données nécessaires au routage (liste de transitions si le routage est cyclique, la probabilité associée à chaque transition si le routage est probabiliste).
- L'objet *transition* a comme attribut : un identifiant alphanumérique, son temps de tir, la liste des places en aval et pour chacune d'elles le poids de l'arc, la liste de places en amont et le poids des arcs, la liste des tirs en cours.

2) *Aspect dynamique*
- L'objet *place* reçoit des transitions en amont les nouveaux jetons (service non bloquant `GetToken()`). Elle affecte a chaque jeton une transition, puis informe les transitions (service non bloquant `Increase()`) du nombre de jetons qui leurs sont réservés. La place sert aussi les demandes de jetons venant des transitions en aval (service bloquant `SendToken()`). Les requêtes sont servies immédiatement sans consommation de temps et directement par le comportement.
- L'objet *transition* reçoit des places en amont le nombre de jetons disponibles (service non bloquant `Increase()`). Si la transition peut tirer, elle demande de manière bloquante (service bloquant `SendToken()`) à chacune des places en amont les jetons, puis initialise une structure de tir. A la fin d'un tir, elle génère les jetons en sortie et les passe aux places en aval (service non bloquant `GetToken()`). Les requêtes sont servies immédiatement sans consommation de temps et directement par le comportement.





Il est important de considérer la mise en œuvre d'une simulation comme un processus itératif. La première étape consiste à modéliser une version simplifiée du système physique, ce modèle est ensuite complété progressivement afin d'augmenter la fidélité de la simulation. Cette construction incrémentale des classes actives peut intervenir à plusieurs niveaux. Le programmeur peut compléter le comportement des objets, il peut aussi compléter les services ou rajouter de nouveaux services, il peut enfin introduire de nouvelles classes actives. De cette construction incrémentale, il est parfois intéressant de conserver différentes versions des classes actives. Ultérieurement, en fonction de la fidélité, des détails et des performances désirés, le modélisateur pourra sélectionner la classe appropriée.

Une fois achevée la conception des classes actives, il reste à coder ces classes puis à construire la simulation.

8.2 Codage des composantes

Lors du codage d'une classe active, trois situations peuvent se présenter :

1. la classe active est déjà disponible dans une librairie et peut donc être directement utilisée. C'est le cas le plus favorable et l'approche bibliothèque de Prosit favorise et généralise cette approche ;
2. la classe n'est pas disponible, mais une classe ayant le même comportement existe dans une librairie. Le modélisateur doit donc coder une nouvelle classe, qui hérite du comportement, et qui spécifie les nouveaux membres (attributs et méthodes) implémentant les services. La construction des composantes d'un modèle par héritage du comportement et spécification des services est à la base de la librairie file d'attente présentée à la section 13.1.1 ;
3. le comportement de la classe est spécifique, le modélisateur doit donc la coder entièrement en faisant appel aux classes de base du framework. C'est le cas le plus défavorable, mais l'architecture adoptée et les mécanismes objets (principalement la surcharge) facilitent grandement cette tâche. Le programmeur doit essentiellement spécifier le mode de gestion des requêtes et des activités annexes (cf. §6.3.3 et §6.4.2) ainsi que le code des services et les interactions avec les autres objets actifs.

Ces trois situations sont représentées à la figure 8.1.

8.3 Construction de la simulation

8.3.1 Le noyau de simulation

Avant d'aborder le processus de codage du modèle, il est important d'analyser de plus près le déroulement d'une exécution. Nous pouvons y recenser divers traitements qui ne font pas partie du comportement des entités modélisées, mais concernent la gestion de la simulation. Nous avons par exemple : la gestion du temps et de la synchronisation, les services collectifs (génération d'identifiants, informations systèmes, génération de nombres aléatoires, etc.), l'aide

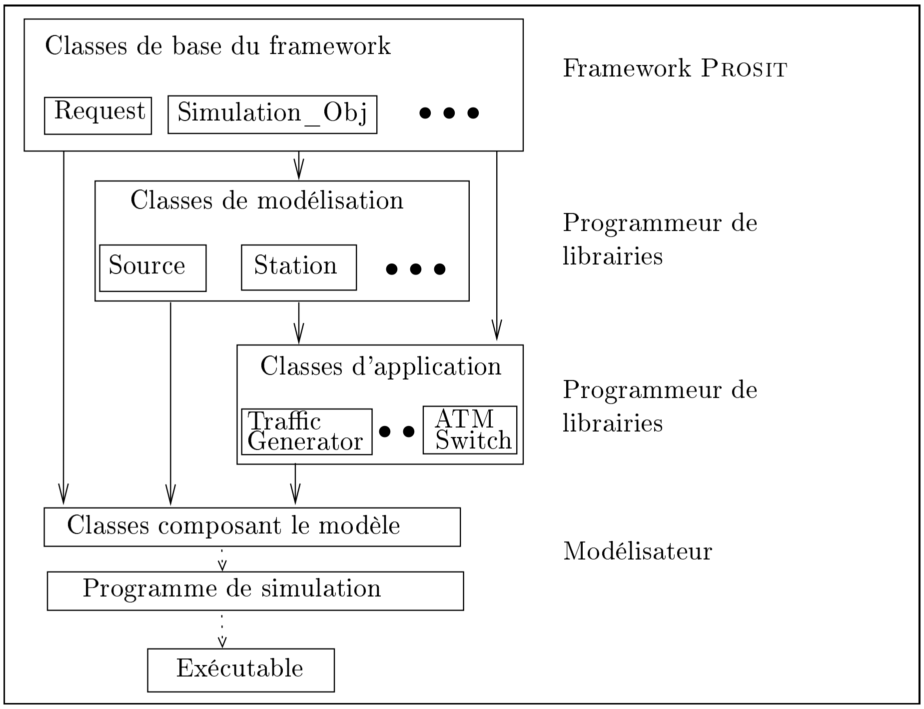


Fig. 8.1 – *Mise en œuvre d'une simulation*

à la mise au point et à l'optimisation (traces, exécution pas à pas, vérification de propriétés et d'invariants, etc.), etc. Ces actions ne peuvent être à la charge des objets actifs composant le modèle et devraient être menées par un objet actif spécifique. Pour cela, nous introduisons l'objet *noyau de simulation* qui est un objet actif particulier dont le comportement consiste à gérer et à contrôler la simulation. Paramétrer la simulation (date maximale de simulation, génération de traces, etc.) revient à paramétrer l'objet noyau. Les diverses informations sur le déroulement de la simulation (nombre d'entités actives, date de fin de simulation, etc.) devront aussi être demandées à l'objet noyau. La sélection du mode de synchronisation se fera aussi en fonction de la classe du noyau.

Nous avons indiqué précédemment que la synchronisation séquentielle et distribuée était assurée par un ordonnanceur. La classe noyau est l'abstraction du framework représentant cet ordonnanceur. Nous présenterons plus en détail le noyau dans la partie dédiée à l'implémentation.

8.3.2 Codage du modèle

La spécification d'une simulation peut être divisée en trois phases : la construction et l'initialisation du modèle, l'exécution de la simulation et enfin les traitements post-mortem. A chacune des ces phases correspondent des actions spécifiques. Il est donc important de les distinguer à la fois conceptuellement, au niveau de la structure du programme et au niveau de son déroulement. Nous détaillons dans la suite chacune de ces phases et nous donnons ensuite un exemple complet.

Construction et initialisation du modèle La construction d'une simulation consiste à créer et à initialiser tout d'abord le noyau de simulation, puis l'ensemble des objets actifs composant le modèle étudié[1]. Ces objets actifs sont instances soit de classes provenant de bibliothèques, soit de classes codées par l'utilisateur. La construction d'une simulation peut donc être vue comme un assemblage d'instances, ce qui correspond au schéma classique de la programmation objet. Il n'est pas nécessaire de créer l'ensemble des objets actifs au début de la simulation. De manière dynamique durant l'exécution, il sera possible de créer ou de détruire des objets actifs.

Dans le cadre distribué, préalablement à la création du noyau et des objets actifs, le programmeur doit créer la machine virtuelle et les clusters.

Lancement et exécution de la simulation Une fois le modèle en place, il faut être en mesure de lancer l'exécution de la simulation. Pour cela, l'objet noyau dispose d'une primitive `run()`.

Traitements post-mortem Les traitements post-mortem, typiquement calcul et affichage de statistiques, sont effectués à la fin de la simulation. Le code associé doit être placé après l'appel de la primitive `run()` sur l'objet noyau.

Un exemple de spécification : simulation d'un super-marché

Nous donnons ici un exemple de spécification de simulation d'un super-marché. Le modèle comprend cinq composantes (porte d'entrée, rayons, caisses, caisse centrale et porte de sortie) et nous créons aussi un objet source qui générera les clients.

8.4 Mise au point

Une fois le système modélisé et le programme de simulation écrit, le modélisateur ne peut exploiter les résultats des exécutions sans avoir préalablement validé le modèle et vérifié le programme (cf. Fig 8.2).

La *validation* consiste à s'assurer que le modèle conceptuel est une bonne représentation de la réalité étudiée. La validation du modèle est indépendante de notre approche et nous renvoyons le lecteur à [Leroudier, 1980] Chapitre 6 ou à [Sargent, 1984].

La *vérification* consiste à s'assurer que le programme de simulation est fonctionnellement correct et que la simulation fait bien ce qui était prévu (*i.e.* qu'il y a une bonne implémentation du modèle conceptuel). L'utilisation d'un langage objet non dédié à la programmation de simulations, permet de faire appel aux techniques classiques de génie logiciel (voir par exemple [Gaudel *et al.*, 1996]) et à celles propres aux langages objets (voir par exemple [Communication of the ACM, 1994]). Le relatif cloisonnement des communautés fait que ces techniques sont rarement utilisées dans le domaine de la simulation. Whitner et Balci présentent dans

1. Il est important que le premier objet actif créé soit le noyau, car ce dernier peut alors être consulté lors de la construction des autres objets actifs.

Exemple de code 8.1 Un exemple de spécification : simulation d'un super-marché

```
...
Kernel *noyau = new Kernel; // creation du noyau
void main() {
  // construction et initialisation du modele
  Office *superviser_office = new Office(...);
  Door *in = new Door(...);
  Counter *counter = new Counter(...);
  Cashier *cashier = new Cashier(...);
  Door *out = new Door(...);
  Main_Source *source = new Main_Source(...);

  // parametrage et lancement de la simulation
  noyau->set_max_simulation_time(5000);
  noyau->run();

  // traitement post-mortem
  noyau->report_stat();
}
```

[Whitner et Balci, 1989] différentes techniques de génie logiciel pouvant être utilisées lors de la vérification d'une simulation. Parmi ces techniques certaines peuvent être utilisées dans le cadre d'une simulation Prosit :

- *L'analyse informelle* est essentiellement une approche intuitive qui comprend l'inspection de code, l'audit, les révisions externes, etc.
- *L'analyse statique* est uniquement basée sur l'étude du code source et peut être réalisée par des outils développés en interne.
- *L'analyse dynamique* étudie le comportement du modèle durant son exécution. Les critères d'analyse sont collectés au cours de cette exécution. Nous trouvons dans cette catégorie les traces d'exécution, le profiling, le monitoring, les tests de non-régression, etc.
- *L'analyse de contraintes* permet de vérifier la conformité du comportement dynamique du modèle avec les hypothèses faites lors de la conception et du codage. Elle est essentiellement basée sur l'utilisation d'assertions, d'invariants et sur l'analyse du système aux limites.

Différentes caractéristiques du modèle de programmation Prosit peuvent être utilisées pour la vérification de simulations. Tout d'abord, l'approche framework permet de spécialiser certaines composantes et d'intégrer de nouveaux mécanismes. La spécialisation d'une composante peut par exemple consister à développer un noyau de simulation chargé de vérifier des propriétés dynamiques (vitesse de progression du temps des objets actifs, ordre d'activation, etc.). L'analyse de traces et de résultats statistiques est aussi très utile, nous présenterons au chapitre 13 l'intégration de ces mécanismes à l'environnement. Ensuite, la réification per-

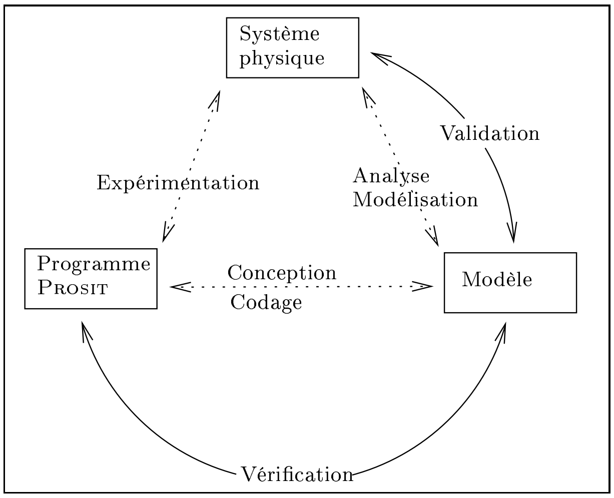


Fig. 8.2 – *Processus simplifié de validation et de vérification d'une simulation*

met une forme d'introspection[2] des objets particulièrement utile, permettant par exemple de vérifier des invariants et des assertions.

8.5 Distribution de la simulation

La distribution d'une simulation peut être motivée par deux facteurs : le manque de mémoire ou un temps d'exécution trop important. Si pour le premier point la distribution permet en général d'atteindre les objectifs (*i.e.* la simulation de modèles plus grands), ce n'est pas toujours le cas pour le second. En effet, la répartition des objets actifs et le maintien de la causalité entrainent des coûts important[3] qui ne se justifient que si le système possède un important degré de parallélisme intrinsèque. L'évaluation du parallélisme d'un modèle et la sélection du "bon" protocole de synchronisation (*i.e.* celui qui donnera la meilleure accélération) sont donc des tâches importantes, mais particulièrement difficiles. L'architecture de Prosit et les mécanismes disponibles (traces, statistiques, etc.) permettront de mettre en place des outils d'évaluation du parallélisme et d'aide à la sélection des protocoles[4]. Nous considérons dans la suite que le modèle possède du parallélisme et que le mode de synchronisation a été sélectionné.

Une simulation est constituée du programme principal et des librairies utilisées, chacune de ces composantes doit être adaptée lors de la distribution.

2. Par introspection, nous entendons une observation méthodique, par l'objet, de ses structures internes.
3. Ces coûts supplémentaires sont évalués au chapitre 14.
4. L'évaluation du parallélisme d'un modèle consiste généralement à analyser un fichier de traces, issu d'une simulation séquentielle, afin de déterminer le chemin critique (voir par exemple [Loucks et Nandy, 1992], [Berry et Jefferson, 1985]).

8.5.1 Génération des classes distribuées

L'existence de classes spécifiques à la simulation distribuée est due au protocole d'interactions entre l'ordonnanceur et les objets actifs (cf. §7.4.2), et aux nouvelles fonctionalités que sont la migration et le binding. La quantité de travail nécessaire à la génération de ces classes est variable et dépend à la fois, de la disponibilité des librairies distribuées, du mode de codage des classes séquentielles utilisées et des optimisations à implémenter. Nous pouvons distinguer trois situations possibles :

1. La simulation séquentielle utilise des classes de modélisation d'une librairie, et une version distribuée de cette librairie est disponible. Le programmeur peut donc directement les utiliser, il ne lui reste plus qu'a coder le programme principal (voir la section suivante).
2. Le programmeur dispose des sources des classes séquentielles. Il doit alors modifier la relation de descendance, en héritant de la classe distribuée correspondante au protocole de synchronisation. Il doit ensuite spécialiser les primitives du protocole d'interaction avec l'ordonnanceur.
3. Le programmeur ne dispose pas des sources des classes séquentielles. Il doit alors dériver une nouvelle classe qui hérite à la fois de la classe séquentielle et de la classe distribuée correspondante au protocole de synchronisation. Il doit ensuite spécialiser les primitives du protocole d'interaction avec l'ordonnanceur. Nous pouvons voir que ce dernier cas introduit un héritage multiple de certaines classes de base du framework. Les ambiguités introduites devront être levées par les mécanismes propres au langage de programmation utilisé (voir par exemple [Stroustrup, 1993] §6.5).

8.5.2 Codage du programme principal

Comme dans le cadre séquentiel, le programme principal comporte la création et l'initialisation des composantes du modèle, le lancement de la simulation et les traitements post-mortem. Le programmeur peut donc réutiliser le programme séquentiel et y rajouter la création de la machine virtuelle et des clusters, ainsi que la création et d'initialisation du noyau correspondant au mode de synchronisation retenu. Il doit enfin modifier les classes des objets actifs crées et spécifier leur placement

Un exemple de spécification d'une simulation distribuée

Nous reprenons ici l'exemple de la simulation du super-marché (exemple de code 8.1 p 68) en distribuant le modèle sur deux clusters (le cluster courant et le cluster `node`). La synchronisation optimiste Time Warp est utilisée.

Exemple de code 8.2 Un exemple de spécification distribuée : simulation d'un super-marché

```
Distributed_Kernel *noyau = new TW_Kernel; // creation du noyau
void main() {
  // construction et initialisation de la machine virtuelle
  Computer *comp = new Computer("m1","uranus.inria.fr");
  Cluster *node = new(comp) Cluster;
  Distributed_Kernel::check_virtual_machine();
  // construction et initialisation du modele
  TW_Main_Source *source = new TW_Main_Source(...); // cluster courant
  TW_Office *superviser_office = new TW_Office(...);
  TW_Door *in = new TW_Door(...);
  TW_Counter *counter = new TW_Counter(...);
  TW_Cashier *cashier = new(node) TW_Cashier(...); // cluster node
  TW_Door *out = new(node) TW_Door(...);
  // parametrage et lancement de la simulation
  noyau->set_max_simulation_time(5000);
  noyau->run();
  // traitement post-mortem
  ...
  noyau->halt(); // arret de la machine virtuelle
}
```

9. FONCTIONNALITÉS ÉTENDUES

Dans ce chapitre, nous montrons en quoi l'architecture retenue favorise le développement de simulations modulaires et réutilisables. Nous montrons ensuite comment PROSIT supporte naturellement la simulation mixte et la simulation hiérarchique. Nous illustrons chacun de ces points par un exemple.

9.1 Simulation modulaire et réutilisable

De nombreuses caractéristiques de PROSIT permettent la construction modulaire de simulations et la réutilisabilité des composants. Ces caractéristiques résultent à la fois des choix d'implémentation et du modèle de programmation retenu. Parmi les choix de mise en œuvre, l'utilisation du paradigme objet tient une place prépondérante, vient ensuite l'architecture framework qui généralise ces deux propriétés à tous les niveaux (programme de simulation, classes de modélisation, structures internes, etc.). Parmi les caractéristiques du modèle de programmation contribuant à la modularité, nous avons l'organisation d'une simulation en un ensemble d'objets actifs interagissant par le biais d'une interface bien définie. Ensuite, le découplage entre le comportement des objets (spécifié dans la méthode `behave()`) et les services offerts (spécifiés dans le code des méthodes de service) permet de développer des librairies de classes de comportement. Enfin, la réutilisabilité des composants, peut aussi être accrue par un choix judicieux des entités qui seront représentées par des objets actifs. Nous illustrons cette dernière possibilité à la section suivante.

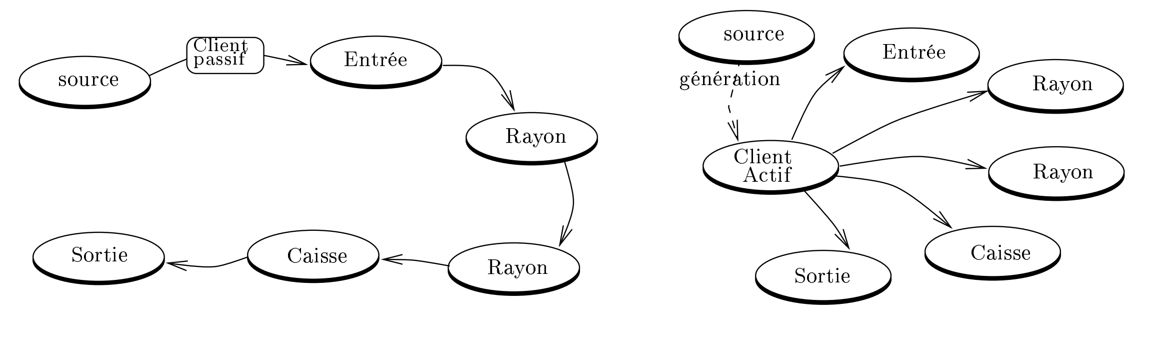
9.1.1 Un exemple de réutilisabilité structurelle : la simulation de files d'attente

Le formalisme des réseaux de files d'attente [Kleinrock, 1975] a été depuis longtemps utilisé pour la modélisation des systèmes à événements discrets. Le système y est décrit comme un ensemble de *stations* de service s'échangeant des *clients*. Une station, qui est une file d'attente, représente une ressource physique ou logique pouvant fournir des services, tandis qu'un client représente un objet circulant ou une requête de service aux stations. Le même modèle peut contenir différentes classes de clients et chaque station est paramétrée selon la partie du système réel qu'elle représente. Outre les stations qui sont les composantes les plus importantes, d'autres nœuds peuvent être inclus dans le réseau notamment : les sources, les puits et les nœuds de synchronisation.

L'analyse du formalisme permet de distinguer deux familles de composantes : les stations et les clients. S'il semble naturel de représenter chaque station par un objet actif, le choix est plus complexe pour ce qui est des clients qui peuvent être vus comme des entités actives ou passives. Discutons des caractéristiques de chacune de ces approches et montrons en quoi elles influencent la réutilisabilité.

Clients actifs Dans cette approche chaque client est représenté par un objet actif. Le comportement de ce dernier consiste à émettre des requêtes (généralement bloquantes) aux objets stations situés sur son parcours. Nous avons appelé cette architecture, l'architecture *client actif* (cf. Fig. 9.1.b). Le principal avantage de cette modélisation réside dans le fait que les stations ne gèrent plus le routage, spécifique à un système, et peuvent donc être réutilisées. Le cheminement d'un client est clairement isolé dans sa classe et non pas répartie entre les serveurs. Un autre avantage provient de la facilité avec laquelle on peut obtenir des statistiques par classes et des statistiques de *bout-en-bout*. Il suffit en effet d'inclure la prise des mesures dans les classes clients. L'inconvénient majeur de cette approche vient du fait qu'un objet actif consomme plus de ressources (mémoire, CPU) qu'un objet passif. Le nombre de clients simultanément présents sera alors plus limité et les simulations auront, généralement, des temps d'exécution plus importants.

Clients passifs La simulation est structurée en un ensemble d'objets stations qui reçoivent des clients, effectuent divers services et routent les clients en sortie. Le client est une simple structure de données, une entité passive. Nous avons appelé cette architecture, l'architecture *client passif* (cf. Fig. 9.1.a). Cette approche suit le schéma classique des implémentations existantes (voir par exemple [Simulog, 1995]), par contre il n'y a pas ici émission explicite de l'objet client, mais demande non bloquante d'un service dont l'objet client est le paramètre. Cette architecture ne facilite pas la réutilisation des classes serveurs, dans la mesure où elles gèrent le routage spécifique à un modèle.



a) Architecture client passif b) Architecture client actif

Fig. 9.1 – *Architectures pour les modèles files d'attente*

9.2 Simulation mixte

La réalisation de simulations hétérogènes répond à un double objectif : faciliter le codage en utilisant le paradigme le mieux adapté à une composante, et réduire les temps d'exécution. Dans Prosit, deux techniques permettent de réaliser des simulations hétérogènes : l'intégration de nouveaux moteurs de simulation et le couplage de classes issues de librairies de modélisation différentes. Nous ne nous intéressons pas ici à l'interconnexion de simulateurs hétérogènes, comme c'est par exemple le cas dans le protocole DIS (cf. §3.1), mais à l'intégration de simulateurs hétérogènes.

9.2.1 Intégration de nouveaux moteurs de simulation dans Prosit

L'intégration d'un nouveau moteur de simulation au sein de Prosit peut intervenir à deux niveaux. Le premier consiste à modifier le noyau de simulation et l'ordonnanceur afin de mettre en place un ordonnancement spécifique (par exemple un ordonnancement cyclique pour la simulation d'automates). La seconde technique, qui sera certainement la plus courante, consiste à intégrer le moteur de simulation au sein d'un objet actif. L'intégration est vue comme un comportement spécifique et est donc une opération transparente pour l'ordonnanceur. Ce nouveau comportement doit néanmoins respecter les contraintes de base des objets actifs (progression croissante du temps de simulation, protocole d'interaction, etc.). Nous illustrons cette seconde technique dans la section suivante.

Un exemple de simulation mixte : intégration d'un moteur équationnel

Ce travail est issu d'une collaboration avec A. Jean-Marie et B. Gaujal [Gaujal *et al.*, 1997a].

Objectifs Nous avons introduit à la section 2.2.3 la simulation équationnelle et indiqué qu'elle apportait un gain très important en vitesse d'exécution. Cependant, cette technique est très spécifique et ne peut être utilisée que pour des classes particulières de systèmes discrets. Il est donc généralement difficile de représenter le système complet par cette approche. D'où le besoin d'utiliser une approche mixte.

Présentation de la technique équationnelle La simulation équationnelle d'un système consiste à utiliser des *équations récursives d'évolution* afin de calculer les états successifs du système. Le principe général est le suivant. Considérons que le n^{ieme} état du système est décrit par un vecteur $X(n)$, et qu'il existe une fonction f tel que :

$$X(n+1) \;=\; f(X(n)) \;. \tag{9.1}$$

L'algorithme de simulation consiste alors à générer la séquence des $X(1), X(2), \ldots, X(N)$, à partir desquels des estimateurs statistiques peuvent être calculés. Cette technique s'applique aux systèmes pouvant être mis sous la forme de l'équation (9.1), c'est le cas des graphes d'événements [Baccelli *et al.*, 1992], et des réseaux à choix libres [Baccelli *et al.*, 1996]. Nous nous intéressons ici aux réseaux de Petri temporisés : la transition i a un temps de tir de τ_i unités et les jetons ont un temps de séjours minimum de σ_j dans la place j. Toutes ces valeurs sont constantes et multiples d'une valeur δ considérée comme l'unité dans la suite. Quand une place possède plusieurs transitions de sorties, un choix existe qui peut être résolu par une politique de routage cyclique, de routage aléatoire, ou encore par un mécanisme de priorité entre les transitions en conflit ; Les jetons sont alors alloués à la transition ayant la plus grande priorité. Pour plus d'informations, nous renvoyons le lecteur à la lecture de [Baccelli *et al.*, 1992, Baccelli *et al.*, 1996].

Les variables d'états utilisées par l'algorithme sont les $X_i(n)$, qui indiquent le nombre de tirs commencés par la transition i à la date n. Ces valeurs peuvent être calculées par la relation suivante :

$$X_i(n) \;=\; \min_{p \in {}^\bullet i} H_{pi}\left(\sum_{j \in {}^\bullet p} X_j(n - \sigma_p - \tau_j) + M_p\right) \;. \tag{9.2}$$

M_p est le marquage initial de la place p, ${}^\bullet i$ est l'ensemble des places précédent la transition i, ${}^\bullet p$ est l'ensemble des transitions précédent les éléments de ${}^\bullet i$, enfin $H_{pi}(m)$ est la *fonction de routage* qui indique combien, parmi les m premiers jetons entrés dans la place p, ont été routés vers la transition i. Nous donnons à l'exemple de code 9.1 l'algorithme de la simulation équationnelle. Cette algorithme est du type *progression à pas fixes* (cf. §2.2.2).

Exemple de code 9.1 Boucle principale de simulation équationnelle d'un réseau de Petri

```
while (/*poursuivre la simulation*/) {
  for (i transition) {
    // calculer le nombre de jetons arrivés dans les places d'entrées m_p
    // incrémenter la fonction de routage H_pi(m_p)
    // calculer le compteur de X_i(n) à l'aide de (9.2)
  }
  n++;
}
```

Une caractéristique de l'approche équationnelle, que nous n'avons pas encore exploitée, est son adaptation aux protocoles conservateurs et optimistes. En effet, un module équationnel possède une valeur calculable de prédiction et la notion d'état est aussi bien définie (c'est essentiellement la structure des $X_i(n)$et des $H_{pi}(m)$).

Intégration du moteur équationnel L'intégration d'un module équationnel à une classe d'objet actif Prosit (cf. Fig. 9.2) consiste à coder un comportement spécifique qui se charge de recevoir et de transmettre les demandes de service au module équationnel (réception de jetons), de faire progresser les compteurs et le temps du module équationnel et enfin de générer les demandes de service issues du module équationnel (émission de jetons). Nous donnons à l'exemple de code 9.2 le comportement d'un objet équationnel.

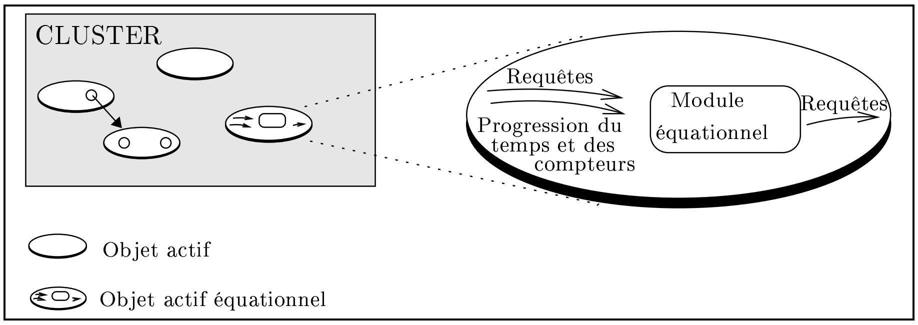


Fig. 9.2 – *Interactions entre le comportement et le module équationnel*

Exemple de code 9.2 Comportement d'un objet équationnel

```
void EqThing::behave()
  while (/*poursuivre la simulation*/) {
    // livraison des demandes de service au module équationnel
    // incrémentation des compteurs de tirs
    // génération des demandes de service
    // incrémentation du temps du module équationnel
    wait(δ);
  }
}
```

Nous présentons au chapitre 15, une utilisation de cette technique dans le cadre de la simulation de réseaux de commutateurs ATM.

9.2.2 Couplage de classes de simulation

Le couplage de classes de simulation consiste à utiliser et à faire interagir au sein d'un même programme des classes modélisant les entités selon des formalismes différents. Il est par exemple envisageable de réaliser une simulation contenant des composantes modélisées par des files d'attente et d'autres, par des réseaux de Petri. La difficulté consiste à faire coopérer les instances et à intégrer plusieurs schémas de contrôle et d'exécution (priorité, mode d'interaction, etc.). Une difficulté d'un autre ordre est celle de la sémantique de l'interaction. En effet, le couplage n'a de sens que si le système simulé est conforme au système physique. Nous considérons le problème de sémantique résolu, le couplage peut alors être réalisé via le codage de classes hybrides ou le codage de classes interfaces.

Le couplage par codage de classes hybrides consiste à développer des classes supportant les deux formalismes. Cette approche peut sembler séduisante, notamment parce qu'elle s'adapte bien au concept d'héritage multiple, mais est généralement difficile à mettre en œuvre, car elle nécessite le codage d'un comportement spécifique intégrant les deux comportements.

La seconde approche consiste à coder des classes interfaces[1], dont le rôle est de transformer la demande de service d'un paradigme en une demande de l'autre paradigme. Ces objets devant aussi se charger, si besoin, de la transformation des paramètres de la demande de service et de la réponse. Nous avons par exemple utilisé cette approche pour le couplage des librairies réseaux de Petri et réseaux de files d'attente. Pour cela, nous avons codé la classe `QNToPN` dont le service d'une requête ayant un client en paramètre consiste à créer un objet jeton puis à le transmettre, via le service approrié, à la place située en aval. Nous avons aussi codé la classe `PNToQN` effectuant les opérations inverses.

9.3 Simulation hiérarchique

L'approche hiérarchique a pour objectif de construire des simulations de plus haut niveau à partir de composants de base. Ces composants peuvent être élémentaires ou à leur tour

1. Par exemple en faisant appel au *pattern adapter* (voir [Gamma *et al.*, 1995] p. 139).

constitués d'autres composants. L'utilisation du paradigme objet favorise cette approche et nous proposons de réaliser la hiérarchie en adaptant le principe d'encapsulation au concept d'objet actif.

Pour cela, nous introduisons les objets *container*. Un *container* est un objet actif pouvant contenir d'autres objets actifs et se chargeant de les ordonnancer. Un objet *container* possède aussi des fonctions-services comprenant la séquence des demandes de service aux objets actifs internes (cf. Fig. 9.3). L'exécution d'un service par l'objet container, correspond à une suite de demandes de service aux objets actifs internes. L'utilisation d'une classe container est entièrement transparente à l'utilisateur et ses avantages sont doubles :

– Le programmeur peut directement ré-utiliser des briques de base, tout en masquant l'architecture et la dynamique du sous-modèle aux utilisateurs. Si ce mécanisme n'était pas disponible soit le programmeur aurait à coder une classe active simulant le sous-modèle, soit l'utilisateur aurait à *décomposer* sa demande de service en une suite de demandes aux composantes internes.

– Cette hiérarchisation permet d'introduire un ordonnanceur spécifiques à un sous-ensemble de composantes, ce qui introduit de nouvelles possibilités pour la simulation hétérogène. Notons que cette hiérarchisation est possible grâce au fait que le noyau de simulation est lui aussi un objet actif (cf. §8.3.1).

Mais la hiérarchisation introduit plusieurs niveaux d'ordonnancement et donc des surcoûts. Elle sera parfois mise à plat au moment de l'exécution.

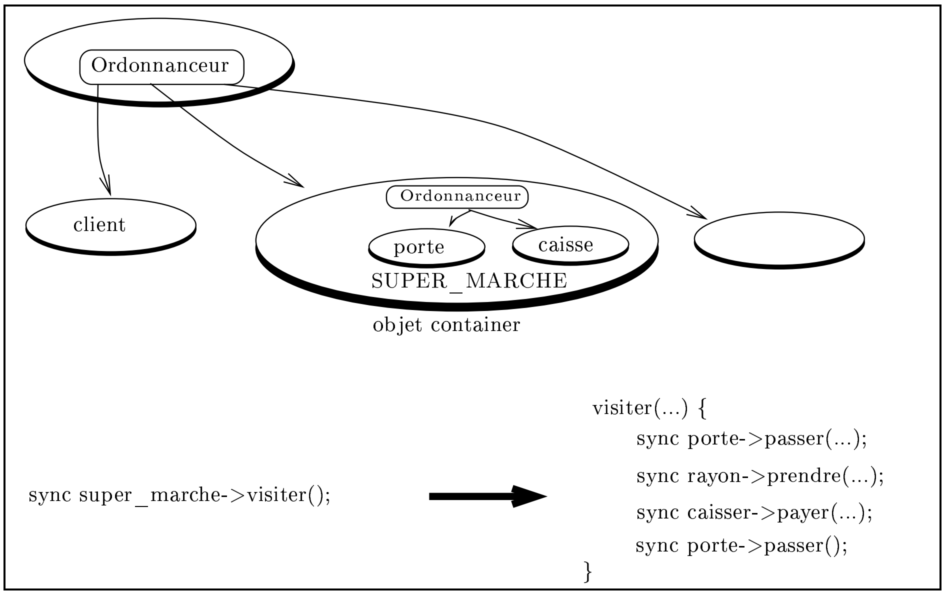


FIG. 9.3 – *Architecture d'une simulation hiérarchique et fonction-service d'une classe* container

9.4 La simulation répliquée

La réplication est une technique qui consiste à effectuer la simulation de plusieurs trajectoires et à analyser les résultats agrégés. Cette approche est utilisée lors de l'étude de systèmes possédant des phases de regénération (chaque phase peut être simulée par une exécution) ou encore pour l'étude des régimes transitoires, elle est aussi adaptée aux problèmes d'optimisation. Nous n'avons pas intégré au modèle de programmation de mécanismes spécifiques et nous montrons aux sections suivantes comment les mécanismes existants permettent de réaliser des réplications en environnement séquentiel et distribué.

9.4.1 Réplication séquentielle

La réplication séquentielle consiste à repéter les trois phases composant une simulation (cf. §8.3.2) jusqu'à ce que la condition d'arrêt des réplications soit vérifiée. Nous illustrons cette technique à l'exemple de code 9.3 qui reprend l'exemple de code 8.1 de la page 68.

Exemple de code 9.3 Réplication séquentielle de la simulation d'un super-marché

```
...
void main() {
  while (/* poursuivre replication*/) {
    // creation des objets actifs
    noyau = new Kernel;
    Office *superviser_office = new Office(...);
    Door *in = new Door(...);
    Counter *counter = new Counter(...);
    Cashier *cashier = new Cashier(...);
    Door *out = new Door(...);
    Main_Source *source = new Main_Source(...);
    // parametrage et lancement de la simulation
    noyau->set_max_simulation_time(5000);
    noyau->run();
    // agregation des statistiques de l'execution courante
    ...
    // destruction des objets actifs
    ...
  }
  // affichage des statistiques globales
  ...
}
```

Une application spécifique du framework pourra nécessiter la mise en place d'un *gestionnaire de réplication* chargé de la collecte et de l'analyse des statistiques. Le gestionnaire décidera aussi de l'arrêt ou de la poursuite des exécutions.

9.4.2 Réplication distribuée

La distribution des réplications est aussi une opération relativement aisée dans le cadre du modèle de programmation distribuée. Elle consiste à créer chaque objet actif composant le modèle, non pas sur un cluster mais sur chacun des clusters disponibles. Chaque cluster exécute alors la simulation complète du modèle. Le cluster principal, lui n'exécute pas de simulation, et se charge d'en détecter la fin sur les clusters esclaves afin de récuperer les statistiques et de relancer une exécution (en detruisant et en recréant les objets actifs comme dans le cadre séquentiel). Nous donnons à la figure 9.4 le diagramme d'état du cluster principal. La distribution des réplications repose donc sur un mode de calcul maître/esclaves. Cette technique nécessite néanmoins la spécialisation du noyau distribué qui fournit les primitives et fonctionnalités suivantes :

- aucune synchronisation entre clusters,
- primitive d'attente de fin de simulation d'un des clusters esclaves,
- exécution sur un esclave d'une nouvelle simulation avec de nouveaux objets actifs.

Diverses restrictions doivent être prises en compte lors de la distribution de réplications. Tout d'abord, les exécutions ayant lieu sur des machines distantes, le code du modèle ne devra pas contenir d'entrées clavier, de sorties graphiques et une attention particulière doit être apportée à la manipulation du système de fichiers. Ensuite, l'exécution parallèle pourra avoir un comportement différent et/ou donner des résultats différents d'une réplication séquentielle dans la mesure où les esclaves ont chacun leur contexte et ne partagent pas le même environnement.

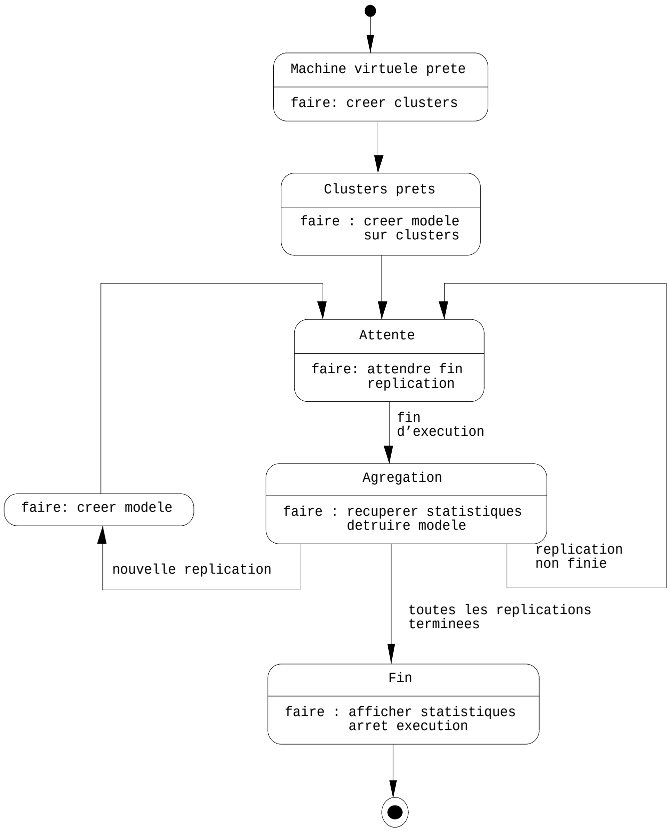


Fig. 9.4 – *Diagramme d'état du cluster principal lors d'une réplication*

10. COMPARAISON AVEC D'AUTRES MODÈLES DE PROGRAMMATION ET ENVIRONNEMENTS DE SIMULATION

Nous effectuons dans ce chapitre une étude comparative du modèle de programmation et de l'architecture PROSIT avec d'autres paradigmes de programmation et certains environnements de simulation.

10.1 Etude comparative du modèle de programmation

Le modèle de programmation à base d'objets actifs est proche du modèle de programmation de C++// [Caromel *et al.*, 1996], du modèle d'acteurs [Agha et Hewitt, 1987] et du modèle des objets réactifs proposé par Boussinot [Boussinot, 1996].
C++// est une extension de C++ développée au sein du projet Sloop de l'INRIA et dans le cadre du projet Européen EUROPA [Caromel et *al.*, 1997]. Le modèle de programmation PROSIT et celui de C++// possèdent de nombreuses similitudes. Tous deux définissent la notion d'objets actifs interagissant par le biais d'appels réifiés de méthodes. La définition d'un objet actif se fait aussi dans les deux modèles par le mécanisme d'héritage et le contrôle est centralisé. Il existe néanmoins des différences importantes entre les deux modèles. La première porte sur la définition des quatres points de réification qui n'est pas identique. Ensuite, seul PROSIT permet la multi-activité d'un objet et introduit la migration d'objet actif. Enfin, la notion de temps logique et de synchronisation automatique par le temps n'existe pas en C++// [1].
On peut aussi noter certaines similitudes entre le modèle PROSIT et le modèle d'acteur. En effet, un acteur est aussi un objet actif, autonome, communiquant avec ses semblables et dont le comportement est défini dans un script, équivalent à la fonction `behave()`. Les acteurs communiquent par messages qui peuvent être traités en séquence (on parle d'*acteurs sérialisés*) ou en parallèle. Mais le modèle proposé par Hewitt et Agha est à l'origine orienté vers l'intelligence artificielle, il ne possède donc pas de notion de temps et le seul mécanisme d'interaction est l'échange asynchrone de messages qui sont aussi des acteurs.
Nous retrouvons par contre la notion de temps dans la programmation réactive. L'approche réactive consiste à introduire une dimension supplémentaire dans la description des systèmes : celle d'un temps logique décomposé en instants. L'écoulement du temps, est perceptible de la même façon par tous les composants et tout se passe comme si une horloge centrale rythmait leur exécution. Le passage à l'instant suivant est global et n'a lieu que lorsque tous les composants ont terminé ce qu'ils avaient à faire dans l'instant courant. Ainsi, un composant qui a

1. en C++// seul l'appel non bloquant de méthode est disponible, la synchronisation se faisant par un mécanisme de *future*.

terminé son travail pour l'instant courant se met en attente de la fin de cet instant. Le schéma d'exécution de PROSIT est très similaire. Le noyau et l'ordonnancement mis en place assurent la gestion d'une horloge centrale et le temps est global à l'ensemble des objets actifs. Par contre, le temps est traité de manière différente en PROSIT qui ne possède pas de notion d'instant mais plutôt la notion de date, un objet pouvant se mettre en attente jusqu'à une certaine date. La progression de l'horloge est aussi variable. Dans le modèle objet réactif proposé par Boussinot, les systèmes sont formés d'objets qui s'exécutent en parallèle. Les méthodes sont des traitements qui évoluent d'instant en instant. Une même méthode ne peut être exécutée plusieurs fois au cours du même instant. Un objet a fini de s'exécuter pour l'instant courant, lorsque toutes ses méthodes ont fini leur exécution pour cet instant. Un instant du système est terminé lorsque tous les objets ont terminé leur exécution pour cet instant. Une nouvelle différence apparaît ici, au sein d'un objet actif PROSIT, la même méthode peut être exécutée simultanément par plusieurs activités et son exécution se déroule dans le temps. De même, contrairement aux objets réactifs, une demande de service peut être bufférisée, servie ultérieurement, ou rejetée instantanément ou ultérieurement. Le modèle possède aussi une extension distribuée (voir [Boussinot *et al.*, 1995] §6). Les objets réactifs sont répartis en sous-systèmes réactifs. Contrairement à PROSIT, le modèle distingue les interactions locales, qui respectent le principe d'*interaction instantanée*, des interactions distantes qui sont asynchrones et prennent un temps non-borné.

Le langage Simula [Lamprecht, 1981] a aussi été une source d'inspiration lors de la conception de notre modèle de simulation. Les mécanismes et les primitives de base sont sensiblement les mêmes, mais PROSIT offre un niveau d'abstraction supérieur. De plus nous avons une architecture plus uniforme dans laquelle toutes les entités manipulées sont des objets (demandes de service, activités, etc.).

Au niveau de l'architecture distribuée, certains environnements fournissent des primitives de migration d'objets. C'est le cas du système SOS du projet SOR de l'INRIA [Shapiro *et al.*, 1989b, Shapiro *et al.*, 1989a], la migration est aussi disponible dans la couche Chorus/COOL [COO, 1991] développée au dessus du micro-noyau Chorus et dans le langage Emerald [Black *et al.*, 1988]. De même la notion de cluster présente dans PROSIT est proche de celle de *contexte* du système SOS, de celle de *d'environnement de processus* de PT-PVM [Krone *et al.*, 1995], de celle de *cluster* de *microC++* [Buhr et Stroobosscher, 1991]. En COOL V2 un *cluster* est un groupe d'objets fortement liés construit au niveau de l'application. *Amadeus* [Cahill *et al.*, 1993] possède aussi des clusters qui sont crées par la commande *startnewcluster*. Dans Emerald il y a une notion cachée de cluster, car les objets sont regroupés en les *attachant*.

10.2 Etude comparative du modèle de simulation

Parmi les environnements de simulations étudiés, nous avons retenu pour cette comparaison ceux possédant un véritable langage de programmation, de préférence objet, et permettant de simuler toute classe de systèmes discrets[2]. Ces environnements sont : Awesime [Grunwald, 1991], C++SIM [Little et McCue, 1993], SIM*PlusPlus* [Rose, 1992], SIM++ [Baezner et Lo-

2. Nous n'avons pas retenu certaines librairies de simulation (par exemple Cedes [Hu, 1993], Disc++ [Blair et Selvaraj, 1989], Prism [Vaughan *et al.*, 1991], EXsim [Chow et Zheng, 1993], Simile [Brasileiro *et al.*, 1993], etc.) soit parce qu'elles étaient dédiées à un domaine d'applications, soit parce qu'elles n'avaient fait l'objet que de peu de recherche et développement.

mow, 1989], *Si* [Sang *et al.*, 1994], Maisie [Bagrodia et Liao, 1990], SimPack [Cubert et Fishwick, 1995][3], Simex [Altmann *et al.*, 1995] et Ptolemy [Buck *et al.*, 1992]. Comme critères d'analyse, nous avons sélectionné l'architecture des modèles, leur mode de programmation et la technique de distribution. Nous donnons à la table 10.1 un récapitulatif des environnements de simulation étudiés et à la table 10.2 une présentation de l'intégration de la distribution.

10.2.1 Architecture des modèles et programmation

La grande majorité de ces environnements fournissent une approche orientée processus de la simulation (Awesime, C++SIM, SIM*PlusPlus*, SIM++, *Si* et Maisie.), seuls SimPack et Ptolemy proposent l'approche orientée événement et Simex supporte les deux approches. L'introduction de la notion de processus se fait en intégrant dans le modèle objet le concept d'activité. Mais cette intégration prend différentes formes. *Si* introduit les processus légers. Awesime, C++SIM, SIM++ introduisent la notion d'objet actif. Simex introduit la notion d'objets possédant plusieurs activités. Les activités de Simex peuvent être représentées par des processus ou des automates. Sim*plusplus* introduit la notion de méthode active.

Dans tous ces environnements, la simulation est vue comme l'interaction, dans le temps, d'un ensemble d'entités. Ces interactions peuvent se faire par le biais d'appels de méthodes (Awesime, C++SIM et SIM*plusplus*) ou alors prendre la forme plus structurée d'échanges de messages (Maisie) ou d'événements (SIM++, *Si* et Simex). En Ptolemy, les composantes d'une simulation s'échangent une structure de données appelée *particule*. Le seul mode d'interaction supporté par ces environnements est le mode asynchrone.

Plusieurs variantes existent aussi pour la synchronisation des entités. Elles peuvent se synchroniser par le biais de sémaphores (Awesime, C++SIM, SIM*PlusPlus*), ou par le biais d'événements (Simex). Une autre forme de synchronisation, est celle qui consiste à se mettre en attente conditionnelle de messages (Maisie) ou d'événements (Sim++). En *Si* les entités se synchronisent aussi bien à l'aide de sémaphores que d'événements. Le mécanisme d'attente d'un message ou d'un événement peut être très simple, comme en Sim++ où il est uniquement possible d'attendre le prochain événement, ou alors un peu plus puissant, comme *Si* qui permet d'attendre un événement d'une classe donnée. Mais c'est Maisie qui offre le mécanisme le plus intéressant car la condition de reprise est une expression booléenne pouvant avoir comme variables les données de l'entité et les messages.

De tous ces environnements seul Ptolemy est à la base orienté vers la simulation hétérogène et hiérarchique. L'approche retenue dans Ptolemy est non dogmatique et basée sur une définition simple de la simulation qui est vue comme un ensemble de traitements à exécuter dont l'enchaînement dépend de critères évalués par un ordonnanceur. Le concept de traitements est supporté par la classe `Block`, et l'approche hiérarchique structure le modèle en `Star`, `Galaxy` et `Universe`. Chaque `Block` possède un ordonnanceur, `Scheduler`, et communique avec les autres `Block` par le biais de *particules*. La mise en œuvre d'une simulation hétérogène peut intervenir en spécialisant l'ordonnanceur d'un `Block`, ou lors des transformations des particules en entrée et en sortie d'un `Block`.

L'approche objet actif de Prosit n'est pas spécifique et se retrouve dans plusieurs environ-

3. La version C++ de cette librairie se nomme aussi SIM++, afin de ne pas la confondre avec celle développée par Baezner et Lomow, nous avons conservé le nom SimPack qui est celui de la version C.

Environnement	Dynamique	Objet actif	Synchronisation	Interactions
Awesime	processus	oui, classe `Thread` exécute `main()`	sémaphore	appel de méthode
C++SIM	processus	oui, classe `Process` exécute `body()`	sémaphore	appel de méthode
SIM*PlusPlus*	processus	non, méthode active	sémaphore	appel de méthode
SIM++	processus	oui, classe `sim_entity` exécute `body()`	attente conditionnelle d'événements	événement
Si	processus	pas d'objet car programmation en C	attente conditionnelle d'événements et sémaphore	événement
Maisie	processus	notion d'`entity`	attente conditionnelle de messages	messages
SimPack	événement	Non	échange de *Token*	échange de *Token*
Simex	événement et processus	objet ayant plusieurs activités	événement	événement
Ptolemy	événement	Non. Notion d'objet ordonnanceur	échange de *particules*	échange de *particules*
PROSIT	processus	objet ayant plusieurs activités	demande bloquante de services	demande de services

TAB. 10.1 – *Récapitulatif des environnements de simulation*

nements. Par contre, la possibilité d'avoir des objets avec plusieurs activités n'est présente que dans Simex. Le mécanisme de contrôle hiérarchique où le noyau gère les objets actifs et chaque objet gère ses activités annexes, est propre à PROSIT et respecte le principe d'encapsulation. Ptolemy propose aussi un modèle hiérarchique mais uniquement avec des objets passifs.

En PROSIT, les seules interactions entre les entités est la demande de service. L'unification des demandes de service aux appels de méthodes et la réification sont deux caractéristiques qui ne se retrouvent que dans nos propositions. Nous considérons qu'elles permettent d'obtenir une meilleure abstraction des mécanismes de simulation et une plus grande puissance dans la programmation. Nous pouvons par exemple ajouter au framework, l'ensemble des mécanismes d'interaction présents dans les environnements étudiés.

Au niveau de la synchronisation, le seul mécanisme introduit par PROSIT est la nature bloquante ou non des demandes de service. Les sémaphores présents dans Awesime, C++SIM et SIM*PlusPlus* peuvent être mis en œuvre par une classe d'objets actifs spécifiques[4]. Nous avons cherché à limiter le nombre de mécanismes à la base du modèle de programmation PROSIT afin de faciliter sa mise en œuvre et surtout son utilisation. L'architecture framework permet ensuite de programmer les primitives désirées. Ainsi, l'attente conditionnelle de Maisie peut très facilement être disponible en codant au niveau de la classe active une fonction évaluant l'expression booléenne et mettant l'objet actif en mode *sleep* tant que la condition n'est pas vérifiée (voir l'exemple de code 10.1[5]).

4. C'est par exemple le cas dans la librairie pour la simulation de réseaux de files d'attente (cf. §13.1.1).

5. La technique illustrée ici consiste à coder l'expression booléenne comme une fonction membre de la classe et à la passer sous forme de pointeur sur fonction membre, une autre technique consisterait à utiliser un *objet-fonction* (voir [Stroustrup, 1993] p. 350).

Exemple de code 10.1 Primitive d'attente sur une condition booléenne

```
typedef bool (Maisie_Obj:: *bool_expr)();
void Maisie_Obj::wait_for(bool_expr func) {
  while(!(this->*func)()) sleep();
}
```

La réalisation de simulations hiérarchiques et hétérogènes est très similaire en Prosit et Ptolemy. Ces deux environnements ont les mêmes objectifs et de nombreux points communs. Tous deux intègrent les mécanismes nécessaires sous la forme d'un framework. La hiérarchisation des ordonnanceurs dans Prosit est inspirée de l'approche adoptée dans Ptolemy. L'intégration de formalismes différents est sémantiquement la même puisqu'en Prosit il faut coder des classes interfaces ce qui est similaire à la conversion des *particules* de Ptolemy. Par contre, le modèle d'exécution de Prosit est très différent et nous introduisons bien moins de concepts, ce qui facilite la programmation. Au niveau de la puissance d'expression, les deux environnements sont comparables. Mais Ptolemy est un projet plus avancé et a fait l'objet de développements importants, il intègre déjà de nombreux paradigmes (principalement pour la simulation de circuits logiques). Il nous reste encore à valider le modèle Prosit par d'autres expériences d'intégration. Les premières réalisations présentées au chapitre 9 ont déjà permis de valider partiellement notre approche.

10.2.2 Techniques de distribution

Comme exemple d'environnements supportant la simulation distribuée, nous avons Maisie [Bagrodia et Liao, 1990], Warped [Marti *et al.*, 1996], TWOS [Reiher et Jefferson, 1990], *Si* [Sang *et al.*, 1996] et SIM++ [Baezner *et al.*, 1990]. SIM++, Warped et TWOS utilisent la synchronisation optimiste, *Si* utilise la synchronisation conservatrice, Maisie selon [Bagrodia et Liao, 1990] supporte les deux.

Parmi ces environnements, seul *Si* permet une forme de migration dynamique de processus similaire à Prosit. TWOS est présenté comme un système d'exploitation dédié et permet la migration de "*phases*". Cette approche est très intéressante dans la mesure où, lors de la migration d'un objet seule une partie de son historique migre, limitant ainsi le volume des communications.

Peu de systèmes permettent à la fois la simulation séquentielle et la simulation distribuée (Maisie, *Si* et SIM++) et seul Maisie semble supporter à la fois la synchronisation optimiste et conservatrice. TWOS et Warped, comme Prosit, sont des environnements d'étude permettant l'implémentation, l'évaluation et l'optimisation de protocoles de synchronisation (ici Time Warp). Warped, comme Prosit, est organisé sous la forme d'un framework et permet de spécialiser de nombreux mécanismes. Mais l'architecture de Prosit reste plus modulaire, permettant notamment d'intégrer facilement divers protocoles de synchronisation.

Environnement	Support de la distribution
Awesime	Activation en parallèle sur une machine multiprocesseurs à mémoire partagée des objets `Thread` ayant la même date
SIM++	Une version Time Warp a été prototypée. Les entités sont regroupés en clusters.
Si	Synchronisation conservatrice
Maisie	Une version distribuée a été prototypée
Ptolemy	La simulation est séquentielle. Les composantes de l'environnement sont distribuées (animation, etc.).
Warped	Framework mettant en œuvre le protocole Time Warp
TWOS	*Système d'exploitation* mettant en œuvre le protocole Time Warp
PROSIT	Les objets actifs sont distribués sur les clusters. Le modèle supporte la synchronisation conservatrice et optimiste (seule la synchronisation conservatrice est implémentée).

TAB. 10.2 – *Environnements de simulation intégrant la distribution*

Troisième partie

RÉALISATION DE L'ENVIRONNEMENT PROSIT

11. SIMULATION SÉQUENTIELLE

Ce chapitre présente l'implémentation du noyau séquentiel réalisée. Nous donnons tout d'abord une vue générale de l'architecture et de la gestion du temps, puis nous abordons les points spécifiques de l'implémentation. Tout au long de cette présentation, nous insistons sur l'aspect modulaire et interchangeable des structures mises en place. En effet, chaque composante peut être redéfinie et spécialisée afin d'adapter l'environnement soit à une simulation spécifique, soit à une nouvelle structure de données ou à un nouvel algorithme.

11.1 Architecture des composants et gestion du temps

11.1.1 Architecture générale

L'environnement de simulation séquentiel repose sur quatre couches et un ensemble d'outils tels que représentés sur le schémas d'architecture générale 11.1. Les quatre couches s'appuyant l'une sur l'autre sont :

- la couche spécifique à une implémentation fournissant le runtime de base ainsi que la librairie de processus légers ;
- les classes de simulation qui permettent de définir et de gérer les objets actifs dans le cadre de la simulation à événements discrets. Cette couche fournit aussi divers noyaux de simulation ;
- les classes de modélisation pour représenter les différents paradigmes de modélisation rencontrés : files d'attente, réseaux de Petri, etc ;
- les classes d'application qui constituent des macro-modèles ou des briques de base pour pouvoir simuler différentes applications d'un même domaine (système de production, système de communication, etc.).

Les bibliothèques de modélisation ainsi que les outils de présentation et d'analyse sont détaillés au chapitre 13.

11.1.2 Architecture du noyau

Dans l'architecture mise en œuvre, le noyau de simulation se comporte comme un ordonnanceur d'objets actifs. L'ordonnancement est réalisé en fonction de la date de chaque objet. Pour cela, le noyau gère un échéancier des objets en attente d'exécution ainsi qu'une liste des objets endormis ou bloqués. Chaque objet désirant avancer sa date est suspendu et inséré

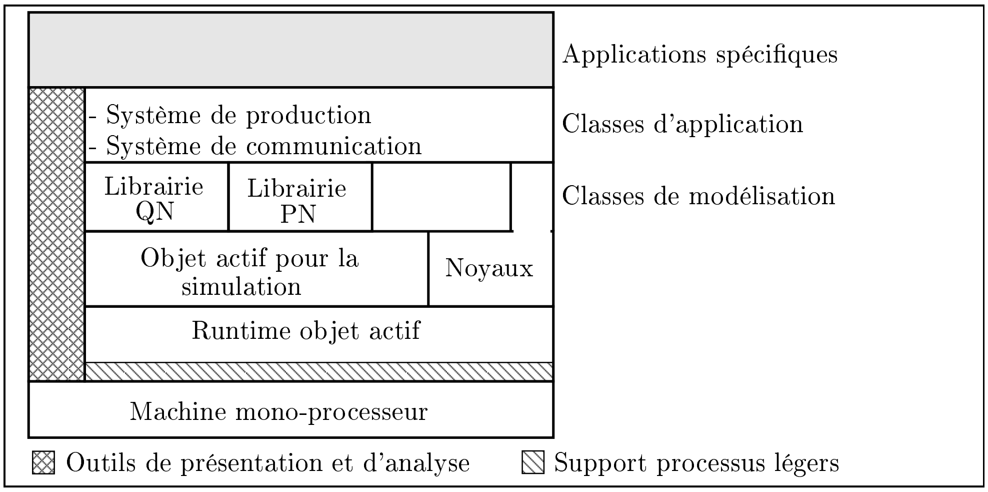


FIG. 11.1 – *Architecture générale de l'environnement séquentiel*

dans l'échéancier. Au cours de la simulation, le noyau extrait de l'échéancier l'objet ayant la plus petite date, met à jour son horloge à la date de l'objet et le réactive. Quand l'objet est à nouveau suspendu, plusieurs cas sont à distinguer :

- l'objet a demandé à consommer du temps, le noyau le réinsère dans l'échéancier ;
- l'objet est bloqué ou en attente passive, le noyau l'insère dans la liste des objets inactifs ;
- l'objet est terminé, il n'est plus géré par le noyau et est éventuellement détruit.

Nous donnons à l'exemple de code 11.1 le comportement du noyau séquentiel.

Exemple de code 11.1 Comportement du noyau séquentiel

```
void Kernel::behave() {
  for (act_obj=next_obj_to_run();act_obj;act_obj=next_obj_to_run()) {
    if(end_simulation_condition()) break;
    set_time(act_obj->get_time());
    act_obj->run();
    re_schedule(act_obj);
  }
  cout << "Fin de la simulation a " << get_time() << endl;
}
```

La figure 11.2 présente l'architecture du noyau qui comprend :

- un échéancier. Cet échéancier doit être l'instance d'une classe descendante de la classe abstraite `Scheduling_List`. Cette dernière définit la signature des primitives de manipulation (insertion, retrait et itérateur). Nous avons déjà implémenté un échéancier linéaire (`Linked_Scheduling_List`) et un échéancier *splay-tree* [Sleator et Tarjan, 1985] (`Splay_Scheduling_List`). D'autres structures de données (voir par exemple [Brown, 1988, Franta et Maly, 1977, Pugh, 1990]) peuvent être utilisées, mais la structure splay-tree est dans le cas général très efficace (voir par exemple les études de complexité et de performance [Chung *et al.*, 1993, Duval et Vaucher, 1975, Jones, 1986]) ;

– une liste des objets actifs non ordonnancés (statut *bloqué* ou *endormi*). Comme pour l'échéancier, cette composante est représentée par une classe abstraite, la classe `Sleeping_List`. La version actuellement implémentée est codée sous forme de table de hash (classe `Hash_Sleeping_List`) ;

– un module statistique. Ce module, éventuellement absent, est chargé de l'ensemble des calculs statistiques. Il est présenté à la section 13.2.2 ;

– un gestionnaire de trace. Ce module, éventuellement absent, est chargé de la génération et du filtrage des traces. Il est présenté à la section 13.2.1 ;

– de divers générateurs (identifiant, nombre aléatoire, etc.).

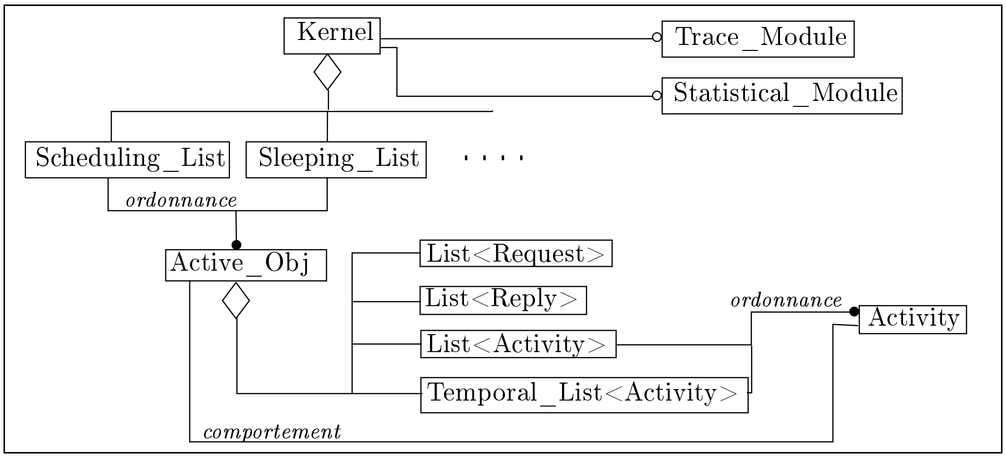


FIG. 11.2 – *Diagramme de classes pour le noyau et les objets actifs*

Nous avons aussi réalisé quelques prototypes de classes noyaux implémentant des traitements spécifiques (cf. Fig. 11.3). Nous avons, par exemple, codé un noyau capable de *swapper* (*i.e.* sauvé sur disque) les activités lors de la simulation de modèles trop grands pour tenir en mémoire. Un autre exemple consiste à rajouter des mécanismes de suivi de l'exécution et de collecte d'informations (nombre d'activations, temps de calcul utilisé par objet actif, etc.) afin d'être en mesure d'optimiser et de déboger une simulation.

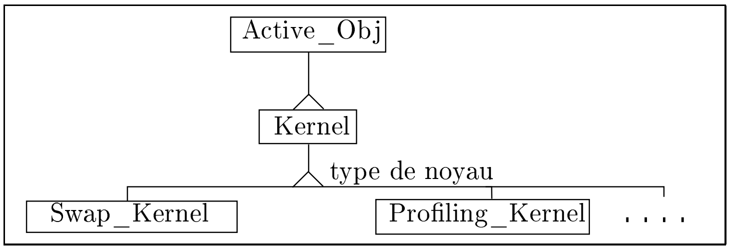


FIG. 11.3 – *Hiérarchie des classes noyaux séquentiels*

11.1.3 Architecture d'un objet actif

Les objets actifs (cf. Fig. 11.2) sont des instances de classes héritant directement ou indirectement de la classe `Active_Obj`. Un objet actif comprend :

- un identifiant unique et un nom alphanumérique. Le nom de l'objet est essentiellement utilisé pour les traces et le débogage ;
- une activité principale qui est automatiquement créée à la construction de l'objet et qui est associée à la méthode `behave()` ;
- une liste des requêtes et une liste des réponses non encore traitées ;
- une liste pour les activités annexes ayant le statut *running* et une autre liste pour celles ayant le statut *blocked*.

11.1.4 Architecture d'une activité

Une activité est toujours rattachée à un objet actif et est représentée par la classe `Activity` (cf. exemple de code 11.2) qui comprend : un identifiant unique, une pile, l'identifiant de la fonction en cours d'exécution, et éventuellement un pointeur sur la requête en cours de service.

Exemple de code 11.2 La classe Activity (partielle)

```
class Activity : public Temporal_Obj {
private:
  MID mid;
  Request *attached_request;
  ActivityStatus status;
  Active_Obj *owner;
  Activity_Id id;
  Stack *stack;
public:
  void run();
}; // end Activity
```

11.2 Flot d'exécution

Nous reprenons les étapes composant une simulation (cf. section 8.3.2) en détaillant cette fois ci les mécanismes mis en œuvre.

11.2.1 Lancement et arrêt d'une simulation

Le lancement de la simulation consiste à passer d'un mode *mono-activité*, dans lequel une seule fonction s'exécute à la fois (c'est le mode normal de programmation), à un mode *multi-activités*, dans lequel les objets actifs s'exécutent sous le contrôle du noyau (cf. Fig. 11.4). Le passage en mode multi-activités se fait en appliquant la méthode `run()` au noyau. Le retour en mode mono-activité intervient quand l'activité principale du noyau termine ; les instructions suivant l'appel à la primitive `run()` sont alors exécutées. L'activité du noyau termine s'il n'y à plus d'objets à activer ou si la condition d'arrêt de la simulation est vérifiée (primitive `end_simulation_condition()`).

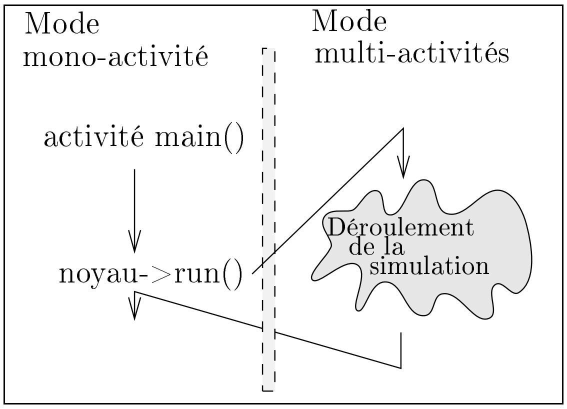


Fig. 11.4 – *Passage du mode mono-activité au mode multi-activités.*

11.2.2 Déroulement d'une simulation

Une fois en mode multi-activités, le noyau ordonnance et gère les objets actifs (cf. Fig. 11.5). A chaque activation, le comportement de l'objet se charge, si besoin, de réactiver ses activités annexes (cf. Fig. 11.6). Dans la version actuelle du simulateur, il n'y a qu'un seul objet actif à la fois et qu'une seule activité à la fois en cours d'exécution. Nous n'avons donc pas une véritable concurrence des activités, mais un entrelacement de ces dernières sous forme de co-routines. Nous avons sélectionné ce mode d'exécution, car nous considérons qu'il est actuellement plus facile à utiliser. En effet, il n'y a pas d'accès concurrents aux données, le code est donc plus facile à écrire et il est possible d'utiliser les librairies existantes et les techniques classiques de mise au point et d'optimisation. L'utilisation de co-routines permet aussi de contrôler le flot d'exécution et donc d'assurer une exécution déterministe de la simulation [1].

11.3 Support d'exécution

11.3.1 Implémentation des activités

Les activités sont implémentées à l'aide d'une librairie de processus légers non préemptifs développée à l'Université de Washington par D. Keppel [Keppel, 1993]. Cette librairie est organisée en deux couches. La première fournit un ensemble de fonctions C gérant les changements de contexte. Ces fonctions sont interfacées avec quelques instructions assembleurs se chargeant de la sauvegarde/restauration des registres du processeur. Le second niveau fournit les mécanismes de base d'un ordonnanceur de processus légers (structure de données, politique

1. Le lecteur intéressé trouvera dans [Sang *et al.*, 1994] la présentation d'une technique permettant d'obtenir le déterminisme en utilisant des processus légers préemptifs.

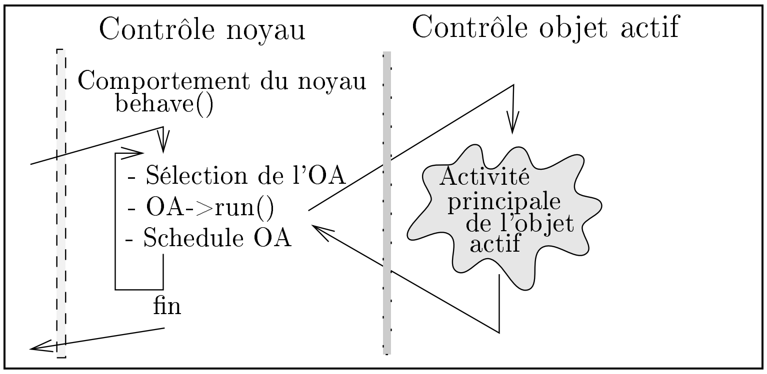


Fig. 11.5 – *Flux de contrôle en mode multi-activités.*

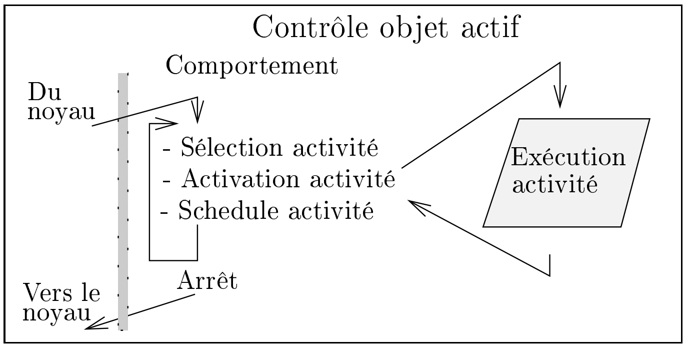


Fig. 11.6 – *Flux de contrôle au sein d'une entité active.*

d'ordonnancement, etc.). Pour Prosit, nous n'avons utilisé que le premier niveau en l'encapsulant dans la classe `C_Kernel`. Cette encapsulation isole notre noyau des mécanismes de bas niveau, ce qui facilite le portage. Nous avons retenu cette librairie pour plusieurs raisons :

– Elle était, au début du développement du framework, l'unique librairie de processus légers disponible sur l'ensemble des architectures et des systèmes d'exploitation. Les librairies POSIX faisant à peine leur apparition.

– Elle était la librairie la plus performante. De plus, sa mise en œuvre simple et structurée permettait une intégration optimale à prosit.

– La disponibilité des sources et leur simplicité nous permettait de mener de nombreuses expériences (migration, persistence, ordonnancement, etc.).

– Elle offre des processus légers non préemptifs. L'approche non préemptive correspond bien au modèle de programmation Prosit (notion de point d'arrêt d'une activité, cf. §6.4.1 page 49). De plus, le contrôle des points de changement de contexte facilite le développement et l'intégration de code (le code n'ayant pas besoin d'être réentrant) et permet des exécutions plus rapides (il n'y a pas de changement de contexte inutile).

Au niveau de l'implémentation, une activité est composée d'une pile (zone mémoire), du contexte du processeur (stocké en sommet de pile) et d'un pointeur de sommet de pile. Les classes `Activity` et `Stack` fournissent une approche structurée de ces mécanismes. Un point d'arrêt d'une activité comprend la mise à jour des informations internes (statut, date, etc.) et un changement de contexte afin de réactiver une autre activité (activité principale de l'objet ou du noyau). Afin d'optimiser la gestion mémoire (réduction du nombre d'allocations/libération, réduction de la fragmentation), nous avons mis en place un gestionnaire de pile chargé de stocker les piles inutilisées et d'en fournir à la demande.

11.3.2 Mise en œuvre des interactions

Demande de service

La syntaxe introduite pour les demandes de service est traitée par un pré-processeur qui se charge d'expanser le code. L'expansion consiste à créer et initialiser l'objet requête et à le transmettre. Les paramètres de contrôle sont directement utilisés sous forme d'appels de méthodes sur l'objet requête. Ainsi l'appel :

```
sync station1->service(param) :priority(20) :max_service_time(78);
```

est transformé en :

```
Request *rq =
        noyau->build_sync_request(station1,station1->service_(param),1,param);
rq->priority(20);
rq->max_service_time(78);
send_request(rq);
```

L'objet requête est créé et partiellement initialisé par le noyau. L'initialisation sera complétée par le point de réification (méthode `send_request()`). La création de la requête par le noyau permet d'introduire aisément de nouvelles caractéristiques. Pour cela, il faut dériver de nouvelles classes requêtes et surcharger la méthode de création sur une nouvelle classe noyau. Signalons l'existence d'une classe pour les requêtes bloquantes (`Sync_Request`) et d'une classe pour les requêtes non bloquantes (`Async_Request`) et donc de deux méthodes de création sur le noyau. L'introduction de ces deux classes simplifie le code et améliore les performances.

La solution proposée à l'avantage d'être facilement implémentable dans la mesure où la modification du code est purement syntaxique, le pré-processeur n'effectuant aucune analyse sémantique. De plus, le code produit est simple et lisible ce qui autorise le débogage direct et la génération manuelle en l'absence du pré-processeur. Un autre avantage de cette approche est la possibilité d'avoir de nouveaux paramètres de contrôle sans modification du pré-processeur.

Modélisation des routines

Nous avons vu que la programmation explicite du contrôle d'un objet actif nécessitait la modélisation et la manipulation des routines. Pour cela, nous avons introduit la classe `MID` qui regroupe l'ensemble des primitives nécessaires.

Dans la version actuelle de l'environnement, à chaque fonction service doit correspondre une fonction de même signature, ayant le même nom mais suivi du caractère `_` (par exemple `service_()` pour `service()`), et retournant l'objet MID modélisant la routine (voir l'exemple de code 11.3). Cette fonction est utilisée lors de la construction de l'objet requête et permet à la fois d'automatiser la construction et de laisser au compilateur la résolution de la surcharge.

Exemple de code 11.3 Construction d'un objet MID

```
MID Porte::passer_() {
   void (Porte::* ptr)()=&Porte::passer;
   return (MID((ptr_Routine)ptr));
}
```

Statut de terminaison d'un service

Comme indiqué à la section 6.2.4, nous utilisons le mécanisme standard d'exception de C++. Lors d'une non-terminaison d'un service, une exception est levée dans le contexte du demandeur, c'est à ce dernier de la récupérer et de la traiter.

Exemple de code 11.4 Récupération du statut de terminaison d'un service à l'aide d'une exception

```
try {
   sync station1->service(23) :priority(20);
}
catch (Service_Exception &exception) {
   // traitant de non terminaison
}
```

Une hiérarchie des classes d'exceptions de base est fournie permettant de gérer les différentes situations. Cette hiérarchie peut être étendue afin d'affiner la modélisation (ex : différencier les causes de rejet, etc.).

Là encore, nous proposons une implémentation de ce mécanisme simple et directe. Lors d'une terminaison anormale, l'objet `Reply` émis contient l'instance qui sera utilisée pour lever l'exception dans le contexte du demandeur. L'exception sera levée lors de la réactivation de ce dernier. Les librairies pourront fournir diverses primitives de rejet de requêtes ou d'arrêt de services (par exemple `reject_request(Request *, Exception *)`, `abort_service()`, etc.).

Ce mécanisme peut sembler contraignant, car chaque appel de service doit être mis dans un bloc `try` suivi de un ou de plusieurs blocs `catch`. Mais le programmeur peut toujours regrouper plusieurs appels dans un seul bloc `try`. De plus, à l'aide d'une option, le pré-processeur pourrait insérer le code de récupération d'exception, mais les exceptions récupérées ainsi que les traitements effectués ne pourraient être que des comportements par défaut.

11.4 Description de l'implémentation

11.4.1 Organisation du code

Le framework séquentiel est organisé en un ensemble de librairies correspondant aux couches de la figure 11.1. Nous avons :

- la librairie support de processus légers, qui est la librairie développée par l'Université de Washington ;
- la librairie comprenant les classes de simulation (38 classes, environ 4000 lignes de code) ;
- les librairies comprenant les classes de modélisation pour les files d'attente (35 classes, environ 4000 lignes de code) et pour les réseaux de Petri (27 classes, environ 2000 lignes de code). Ces librairies seront détaillées au chapitre 13.

11.4.2 Matériels et logiciels utilisés

L'environnement Prosit est porté sur l'ensemble des plates-formes et des systèmes d'exploitation suivants : architecture INTEL et systèmes LINUX et Windows, alphastation et système Digital UNIX, architecture SUN et systèmes SunOS 4 et SunOS 5 (Solaris). La programmation est réalisée en C++ et le compilateur utilisé est celui fourni par *GNU* (`g++`). La librairie de classes *GNU* a aussi été utilisée le plus souvent possible (générateur de nombre aléatoire, structures de données –table de hash et liste–, etc.).

12. SIMULATION DISTRIBUÉE

Nous présentons dans ce chapitre l'implémentation réalisée du modèle distribué. Après avoir introduit l'architecture générale, nous présentons la librairie de communication utilisée et la mise en œuvre du framework au dessus de celle-ci. Nous détaillons ensuite le mécanisme de migration des objets actifs, puis les protocoles de synchronisation implémentés.

12.1 Architecture

Comme dans le cadre séquentiel, l'environnement distribué est organisé en couches (cf. Fig. 12.1). Seuls les deux premiers niveaux (support pour la communication et la distribution, *runtime* objet actif) sont spécifiques à une implémentation. Les quatre premières couches sont :

- la couche fournissant les primitives de base pour la gestion des processus et la communication,
- la couche fournissant les mécanismes de distribution, d'interaction et de migration pour les objets actifs,
- les classes de simulation qui permettent de définir et de gérer les objets actifs dans le cadre de la simulation distribuée,
- les classes spécifiques à un mode de synchronisation (par exemple synchronisation conservative, optimiste, etc.).

L'environnement comprend aussi des outils de présentation et d'analyse des résultats, ainsi que des outils de gestion du parallélisme. Ces derniers sont essentiellement graphiques, et permettent le contrôle visuel et la mise au point des simulations. Nous présentons ces outils au chapitre 13.

En début de phase d'implémentation, nous avons étudié et évalué plusieurs supports pour les niveaux 1 et 2. Nous présentons dans la suite, les environnements étudiés en indiquant à chaque fois leurs principales caractéristiques. Les évolutions très rapides de ces environnements ont nécessité la réactualisation de l'étude et certains choix, bien que non retenus à l'origine, seraient maintenant de bons candidats pour une nouvelle implémentation du framework.

12.1.1 Architectures envisagées

Nous avons évalué pour l'implémentation de Prosit, six supports de communication et d'exécution (cf. Fig. 12.2). Ces supports sont de natures différentes et ne fournissent pas les

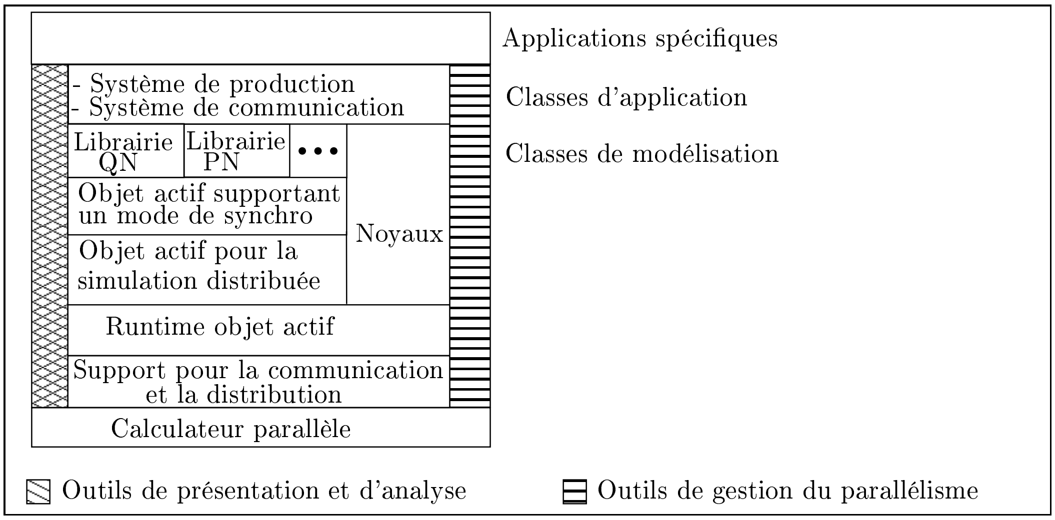


Fig. 12.1 – *Architecture générale de l'environnement distribué*

mêmes fonctionnalités. Ils peuvent être divisés en trois catégories : les librairies de communication (socket, PVM et Schooner), les langages distribués (Java et C++//) et enfin les environnements de programmation distribuée (Corba).

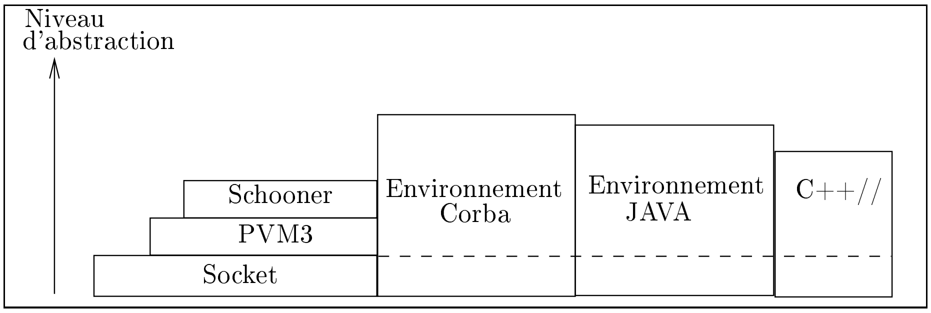


Fig. 12.2 – *Supports d'exécution pour* Prosit

Socket La première possibilité consistait à développer Prosit directement au dessus de l'interface socket [Stevens, 1990]. Cette approche aurait apporté de bonnes performances, mais n'a pas été retenue en raison de son trop bas niveau de programmation. En effet, il aurait fallu développer de nombreux mécanismes (gestion des processus lourds, encodage/décodage des données, etc.) et la quantité de travail supplémentaire sortait très largement du cadre de cette thèse.

PVM PVM [Beguelin *et al.*, 1994] est actuellement le standard de-facto des librairies de communication (concurrencé depuis peu par MPI [Forum, 1993]). Le modèle de programmation PVM est basé sur l'envoi de messages. Son modèle d'exécution repose sur un ensemble dynamique de processus lourds communiquant par échanges asynchrones de messages. Les processus peuvent être hétérogènes, PVM fournit les primitives nécessaires pour l'encodage des données. Les deux principales qualités de PVM résident dans sa disponibilité sur un grand nombre de systèmes (plus de 60 portages sur 30 architectures différentes) et sa capacité à gérer des configurations dynamiques. Nous ne l'avons pas directement utilisé car nous considérons qu'il fournit un modèle d'interaction de trop bas niveau.

ORB L'architecture CORBA (Common Object Request Broker Architecture) [OMG, 1991, OMG, 1992] spécifiée par l'OMG (Object Management Group) permet le développement d'applications réparties. Cette architecture connaît depuis peu (2 à 3 ans) un succès croissant. Un objet en CORBA est placé sur un serveur et peut à la fois émettre et recevoir des requêtes/réponses d'autres objets placés sur le même serveur ou sur des serveurs distants. Les interactions entre les objets sont gérées par un bus logiciel, l'ORB (Object Request Broker). Grâce à l'ORB, le client est totalement abstrait des mécanismes de communication et d'activation des objets. C'est l'ORB qui gère l'ensemble des relations client/serveur entre objets.

La technique de mise en œuvre utilisée (cf. Fig 12.3) consiste à décrire l'interface de l'objet à l'aide d'un langage standardisé, l'IDL (Interface Definition Language), puis à compiler cette interface afin d'obtenir une *classe proxy* dont l'instance sera le représentant local de l'objet distant, et une *classe squelette* dont l'implémentation de l'objet devra hériter. Pour effectuer une requête, le client peut soit utiliser une invocation statique via le proxy local de l'objet, soit la construire dynamiquement. L'ORB se charge de localiser l'objet référencé et d'invoquer la méthode à travers le squelette. Le client et l'implémentation de l'objet peuvent accéder directement à l'ORB à travers une interface spécifique. Une seconde interface d'accès à l'ORB, l'adaptateur d'objet, est réservée à l'implémentation de l'objet (enregistrement, désactivation d'objets, etc.).

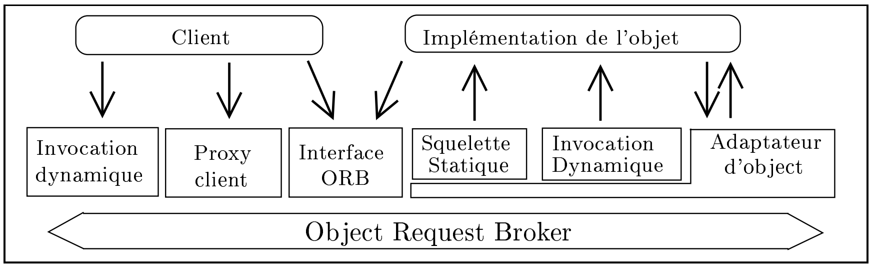


FIG. 12.3 – *L'architecture CORBA*

En plus de la gestion des interactions, l'ORB fournit aussi divers services (cf. Fig. 12.4) directement intégrés à l'architecture et possédant une interface IDL. Ces services font partie de l'Object Management Architecture (OMA).

De nombreuses raisons nous ont amenées à ne pas utiliser CORBA pour le premier prototype PROSIT :

- les ORB ne fournissent généralement pas la possibilité d'intercepter l'émission et la réception des requêtes et des réponses. Or ce mécanisme est nécessaire à la mise en œuvre de la réification ;

- CORBA ne propose pas de support pour la migration. Néanmoins certains services du niveau OMA peuvent être utilisés. Peter et Milhau présentent dans [Peter et Milhau, 1996] les mécanismes permettant d'obtenir un premier niveau de mobilité ;

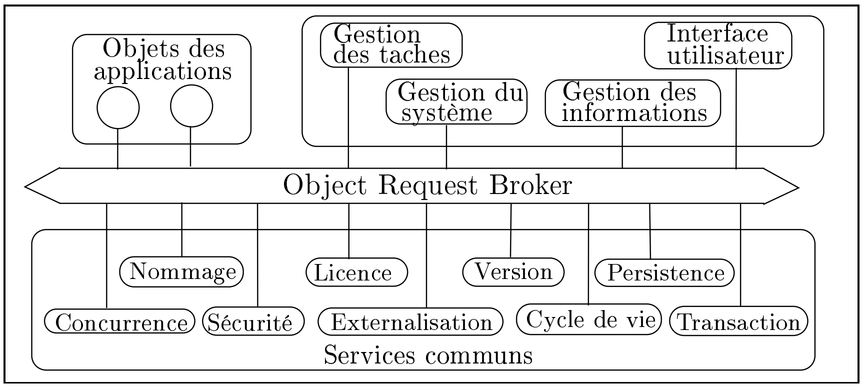


Fig. 12.4 – *L'architecture OMA*

- CORBA ne propose pas de modèle d'exécution. L'intégration de processus légers est fortement dépendante du produit utilisé. Signalons que le niveau OMA fournit des services de gestion de la concurrence et de transactions ;
- on ne contrôle pas les communications, il est alors difficile d'utiliser de manière transparente certaines techniques d'optimisation comme la bufférisation, la diffusion, etc. De plus, les premières expériences (voir par exemple [Kuhl *et al.*, 1994]) ont signalé un coût important au niveau de la communication entre objets distants, pénalisant les simulations. Les performances sont néanmoins en constante amélioration ;

Nous avons en revanche décidé d'adopter un modèle d'exécution le plus proche possible de celui de l'OMG. Un portage ultérieur sera ainsi plus facile à réaliser.

Java Java est un nouveau langage objet, développé par Sun, qui a fait l'objet en deux ans d'un investissement très important. Java reprend les principes de base de C++, en simplifiant le modèle de programmation (pas de pointeur, gestion mémoire automatique, etc.). Les programmes Java sont portables et permettent la migration dynamique de code, car interprétés par une machine virtuelle. Parmi les atouts clés de Java, nous avons l'intégration des processus légers au niveau du langage et l'intégration de la communication. Bien que de nombreuses caractéristiques de Java (gestion automatique de la mémoire, programme indépendant des plateformes, processus légers intégrés au langage) le rendent particulièrement intéressant pour une implémentation de Prosit, il n'a pas été retenu. Ce choix s'explique par plusieurs raisons. A cette époque, les portages de la machine virtuelle étaient peu fréquents, bogués et souvent instables. De même, les exécutions étaient particulièrement lentes, et l'adoption de Java, demandait un nouveau codage du framework séquentiel développé en C++.

Mais Java a connu une diffusion très rapide et, est maintenant un langage incontournable lors de l'intégration de l'informatique et des réseaux. Les machines virtuelles sont maintenant disponibles pour quasiment toutes les architectures, et diverses techniques permettent d'améliorer les performances (compilation au vol, optimisation par analyse de flot, processeur dédié, etc.). De plus, de nombreuses composantes du *Java Development Kit* sont adaptées à nos besoins : *Remote Method Invocation*, *Object Serialization*, *Reflection*, *JAVA IDL*. Un codage du modèle d'exécution Prosit en Java est donc envisagé.

C++// C++// [Caromel *et al.*, 1996] est une extension de C++ développée au sein du projet Sloop. Les principes à la base de C++// sont une adaptation et une extension du travail de Caromel sur Eiffel// [Caromel, 1991]. Le développement de C++// s'inscrit dans le cadre du projet européen Europa [Caromel et *al.*, 1997] qui a pour objectif de développer et de standardiser un environnement de programmation parallèle en C++. Parmi les solutions adoptées, citons : l'utilisation d'un méta protocole, le polymorphisme entre objets et processus, l'attente par nécessité systématique et la continuation automatique.

C++// devait à l'origine constituer le support d'exécution de Prosit, sa non-disponibilité sous un forme stable au début du développement du framework distribué, nous a amené à l'abandonner. Néanmoins, nous avons collaboré [Baude *et al.*, 1996] avec l'équipe de développement et certains points de Prosit sont proches de l'architecture de C++//. Une fois cet environnement stabilisé, un portage pourra être envisagé.

Schooner Schooner [Furmento et Baude, 1996] est une librairie de communication pour le développement d'applications parallèles en C++. Schooner est bâti au dessus de PVM et est développé au sein du projet Sloop. Certaines composantes de Schooner sont similaires aux spécifications de Prosit (ex : computer, cluster), de plus de nombreuses caractéristiques sont adaptées à notre modèle d'exécution (ex : messages actifs, objets communicants, etc.). Ces deux facteurs ainsi que la proximité de l'équipe de développement de Schooner nous ont amenés à adopter cette librairie afin d'implémenter les communications dans le premier prototype Prosit.

12.1.2 Architecture retenue

Pour la mise en œuvre du framework distribué, nous avons retenu l'approche *proxy* de CORBA, et Schooner comme support de communication. Tout accès à un objet actif (local ou distant) se fera à travers un objet proxy (cf. Fig. 12.5). A chaque classe d'objets actifs sera associée une classe proxy fournissant l'ensemble des primitives d'accès. Le mécanisme séquentiel d'interaction a été étendu de manière transparente, le proxy se chargeant de recevoir et de transmettre les objets requête et réponse.

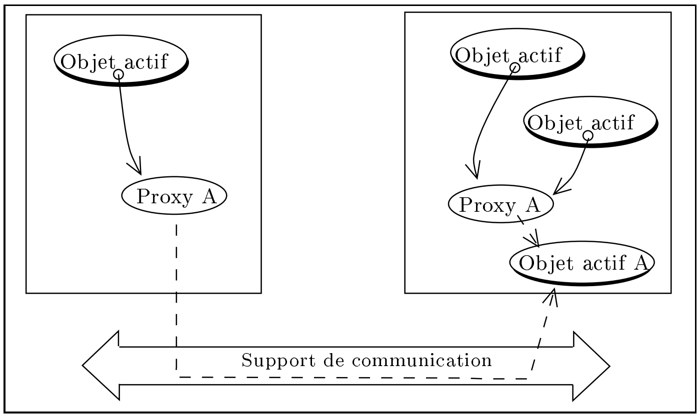


Fig. 12.5 – *L'architecture proxy*

Dans la version actuelle, le programmeur doit coder manuellement ses classes proxys. Cette étape est relativement contraignante, mais reste simple et standardisée grâce à l'héritage de la classe `Proxy` du framework. Nous envisageons à terme, de faire appel à un compilateur IDL. Le programmeur pourrait alors spécifier, en IDL, l'interface de sa classe [1], c'est-à-dire les méthodes (i.e. services) qui pourront être appliquées aux instances. La *compilation* de cette description permettrait de générer automatiquement la classe *proxy*, la classe *squelette* ainsi que le code de *marshalling* des données. La description IDL supportant l'héritage, le programmeur pourra indiquer que son interface hérite d'une interface existante (en général d'une interface fournie avec les librairies). Nous illustrons à l'exemple de code 12.1 cette approche.

Exemple de code 12.1 Code IDL pour la classe Caisse

```
interface Caisse : Server_Normal_FIFO {
  void payer(in int nb_articles);
}
```

12.2 Le modèle de communication SCHOONER

Nous décrivons dans cette section les principales fonctionnalités de SCHOONER. Cette présentation n'est pas exhaustive et le lecteur intéressé pourra se reporter à [Furmento et Baude, 1996] pour plus de renseignements.

SCHOONER est une librairie de communication, écrite en C++, et développée au sein de l'équipe Sloop. Elle offre une vision structurée et abstraite du parc de machines, et permet, de manière relativement simple, l'extension et la redéfinition de ses comportements de base. Cette librairie permet aussi de développer les applications, indépendamment de la bibliothèque d'échanges de messages utilisée. La librairie SCHOONER est organisée en trois niveaux (cf. Fig. 12.6) :

1. le premier niveau constitue l'interface avec la bibliothèque d'échanges de messages utilisée. Actuellement, seul PVM est interfacé ;

2. le second niveau fournit les mécanismes de base permettant de manipuler les machines et les processus lourds. Il fournit aussi le mécanisme de communication entre processus. L'architecture d'exécution est organisée en computer et cluster. Ces composantes ont des caractéristiques similaires à celles du modèle PROSIT (cf. Chapitre 7). La communication entre clusters est réalisée via un mécanisme de messages actifs [Eicken *et al.*, 1992].

3. le troisième niveau fournit un mécanisme de communication entre objets C++. Une application est alors structurée en objets communicants par échanges asynchrones de messages. Un objet communicant doit hériter de la classe `Comm_Object` et les messages de la classe `Data_Comm_Object`. L'accès à un objet communicant se fait via un objet représentant, qui est instance de la classe `Proxy_Comm_Object` (cf. Fig. 12.7).

1. Afin de gérer le transfert des paramètres et du résultat, une description IDL des types construits (classes, record, etc.) devra aussi être fournie.

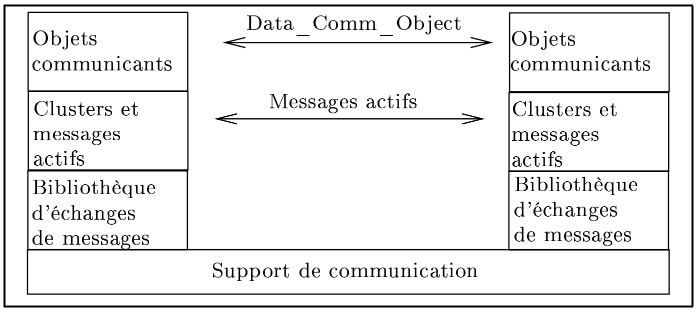


Fig. 12.6 – *L'architecture en couches de* Schooner

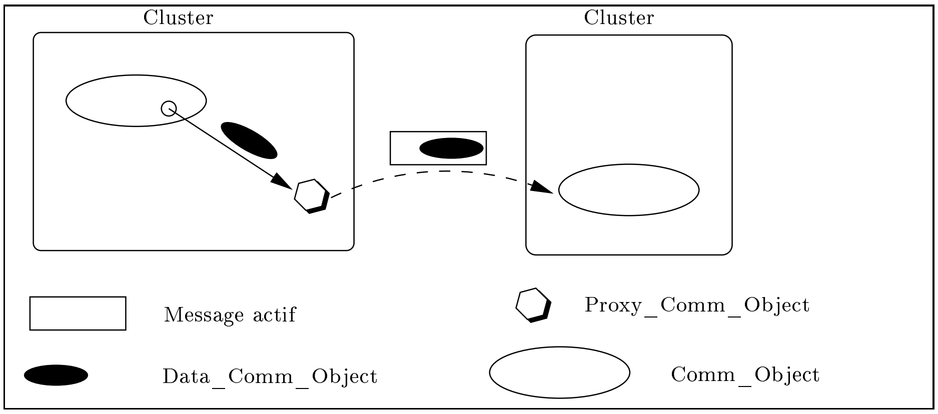


Fig. 12.7 – *Mécanisme de communication de* Schooner

12.3 L'architecture du framework au dessus de Schooner

12.3.1 Mise en œuvre des proxys au dessus de Schooner

Tout objet actif, dans la version distribuée, hérite à la fois de la classe `Simulation_Obj` et de la classe `Comm_Object` (Fig. 12.8), récupérant ainsi l'ensemble des fonctionnalités des deux classes. Ce sont donc des objets actifs pouvant émettre et recevoir des messages à travers le réseau. Les proxys Prosit étant aussi amenés à communiquer avec d'autres objets, héritent de la classe `Comm_Object`. Les interactions entre objets actifs se font en spécialisant la classe `Data_Comm_Object`.

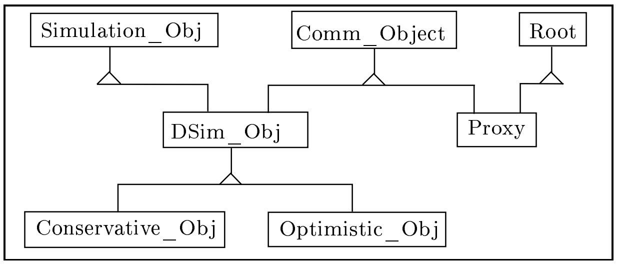


Fig. 12.8 – *Hiérarchie pour les objets actifs et les proxys*

La classe `Proxy` fournit l'ensemble des primitives disponibles sur la classe `Simulation_Obj`.

Elle définit aussi les primitives de gestion de la localisation de l'objet (`is_local()`, `is_remote()`, `location()`, etc.) et les primitives systèmes (réception des messages SCHOONER, gestion du binding, etc.). Sur un cluster, le proxy d'un objet actif n'existe qu'en un seul exemplaire.

Exemple de code 12.2 La classe Proxy (partielle)

```
class Proxy : public Root, public Comm_Object {
private:
   Proxy_Comm_Object *comm_data;// pour la communication SCHOONER
   Cluster *obj_location;// localisation de l'objet actif
   DSim_Obj *obj; // pointeur sur objet actif, si local
   Entity_Id obj_id;
public:
   bool is_available();
   bool is_local();
   bool is_remote();
   Cluster *location();
}; // end Proxy
```

12.3.2 Architecture d'un cluster

Sur chaque cluster est alloué un objet noyau chargé de l'ordonnancement des objets actifs locaux et de la synchronisation globale. A chaque famille de synchronisation sera associée une classe noyau spécifique qui devra hériter de la classe `Distributed_Kernel`. Cette classe hérite à la fois de `Kernel` afin de récupérer l'ordonnancement temporel, et de `Comm_Object` afin d'être en mesure de communiquer avec les autres noyaux. Le noyau gère aussi deux autres composantes particulièrement importantes (cf. Fig. 12.9) :

- un gestionnaire d'objets (classe `Prosit_Obj_Manager`) qui répertorie l'ensemble des objets actifs et des proxys présents sur le cluster ;
- un gestionnaire de communication (classe `Prosit_Comm_Manager`) qui gère l'interface avec la librairie de communication. Le gestionnaire implémente aussi l'extension synchrone des communications de SCHOONER (qui sont par définition asynchrones). Grâce à ce gestionnaire, PROSIT est indépendant (à condition que les mêmes fonctionnalités soient disponibles) du support de communication.

12.3.3 Gestion des objets actifs

Création et destruction d'objets actifs

Le programmeur ne crée pas directement les objets actifs, mais instancie un proxy en spécifiant éventuellement le cluster cible. C'est l'objet proxy qui se charge d'instancier l'objet actif sur le cluster destination. Le mécanisme de création des *objets communicants* de SCHOONER est utilisé. L'objet actif et le proxy s'enregistrent auprès du gestionnaire d'objets.
Contrairement au mécanisme de création de SCHOONER, qui est asynchrone, la création d'un

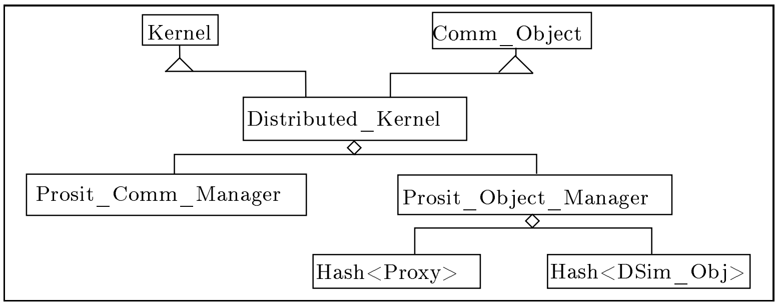


Fig. 12.9 – *Diagramme de classes pour le noyau en environnement distribué*

objet actif est synchrone. Il y a récupération de l'identifiant de l'objet actif, ce qui permet de vérifier que la création et l'initialisation se sont bien passées.

A la destruction de l'objet actif, le gestionnaire d'objets est prévenu, ce qui permet de mettre à jour les structures internes. Un message est diffusé, afin que chaque sous-système mette à jour le proxy de l'objet détruit. Les proxy ne sont pas directement détruits, un drapeau est mis à jour, et une exception est levée à toute tentative d'accès (seule la méthode `is_available()` permettant de tester la disponibilité de l'objet ne lève pas d'exception).

Mécanisme de binding et gestion des objets

Le mécanisme de binding permet d'obtenir un pointeur sur le proxy local d'un objet actif en indiquant son identifiant ou son nom. Pour cela chaque classe proxy fournit une méthode statique `bind()`. Nous donnons en 12.3 un exemple de primitive `bind()`. Lors de la mise en œuvre de ce mécanisme, trois cas sont distingués : (1) le proxy existe déjà, il est donc enregistré dans la table des proxy gérée par le gestionnaire d'objets, il suffit de retourner un pointeur dessus ; (2) le proxy n'existe pas mais l'objet actif est local, il est donc enregistré auprès du gestionnaire d'objets, un proxy est donc créé et initialisé ; (3) le proxy n'existe pas et l'objet actif n'est pas local, le noyau diffuse à l'ensemble des clusters un message de demande de localisation. Le noyau situé sur le cluster hébergeant l'objet actif répondra. Une fois la localisation résolue, un proxy est créé et initialisé.

Exemple de code 12.3 Codage d'une primitive de binding

```
PCM_Porte* PCM_Porte::bind(Entity_Id &id){
  // test si proxy existant
  if(Proxy::proxy_registered(id))
    return(dynamic_cast<PCM_Porte*>Proxy::return_proxy(id));
  // test si OA est local
  if(Proxy::object_registered(id)) return(new PCM_Porte(Proxy::return_obj(id)));
  // OA distant et pas de proxy local, resolution dynamique
  Location_Info location=Proxy::find_location(id);
  return(new PCM_Porte(id,location));
}
```

12.3.4 Détails du mécanisme d'interaction

Nous présentons ici le fonctionnement du mécanisme de réification dans le cadre distribué en passant en revue les quatres points de réification.

Emission de la requête

Une fois l'appel réifié en objet requête, la fonction `send_request()` du client est appelée avec la requête en paramètre. Cette fonction complète l'initialisation de la requête et la transmet au proxy du serveur via la primitive `get_request()`. Nous avons mis en place une nouvelle hiérarchie de classes (cf. Fig. 12.10) requête et réponse contenant les informations spécifiques à la distribution (cluster source, identifiant SCHOONER du demandeur, etc.).

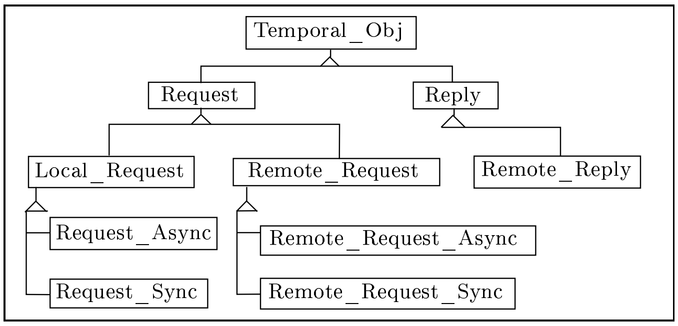


FIG. 12.10 – *Hiérarchie des classes requêtes et réponse*

Le proxy implémente la méthode de calcul dynamique de l'adresse. Si l'objet est local (cf. Fig. 12.11.a), le proxy utilise la méthode de résolution de ce dernier. L'objet MID contient alors l'adresse de la fonction membre. Par contre, si l'objet est distant (cf. Fig. 12.11.b), l'objet MID contient un identifiant unique.

A la réception de la requête par le proxy du serveur, ce dernier distingue deux cas (voir l'exemple de code 12.4). Si l'objet est local, il y a transfert direct de la requête. Par contre pour un objet distant, grâce à l'objet MID, le proxy crée et initialise le message SCHOONER spécifique à cette requête puis l'émet de manière asynchrone. A la réception du message SCHOONER, l'objet requête est reconstruit et livré au serveur.

Exemple de code 12.4 Exemple de stub de proxy

```
void Proxy::get_request(Request *rq){
  // si objet local
  if(is_local()) {
    rq->set_destinataire(obj);
    obj->get_request(rq); }
  else
    Reified_DCO *dco = create_dco(rq->get_mid());
    dco->init(rq);
    dnoyau->asend(comm_data,dco,cid);
}
```

Réception de la requête

La réception et le traitement de la requête ne diffèrent pas du mode séquentiel.

Emission de la réponse

A la fin du service d'une demande bloquante, le serveur émet une réponse via la primitive `send_reply_for()`. Cette dernière est surchargée au niveau de la classe `DSim_Obj` et effectue les actions suivantes :

- si le demandeur est local, il y a construction d'un objet réponse et transfert. On se retrouve dans le même mode de fonctionnement qu'en séquentiel.
- si le demandeur est distant, grâce au MID, le proxy crée et initialise la réponse, puis le message SCHOONER spécifique à la requête, et enfin l'émet de manière asynchrone.

A la réception du message SCHOONER, l'objet réponse est reconstruit et livré au client.

Réception de la réponse

La réception et le traitement de la réponse ne diffèrent pas du mode séquentiel.

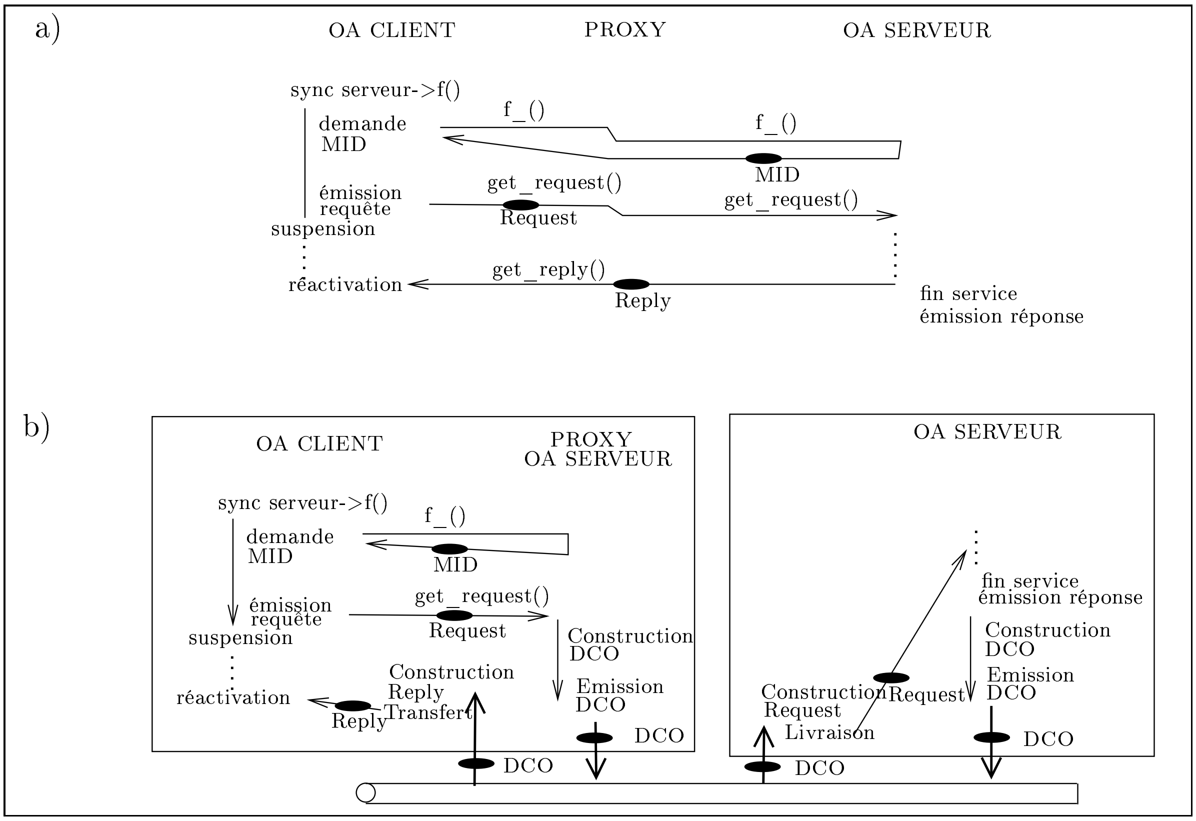


FIG. 12.11 – *Mécanisme de réification – a) en séquentiel, b) en distribué –*

12.4 Mise en œuvre de la migration des objets actifs

La migration d'un objet actif comprend la migration des données et la migration du processus léger associé à l'activité principale. Les objets migrants doivent hériter de la classe `DSim_Obj`. Cette classe fournit les primitives de base implémentant la migration. Un objet actif demande à migrer en utilisant la primitive `migrate()` avec comme paramètre le cluster cible. L'objet sera automatiquement réactivé sur le cluster destination. Les données internes au framework sont automatiquement mises à jour et le programmeur transfère les attributs de l'objet ainsi que les données allouées dynamiquement par l'activité en surchargeant les méthodes `pack_user_data()` et `unpack_user_data()`. Ces deux méthodes prennent en paramètre un objet `Buffer` qui est le tampon d'émission/réception de SCHOONER. Après avoir migré, l'objet actif se trouve dans un autre environnement global, le transfert des attributs et des données dynamiques, par le programmeur, permet à l'objet de poursuivre son exécution de manière transparente.

Après avoir migré, l'objet actif doit être en mesure de recevoir les demandes de services des objets clients. Grâce à l'approche *proxy*, il est inutile de mettre à jour les références des clients (le proxy de l'objet migrant est toujours présent), il faut par contre mettre à jour les objets proxys associés à l'objet migrant. Cette mise à jour consiste uniquement à réactualiser la localisation de l'objet. Pour cela plusieurs approches sont possibles :

- lors de la migration, le noyau diffuse la nouvelle localisation de l'objet, chaque gestionnaire d'objets met à jour, s'il existe, le proxy local ;
- la requête est émise vers le cluster où se trouvait l'objet serveur, ce dernier ayant migré, la requête revient avec un code d'erreur. Le proxy effectue alors une recherche (similaire à celle exécutée lors de l'opération de binding) et ré-émet la requête vers le bon cluster ;
- une autre technique, qui est une variante de la précédente, est celle des *liens de poursuite* [INRIA, 1990]. Lors de la migration, le gestionnaire d'objets du site de départ mémorise la nouvelle localisation de l'objet, cette information est utilisée pour les recherches ultérieures.

Nous n'avons implémenté aucun de ces mécanismes dans la version actuelle de PROSIT, ni encore étudié leurs influences sur la synchronisation. Seuls les objets clients peuvent pour l'instant migrer.

La migration d'objet actif est une fonctionnalité très intéressante, mais sa mise en œuvre au sein du langage C++ possède des contraintes et des limitations incontournables. Ces inconvénients proviennent du fait que le code de simulation ne bénéficie pas de mécanismes spécifiques au niveau du langage, ni d'un pré-traitement effectué à la compilation. Pour pallier à ces problèmes, nous avons introduit deux formes de migration. La première, consiste à migrer l'activité principale de l'objet et à la poursuivre sur le cluster destination. C'est cette première approche qui est la plus complexe à mettre en œuvre. La seconde technique consiste à ne pas migrer l'activité mais à la re-créer sur le cluster cible. Le programmeur doit alors se charger du point de reprise dans le code et des données locales à l'activité. Nous présentons chacune de ces approches dans la suite.

12.4.1 Migration avec transfert de l'activité

La migration d'un objet actif avec transfert de l'activité principale (le mécanisme est le même pour les activités annexes), consiste à transférer les données associées à l'objet, celles associées à l'activité et à transférer le processus léger. Si le transfert des données ne pose pas de problème, celui du processus léger est une opération complexe comprenant :

- la copie des structures internes du processus léger (pointeur de pile, etc.) et la copie de la partie utilisée de la pile dans un tampon,
- le transfert du tampon,
- à l'arrivée, l'allocation d'un espace mémoire et la recopie de la pile. Mais cette dernière se trouve maintenant à une adresse différente de celle de départ et doit donc être mise à jour.

C'est la mise à jour de la pile qui constitue l'opération la plus complexe. Elle comprend la mise à jour du chaînage des blocs d'appel[2] et la mise à jour des données (du framework[3] et de l'utilisateur) stockées en pile sous la forme de pointeur. La mise à jour de ces informations est dépendante de l'architecture (qui détermine la taille des pointeurs et le sens de progression de la pile) et du compilateur utilisé (qui détermine la structure des blocs d'appel). Il faut aussi que le cluster cible et le cluster destination soient identiques (même architecture et même génération de processeur).

Nous avons réalisé une implémentation du mécanisme de migration pour le compilateur g++ et pour les quatres architectures actuellement supportées par PROSIT. La sauvegarde et la restauration du processus léger sont réalisées par les deux méthodes `pack_activity()` et `unpack_activity()` de la classe `DSim_Obj`. Si nous avons montré la faisabilité de ce mécanisme et effectué de nombreuses expériences (cf. §14.3.2 et §14.5), nous devons néanmoins compléter sa validation et sa mise en œuvre.

12.4.2 Migration avec création d'une nouvelle activité

La migration avec création d'une nouvelle activité consiste à ne migrer que la partie données de l'objet actif. Une fois migrées, les données sont mises à jour, une nouvelle activité est créée et l'objet est réactivé. A la réactivation, le comportement de l'objet ré-exécute la fonction `behave()` sans mémoire du point d'arrêt. C'est donc au programmeur d'intégrer dans le code du comportement la notion de point de reprise et de stocker, dans les attributs, l'information nécessaire à cette reprise. L'intégration du mécanisme de reprise peut soit faire appel à des techniques de pré-traitements, comme c'est le cas en REACTIVE-C [Boussinot et Doumec, 1992, Boussinot, 1992] et en SimPOL [Harold et Malloy, 1993], soit être entièrement laissée à la charge du programmeur. Dans la version actuelle, nous avons retenue la seconde approche car elle ne nécessitait aucun traitement spécifique. Nous avons introduit la classe `Data_Migrator` (descendant de `DSim_Obj`) qui surcharge les méthodes `pack_activity()` et

2. Un bloc d'appel représente le contexte d'exécution d'une fonction (adresse de retour, paramètres et variables locales). Ces blocs sont généralement chaînés entre eux.
3. Ces données sont entre autres : un pointeur sur l'objet actif, un pointeur sur l'activité, etc.

`unpack_activity()` afin de ne plus migrer le processus léger. Nous donnons à l'exemple de code 12.5 une fonction `behave()` avec prise en compte du point de reprise.

Exemple de code 12.5 Méthode behave() avec prise en compte du point de reprise

```
class Client : public Data_Migrator {
private:
   int etat;
   void pack_user_data(Buffer *buff) {*buff << etat;};
   void unpack_user_data(Buffer *buff){ *buff >> etat; };
public:
   void behave() {
      if(etat==1) goto debut_ss1;
      if(etat==2) goto debut_ss2;
      if(etat==3) goto debut_ss3;
      if(etat==4) goto debut_ss4;
      debut_ss1:
         ...
         etat=2;
         migrate(...);
      debut_ss2:
      ...
   }
}; // end Client
```

Les principaux avantages de cette approche sont la portabilité et la possibilité de migrer les objets actifs entre clusters hétérogènes. Les performances devraient aussi être meilleures, dans la mesure où il y a moins de données transférées et où les mécanismes intervenant sont moins complexes. Mais ces avantages s'obtiennent au détriment d'une programmation simple et naturelle.

12.4.3 Exemple de systèmes supportant la migration

La migration peut porter sur le transfert d'objets, de processus légers ou encore d'objets actifs. Nous ne traitons pas ici de la migration de processus Unix, le lecteur intéressé peut consulter la référence [Nuttall, 1994]. La migration d'objets a été étudiée dans le système SOS du projet SOR de l'INRIA [Shapiro *et al.*, 1989b, Shapiro *et al.*, 1989a], elle est aussi disponible dans la couche Chorus/COOL [COO, 1991] développée au dessus du micro-noyau Chorus [Chorus, 1989]. La migration de processus légers a fait l'objet de nombreuses implémentations qui diffèrent selon le niveau d'intégration. Dans le cas de Mach [Ford et Lepreau,] et de Chorus, la migration est directement intégrée au niveau du système d'exploitation ce qui facilite grandement son utilisation. Mais l'approche la plus répandue consiste à prendre en compte la migration au niveau de la librairie de processus légers, comme c'est le cas avec les librairies Marcel [Namist, 1996] et Ariadne [Mascarenhas et Rego, 1995]. La dernière forme de migration, qui est une synthèse des précédentes, est la migration d'objet actif. Le langage Emerald [Black *et al.*, 1988] et l'environnement PROSIT sont des exemples d'environnements supportant la migration d'objet actif.

12.5 La synchronisation conservative

La synchronisation conservative est le premier protocole implémenté dans Prosit. Trois versions sont disponibles, chacune associée à une classe noyau (cf. Fig. 12.12) :

- `CM_Kernel`, implémente le protocole conservatif, mais sans messages nuls ;
- `CM_Auto_Kernel`, implémente le protocole conservatif avec émission automatique de messages nuls ;
- `CM_DD_Kernel`, similaire au noyau précédent, mais les messages nuls sont envoyés uniquement à la demande.

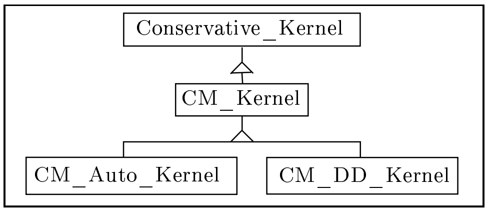


Fig. 12.12 – *Classes noyaux pour la synchronisation conservative*

12.5.1 Principe de fonctionnement

Chaque noyau gère une liste de canaux pour les clusters successeurs et une autre pour les prédécesseurs. Un objet n'est réactivé que si :

$$\textit{date(objet)} \leq \textit{min(date(canal predécesseur encore ouvert))}.$$

Pour chaque canal d'entrée, le noyau maintient les informations suivantes : la date du dernier message provenant de ce cluster (requête, réponse, migration, message nul), l'état du canal : ouvert (un message peut encore arriver) ou fermé (plus aucun message n'arrivera), la liste des messages provenant de ce cluster (requête, réponse), non encore livrés par le noyau aux destinataires (i.e ce sont les messages dans le futur du destinataire).

Pour chaque canal de sortie, le noyau maintient les informations suivantes : la date du dernier message émis vers ce cluster (requête, réponse, migration, message nul), l'état du canal.

Le programmeur doit explicitement indiquer, lors de la phase prologue (cf. l'exemple de code 12.6), la topologie entre clusters (primitive `set_succ()` du noyau). Un canal doit être créé du cluster *c1* vers le cluster *c2*, si au cours de la simulation une requête ou une réponse est émise par un objet actif situé sur *c1* vers un objet actif situé sur *c2*, ou encore si un objet actif situé sur *c1* migre vers *c2*.

Le noyau gère une liste des objets actifs participant au calcul de lookahead. C'est lui qui gère l'émission des messages nuls et les demandes de messages nuls.

Exemple de code 12.6 Création de clusters et spécification de la topologie

```
Cluster *cl1 = new("host1") Cluster;
Cluster *cl2 = new("host2") Cluster;
...
cm_noyau = new CM_Auto_Kernel;
...
cm_noyau->set_succ(Cluster::current(),cl1);
cm_noyau->set_succ(cl1,Cluster::current());
cm_noyau->set_succ(Cluster::current(),cl2);
```

12.5.2 Objets actifs

Les objets actifs intervenant dans une simulation conservative doivent être d'une classe descendante de la classe `Conservative_Obj`. Cette classe fournit les primitives suivantes :

- `get_lookahead(Time t)`, qui permet d'obtenir le lookahead de l'objet actif à la date t. Par défaut cette primitive retourne 0. Cette primitive est appelée par le noyau.
- `external_interactions()`, qui permet de savoir si cet objet doit participer au calcul du lookahead. Par défaut cette primitive retourne `true`.

Ces primitives doivent être spécialisées par le programmeur et sont détaillées dans la section suivante.

12.5.3 Calcul du lookahead

Le lookahead d'un objet actif est une prédiction effectuée par ce dernier sur ses interactions avec les autres clusters. Si α est le lookahead à la date t, nous savons que cet objet n'effectuera aucune des actions suivantes avant la date $t + \alpha$: migration vers un autre cluster, émission d'une requête vers un objet situé vers un autre cluster, émission d'une réponse vers un objet situé vers un autre cluster. Le lookahead d'un cluster est défini comme étant le minimum du lookahead des objets actifs participant au calcul. Le calcul rapide d'un bon lookahead (i.e. grand) est une opération très importante pour l'efficacité du protocole. Divers points faibles peuvent être notés :

- Le nombre d'objets à consulter peut être important. C'est pour cette raison que nous avons introduit la primitive `external_interactions()`.
- L'opération *min* peut être source de mauvaises performances si le placement des objets est mal étudié.
- Le protocole conservatif fonctionne si au moins un des clusters du cycle possède un lookahead non nul. Pour cela l'ensemble des objets actifs participant au calcul du lookahead doivent avoir un lookahead non nul. Nous augmentons donc la granularité au détriment de la qualité du lookahead.

12.6 Description de l'implémentation

12.6.1 Organisation du code

La version distribuée du framework comprend une seule librairie (39 classes, environ 5000 lignes de code). La prochaine version de PROSIT comprendra une librairie pour les classes de base et une librairie par protocole de synchronisation.

12.6.2 Matériels et logiciels utilisés

Comme pour l'environnement séquentiel (cf. §11.4.2), la version distribuée est portée sur l'ensemble des plates-formes et des systèmes d'exploitation suivants : architecture INTEL et système LINUX, alphastation et système Digital UNIX, architecture SUN et systèmes SunOS 4 et SunOS 5 (Solaris). La programmation est réalisée en C++ et le compilateur utilisé est celui fournit par *GNU* (g++). La version 1 de SCHOONER est utilisée et la version de PVM est la 3.3.11.

13. BIBLIOTHÈQUES ET OUTILS DÉVELOPPÉS

Nous avons décrit aux chapitres 6 et 7 l'architecture Prosit : un noyau de simulation sous forme de framework autour duquel s'articulent des bibliothèques de modélisation ainsi que des outils de suivi et d'analyse. Nous présentons dans ce chapitre les librairies et outils développés. Ces réalisations avaient un double objectif : enrichir l'environnement Prosit afin de le rendre utilisable au sein de notre laboratoire, et aussi valider et évaluer nos diverses propositions.

Les développements logiciels présentés proviennent d'une collaboration avec de nombreuses personnes : D. Finzi et B. Gaujal pour la libraire de simulation de réseaux de Petri [Finzi, 1996] ; F. Gioanni pour le mécanisme de traces [Gioanni, 1996] ; M. Badel, A. Jean-Marie et T. de Pretto pour l'outil statistique [de Pretto, 1995, Badel *et al.*, 1996]. L'interface graphique pour la construction de réseaux de files d'attente est inspirée et a partiellement réutilisé l'interface développée pour le simulateur Parseval [Rakotoarisoa, 1994]. L'interface graphique pour la construction de réseaux de Petri est celle de l'environnement ERS développé par A. Jean-Marie [Jean-Marie, 1997].

13.1 Les bibliothèques de modélisation

Les classes de modélisation étendent les classes de base (`Simulation_Obj`, etc.) en définissant un paradigme de modélisation : files d'attente, réseaux de Petri, etc. A chaque composante du paradigme est associée une classe paramétrable et réutilisable. L'ensemble de ces classes est alors regroupé dans une librairie. L'environnement Prosit dispose actuellement d'une librairie pour la simulation de files d'attente et d'une librairie pour la simulation de réseaux de Petri. Nous présentons dans la suite la librairie de simulation de files d'attente, en montrant comment les mécanismes du framework ont été utilisés.

13.1.1 Simulation de réseaux de files d'attente

Nous avons développé une librairie de modélisation pour la simulation de réseaux de files d'attente. Conformément à l'analyse du formalisme présentée à la section 9.1.1, nous avons mis en place des classes pour les architectures clients actifs et clients passifs. Lors de la mise en œuvre de la librairie, nous n'avons pas dupliqué l'ensemble des classes stations pour chacune des architectures. Par exemple, la station serveur du modèle client passif peut hériter de la station serveur du modèle client actif et rajouter une fonction service ayant comme paramètre le client et le routant en fin de service. Suivant ce principe, nous avons implémenté une librairie supportant les deux architectures. Dans la suite, nous présentons brièvement les principaux mécanismes de l'approche client actif (cf. Fig. 13.1). Les performances de cette librairie sont évaluées au chapitre 14.

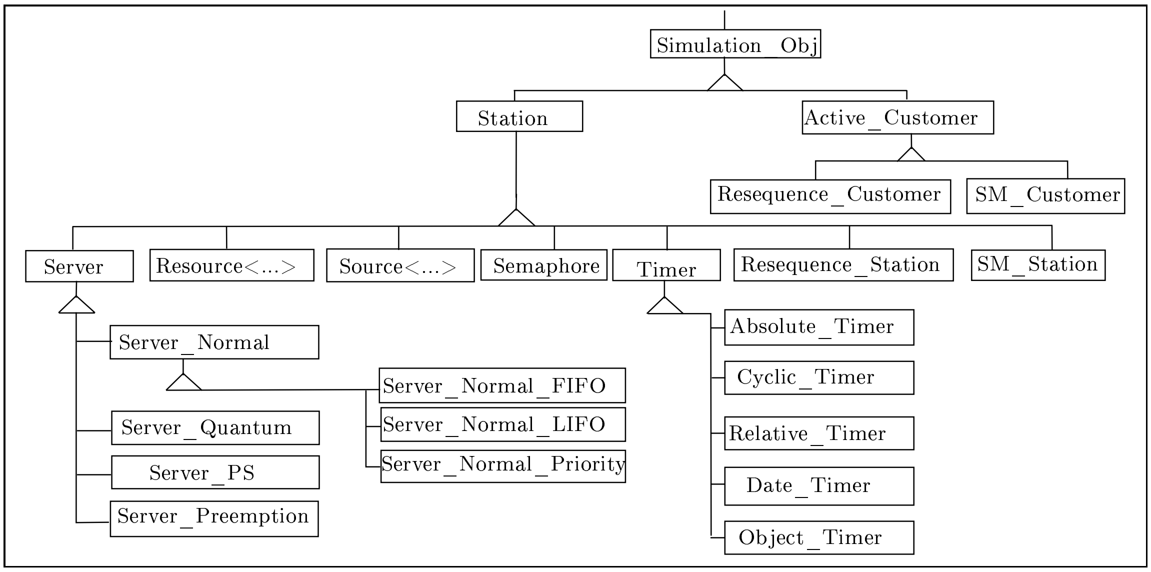


FIG. 13.1 – *Hiérarchie des classes pour la librairie files d'attente*

Les clients

Les clients sont spécifiques au modèle simulé. C'est donc au modélisateur de coder ses classes de clients, en héritant de la classe `Active_Customer`, et en rajoutant les attributs nécessaires et les demandes de service dans la fonction `behave()`.

Les sources sont chargées de générer les clients d'une certaine classe suivant une distribution (temps inter-génération). L'activité de l'objet source consiste, de manière cyclique, à créer et à initialiser un client et à consommer du temps. La classe source est générique sur la classe des clients à créer. Cette classe s'utilise par héritage, car le modélisateur doit spécifier l'initialisation des clients.

Les stations serveurs

Une station serveur de base est un objet actif possédant une file d'attente de requêtes et fournissant des services. Un serveur est caractérisé par :

- le nombre de serveurs. C'est le nombre de requêtes pouvant être simultanément en cours de service. Une requête pouvant être soit directement servie par l'activité principale, soit servie par une activité annexe. Dans le second cas, le nombre de serveurs correspond au nombre d'activités annexes pouvant simultanément s'exécuter ;
- la capacité et la discipline d'attente. C'est le mode de gestion des requêtes en attente de service. Le rejet d'un client correspond au rejet de la requête de demande de service ;
- la politique d'allocation du serveur. C'est la politique de gestion des activités annexes.

Ces caractéristiques s'implémentent par différentes classes de comportement. Là encore, le modélisateur hérite de la classe fournissant le comportement désiré et code ses fonctions services (voir l'exemple de code 13.1).

Exemple de code 13.1 Sélection du comportement par héritage

```
class Porte : public Server_Normal_FIFO {
private:
  Random *loi_service;
public:
  void passer() {
    wait((*loi_service)());
  }
  ...
}; // end Porte
```

Les mécanismes de synchronisation

Les principaux mécanismes de synchronisation sont réalisés par des objets actifs spécifiques tels que le gestionnaire de ressource, le sémaphore et le drapeau. La synchronisation est toujours réalisée via une demande de service. Ce service pouvant être instantané ou par exemple durer le temps qu'une ressource se libère.

Deux autres mécanismes de synchronisation, le reséquencement et la fission/fusion, requièrent une attention particulière dans le modèle client actif. Ces mécanismes ont demandé la mise en place de composantes spécifiques. Nous présentons leurs mises en œuvre, car elles illustrent bien la capacité d'adaptation et de spécialisation du framework.

Mécanisme de reséquencement Le reséquencement consiste à ré-ordonner l'exécution d'un ensemble de clients relativement à un enregistrement temporel préalablement réalisé. Suite au reséquencement les clients poursuivent leur exécution dans l'ordre exact de l'enregistrement. Ce mécanisme est explicitement réalisé par les clients, et comprend deux étapes : (1) demande d'enregistrement temporel auprès d'une station de reséquencement, (2) demande de reséquencement auprès de la station ayant réalisé l'enregistrement. Une demande d'enregistrement est instantanée. Un client demandant un reséquencement sera suspendu tant que tous ses prédécesseurs n'ont pas été reséquencés. Afin de faciliter la programmation, nous avons introduit une classe de clients (`Resequence_Customer`) possédant des primitives de reséquencement (`register()`, `re_sequence()`). Le reséquencement n'est alors plus vu comme une demande de service, mais comme un appel de fonction membre, ce qui permet de masquer le paramétrage de la demande de service.

Eclatement et regroupement de clients L'éclatement d'un client est l'opération consistant à détruire un client et à en générer d'autres à sa place. Le nombre et la (les) classe(s) des clients générés sont spécifiés par le programmeur. L'éclatement peut se faire avec ou sans mémorisation de l'origine et avec ou sans consommation de temps. Le regroupement est l'opération inverse qui consiste à fusionner plusieurs clients en un ou plusieurs autres clients.

Le mécanisme d'éclatement/fusion nécessite la participation du noyau de simulation dans la mesure où les clients demandent leur suspension et veulent être gérés par une station spécifique. Pour cela, nous avons introduit deux nouveaux états pour les objets actifs : `split` et `merge`. Un client désirant fusionner ou fissionner utilise respectivement la primitive `merge()` ou `split()`

en indiquant la station gestionnaire. Le client est alors suspendu et le noyau le transmet, via une demande de service non bloquante, à la station spécifiée. Le client est transmis sous forme de paramètre du service, et est considéré comme un objet passif n'étant plus géré par le noyau. Le service du client, par la station, est spécifié par le programmeur et consiste à réaliser la fission/fusion. Pour implémenter ces mécanismes, nous avons introduit une nouvelle classe noyau (`Kernel_Queueing`) ainsi qu'une classe pour les clients (`SM_Customer`) et les stations (`SM_Station`).

La mise en œuvre réalisée est une bonne illustration du potentiel d'adaptation et de spécialisation du framework PROSIT.

Les temporisateurs

Les temporisateurs permettent d'effectuer, au sein de la simulation, un traitement particulier à un instant donné. La librairie fournit différentes classes de comportement pour les différentes catégories de temporisateurs (cf. tableau 13.1). Le modélisateur n'ayant plus qu'à spécifier le traitement à effectuer.

A date fixe	Date absolue	*Absolute_ Timer*
	Déplacement	*Relative_ Timer*
Cyclique	Toutes les t unités de temps.	*Cyclic_ Timer*
	Après chaque réactivation d'objet actif.	*Object_ Timer*
	A chaque changement de date.	*Date_ Timer*

TAB. 13.1 – *Les catégories de temporisateurs*

Les statistiques

Les classes de la librairie fournissent diverses statistiques en faisant appel à l'outil STAT-TOOL (cf. section 13.2.2). Toute entité possède les deux primitives suivantes : `compute_stat()` qui indique à l'entité qu'elle doit calculer des statistiques et `report_stat()` qui demande à l'entité d'imprimer les résultats statistiques. Chaque résultat peut aussi être accédé individuellement (cf. tableau 13.2).

`get_mean_service()`	temps de service moyen
`get_mean_requests()`	nombre moyen de requêtes
`get_mean_response()`	temps de réponse moyen
`get_nb_service()`	nombre de services

TAB. 13.2 – *Accès aux statistiques d'une station serveur*

13.2 Les outils d'analyse et de suivi

13.2.1 Mécanisme de traces

La génération de traces permet le suivi, à tout moment, de l'évolution du modèle simulé et du simulateur lui-même. Grâce à l'analyse post-mortem, il est par exemple possible de valider le noyau, de déboger un modèle ou une librairie, de calculer des statistiques, etc. Un module de trace générique, appelé TRACE-TOOL, a donc été développé et intégré à PROSIT.

TRACE-TOOL comprend des librairies définissant le contenu et le format des traces, et un module gestionnaire chargé de l'instanciation des objets traces et de leur transfert sur disque. L'intégration de TRACE-TOOL est faiblement intrusif et consiste à spécialiser le gestionnaire et à développer une librairie de traces spécifique à l'application. A l'utilisation, le programmeur configure le module en indiquant les événements à tracer et les conditions temporelles de génération. Il peut aussi spécifier une fonction par type de traces qui sera automatiquement appelée à chaque génération de la trace. Pour générer une trace, il suffit d'appeler la méthode correspondante du gestionnaire en passant les paramètres nécessaires.

Le noyau de simulation est ainsi totalement abstrait du format [1] et des mécanismes d'écriture. Le module offre un double niveau de sélection avec la possibilité de demander les traces pour l'ensemble des objets actifs ou bien pour un objet actif spécifique. Trois catégories de traces sont distinguées : (1) le niveau système qui regroupe les traces portant sur le noyau de simulation, (2) le niveau modèle qui regroupe les traces portant sur le modèle de simulation (ex : files d'attente, réseaux de Petri, etc.), (3) le niveau utilisateur avec les traces introduites par le modélisateur dans le code de sa simulation. Nous donnons à l'exemple de code 13.2 l'extrait d'un fichier de traces niveau système.

Lors d'une exécution distribuée, chaque cluster génère un fichier de traces sur le disque local. Un programme annexe se charge de regrouper l'ensemble de ces fichiers en corrigeant le déphasage entre les horloges.

13.2.2 Traitements statistiques

Cette section est une adaptation résumée de l'article [Badel *et al.*, 1996].

Les traitements statistiques sont un objectif majeur, voire l'objectif principal, de nombreuses simulations. C'est pourquoi, les environnements de simulation proposent généralement des mécanismes et des outils pour la collecte et la manipulation des échantillons. Malheureusement, le traitement des mesures est généralement intégré au noyau de simulation et rarement extensible par l'utilisateur. Le développement d'un nouveau simulateur nécessite donc soit l'acquisition de produits commerciaux, soit le codage de ses propres outils. Chacune de ces solutions est critiquable. Les produits commerciaux coûteux, doivent être disponibles pour toutes les architectures et posent souvent des problèmes en terme de distribution du produit

1. Les trois formats de traces les plus répandus sont PICL [Worley, 1992], TDL [Mohr, 1993] et SDDF [Aydt, 1992]. Grâce à l'architecture adoptée, il est aisé d'adapter TRACE-TOOL à chacun de ces formats. Nous avons décidé d'implémenter un format propre à PROSIT. Ce format est simple à générer, est lisible et est en adéquation avec notre modèle d'exécution. Nous avons aussi codé une seconde librairie de traces au format de l'outil d'animation (cf. section 13.2.3).

Exemple de code 13.2 Exemple de fichier de traces

```
// ***********************************************************
// Fichier : trace.out
// Date en debut de simulation : Thu Jun 27 10:04:19 1996
// ***********************************************************
ACT_OBJ_REG:<0,0,2>,CaisseCentrale,0.0000,0s54656us;
ACT_OBJ_REG:<0,0,3>,Caisse,0.0000,0s55632us;
ACT_OBJ_REG:<0,0,4>,Entree,0.0000,0s56608us;
ACT_OBJ_REG:<0,0,5>,Rayon,0.0000,0s56608us;
ACT_OBJ_REG:<0,0,6>,Sortie,0.0000,0s57584us;
ACT_OBJ_REG:<0,0,7>,Source,0.0000,0s57584us;
SIM_START:0.0000,0s70272us;
ACT_OBJ_SLP:<0,0,2>,CaisseCentrale,0.0000,0s81984us;
................
TIM_CHG:15800.9158,15802.9158,453s825296us;
ACT_OBJ_FIN:<0,0,2004>,SOURCE1-Jr1997,15802.9158,453s825296us;
SIM_END:15802.9158,453s826272us;
// ***********************************************************
// Date en fin de simulation : Thu Jun 27 10:11:52 1996
// Duree de la simulation : 453s 826272us
// ***********************************************************
```

final. Coder ses propres outils est toujours laborieux en temps de développement et de débogage. De plus, l'absence d'une architecture commune, ne permet pas de réutiliser l'effort de développement entre plusieurs programmeurs, ou entre plusieurs équipes. Face à ce constat, nous avons conçu et développé un outil statistique extensible et réutilisable. Les principales caractéristiques de Stat-Tool sont les suivantes :

- l'ajout d'un nouveau traitement statistique consiste simplement à développer et à enregistrer un module,
- le code réalisant les traitements statistiques est dissocié du code de simulation,
- l'outil peut dynamiquement distribuer les calculs sur des machines hétérogènes. Cette répartition des traitements correspond à la distribution fonctionnelle présentée à la section 3.1. Les performances sont aussi améliorées par des techniques de bufférisation et de placement.

Nous présentons dans la suite l'architecture de Stat-Tool et son intégration à l'environnement Prosit.

L'architecture de Stat-Tool

Stat-Tool peut être vu comme un module autonome réalisant des traitements sur des flux de données provenant du simulateur (cf. Fig. 13.2). Dynamiquement, il est possible de demander de nouveaux traitements, de consulter un résultat ou encore d'arrêter une analyse.

L'interface entre le noyau et l'outil statistique est organisé de manière hiérarchique et fonctionnelle en trois composantes : les entités, les variables et les flux de données. Ces composantes permettent de structurer l'interface en dissociant la partie simulation de la partie traitement. Elles permettent aussi, au module statistique, de distribuer de manière transparente les calculs.

Les entités Une entité est associée à un objet de simulation (par exemple à une file d'attente ou à une transition d'un réseau de Petri) et est composée de variables sur lesquelles sont exécutés les traitements statistiques.

Les variables Une variable représente une valeur quantitative d'une entité que le modélisateur désire étudier (par exemple le nombre de clients ou le temps de service dans une file d'attente). Une variable peut être un type de base (flottant, entier, etc.) ou un type structuré (classe, record, etc.).

Les flux de données Le transfert des échantillons, valeurs prises par les variables, au module statistique se fait à l'aide de flux de données. Un flux est bidirectionnel et relie une variable au module statistique. Les échantillons sont transférés soit sous forme discrète (seule la valeur est transférée) soit sous forme continue (la date d'échantillonnage est aussi transférée). Une fois le flux de données créé, le modélisateur peut indiquer les traitements à effectuer sur les échantillons provenant de ce flux.

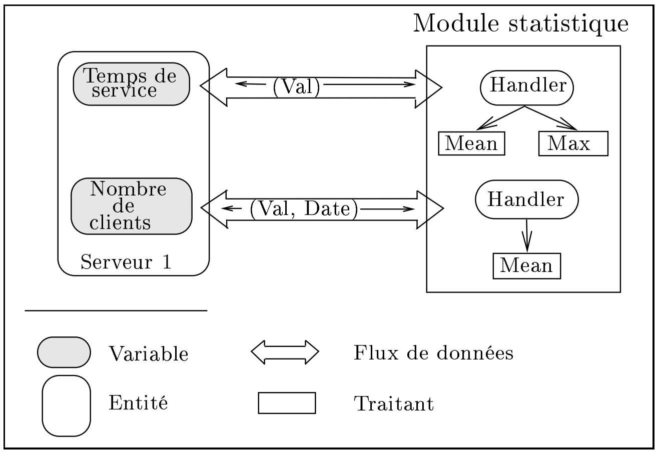


FIG. 13.2 – *Architecture de* STAT-TOOL

Au niveau de STAT-TOOL, un *handler* est associé à chaque flux de données. Le *handler* exécute les instructions provenant du flux (demande de résultat, nouveau traitement, etc.) et gère les *traitants*. Un *traitant* réalise un traitement statistique spécifique (moyenne, variance, histogramme, etc.) sur le flux de données. Rajouter une nouvelle fonctionnalité statistique consiste simplement à coder une nouvelle classe de traitant et à l'enregistrer. Les traitements statistiques actuellement implémentés sont : la moyenne, la variance, le calcul d'histogramme et le calcul de l'intervalle de confiance.

La distribution des calculs est une extension naturelle de STAT-TOOL. Les variables peuvent être séparées des traitants en associant aux flux de données une liaison réseau (cf. Fig. 13.3).

La version distribuée est composée d'un module maître qui peut dynamiquement créer des modules esclaves. Chaque esclave est un module statistique complet qui peut gérer plusieurs *handler*. La distribution est donc entièrement transparente à l'utilisateur. Afin d'améliorer encore les temps d'exécution, STAT-TOOL est capable de bufferiser les échantillons au niveau du flux de données et de distribuer la charge (la granularité de distribution est le *handler*) entre les esclaves. La distribution est motivée par les besoins croissants en ressources (CPU, mémoire) générés par les grandes simulations et certains traitements statistiques.

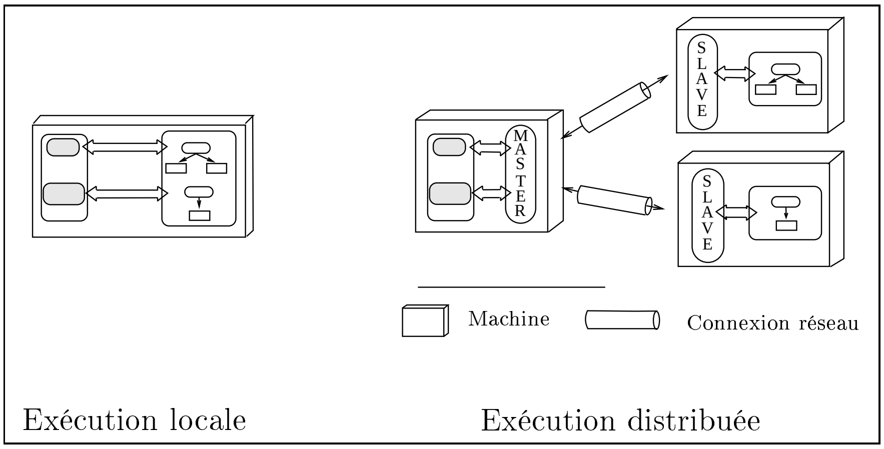


FIG. 13.3 – *Architecture* STAT-TOOL *locale et distribuée*

STAT-TOOL est écrit en C et possède une interface C++. L'outil comprend une librairie pour les exécutions séquentielles et une autre pour la distribution. Nous donnons en 13.3 un exemple d'utilisation directe de STAT-TOOL.

Intégration STAT-TOOL **à** PROSIT

Les statistiques sont incorporées au niveau des librairies de modélisation, chaque classe est responsable de son interface avec STAT-TOOL. Le noyau de simulation se charge de créer et d'initialiser le module statistique. Le programmeur peut accéder directement au module statistique et donc demander des traitements.

13.2.3 Interfaces graphiques et générateurs de code

Nous présentons dans cette section divers logiciels, adaptés ou développés, intégrés à l'environnement PROSIT et permettant : l'édition graphique de modèles, l'animation du déroulement d'une simulation et le suivi de l'exécution. L'objectif principal de ces outils est de faciliter la mise en œuvre de simulations, notamment pour les utilisateurs ne sachant pas programmer. Les objectifs secondaires étant la mise en place d'une plate-forme de formation et de démonstration, ainsi que l'évaluation de la facilité d'adaptation de l'architecture PROSIT.

Les résultats du sondage présentés dans [Cochran *et al.*, 1994] reflètent bien la demande, chez les utilisateurs, d'interfaces graphiques. Cette demande est particulièrement forte depuis les avancés récentes dans le domaine des interfaces hommes-machines (graphisme 3-D, animation temps réel, etc.). De nombreuses propositions d'interfaces graphiques ou de librairies

Exemple de code 13.3 Utilisation de STAT-TOOL

```
Stat_Module_Error error;
double date;
int sample1;
struct {int val; double time;} sample2;
//  creation of Stat-Tool
Stat_Module *stat = new Stat_Module;
// creation of an Entity
server = new Entity(1,"Server 1");
// creation of 2 Variables
service = new Variable(1,"Service Time",ST_Double(),server);
population = new Variable(2,"Nb of customers",ST_Int(),server);
// creation of the data streams
stat->create_stream(service,ST_Discrete,1);
stat->create_stream(population,ST_Continuous,1);
// analyses to perform
stat->compute(service, ST_Mean);
stat->compute(population, ST_Mean);
// transfer the samples
for (...) {
   .......
   stat->in(service,sample1);
   stat->in(population,sample2);
   .......
}
// result query
result1=(double *)(stat->result(service,ST_Mean,date,error));
result2=(double *)(stat->result(population,ST_Mean,date,error));
```

d'animation, dédiées à la simulation, ont donc vu le jour (voir par exemple [Hill, 1994, Hill *et al.*, 1993], [Bellonot et Duty, 1995], [Damitio *et al.*, 1993], [Jackson et Zobel, 1993]). Les interfaces présentées ici n'ont pas la prétention de rivaliser avec l'état de l'art.

Edition graphique de modèles

L'édition graphique de modèles et la génération automatique de codes sont des techniques classiquement utilisées pour réduire la phase de développement et améliorer la qualité des programmes. Mais comme toute programmation graphique, la puissance d'expression est limitée à celle contenue dans l'interface. Dans le cadre de Prosit, le code généré comprend l'instanciation et l'initialisation des composantes du modèle, le paramétrage de la simulation et son lancement, puis éventuellement l'affichage des résultats statistiques. Le code généré peut être modifié par l'utilisateur.

Deux éditeurs graphiques, l'un pour les files d'attente, l'autre pour les réseaux de Petri, peuvent être utilisés pour générer du code Prosit. Ces deux éditeurs sont basés sur l'outil Graphmod [Simulog, 1992b] de la société Simulog. La technique de génération ainsi que la nature du code produit varient entre ces deux interfaces (cf. Fig. 13.4).

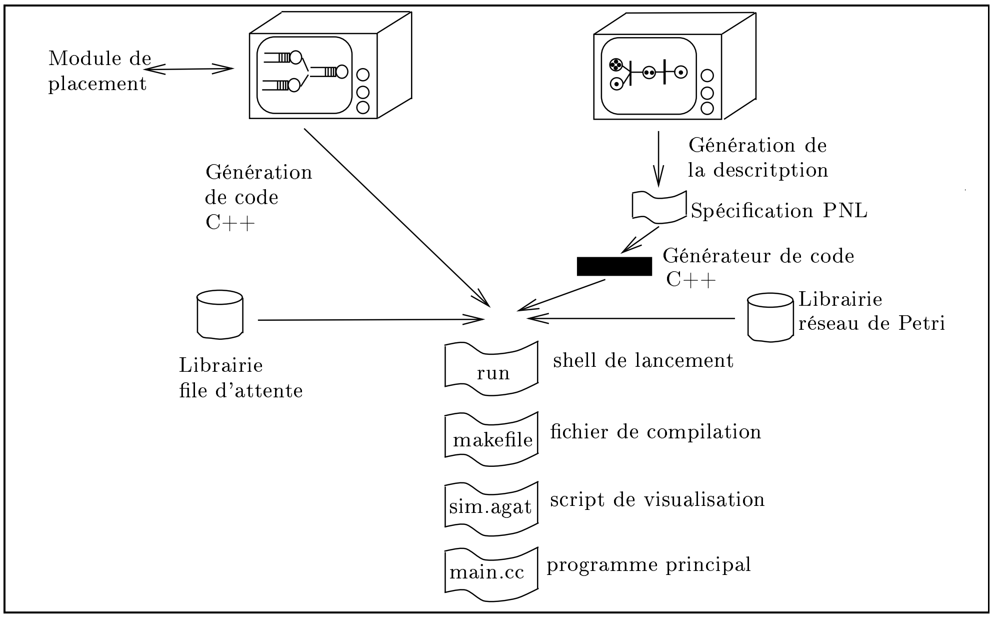


Fig. 13.4 – *Génération automatique de code* Prosit *à partir des interfaces graphiques*

L'interface graphique pour les files d'attente est codée en C++ et génère directement du code Prosit selon le formalisme client passif (cf. Fig. 13.5). Le code peut être généré pour une exécution séquentielle ou distribuée. Dans le cas distribué, le programmeur spécifie son placement et la politique de synchronisation désirée. Afin d'aider le programmeur à distribuer son modèle, nous avons développé et interfacé à l'outil graphique un module de placement automatique. La version actuelle du module est très rudimentaire et ne regroupe que les composantes fortements connexes du graphe (on obtient alors une topologie acyclique au niveau des clusters). Nous envisageons de compléter ce module avec des algorithmes plus complexes (voir par exemple [Loucks et Nandy, 1992, Loucks et Nandy, 1993], [Konas et Yew, 1995]). L'interface peut aussi partiellement générer du code QNAP2. Cette fonctionnalité a été rajoutée lors de la validation et de l'évaluation du noyau PROSIT et de la librairie files d'attente.

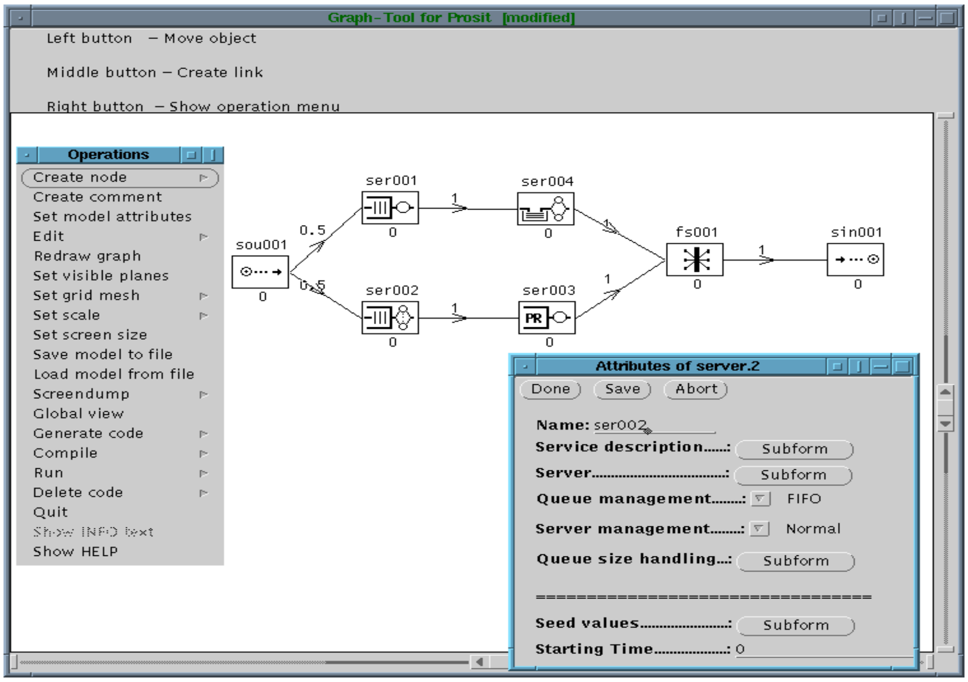


FIG. 13.5 – *Exemple de session de travail*

L'interface graphique pour les réseaux de Petri génère une description du modèle au format PNL[2]. L'utilisation de cette interface a donc consisté à écrire un programme de conversion PNL vers du code PROSIT.

Animation de simulation

L'animation consiste à visualiser post-mortem le déplacement des jetons ou des clients (cf. Fig. 13.6). L'animation est réalisée à partir d'un fichier de traces généré par TRACE-TOOL et d'un modèle construit à l'aide d'une des interfaces graphiques précédentes. L'outil d'animation est basé sur la librairie ANIMATOR [Simulog, 1992a] de la société Simulog.

Suivi d'exécution

Le suivi de l'exécution d'une simulation est particulièrement important notamment en phase de mise-au-point et de validation. Pour cela nous avons intégré à PROSIT un outil d'animation d'algorithmes et nous avons développé une interface de suivi de la simulation distribuée.

2. Le format PNL (Petri Net Language) est un format de description de réseaux de Petri conçu par A. Jean-Marie de l'équipe Mistral de l'INRIA [Jean-Marie, 1997].

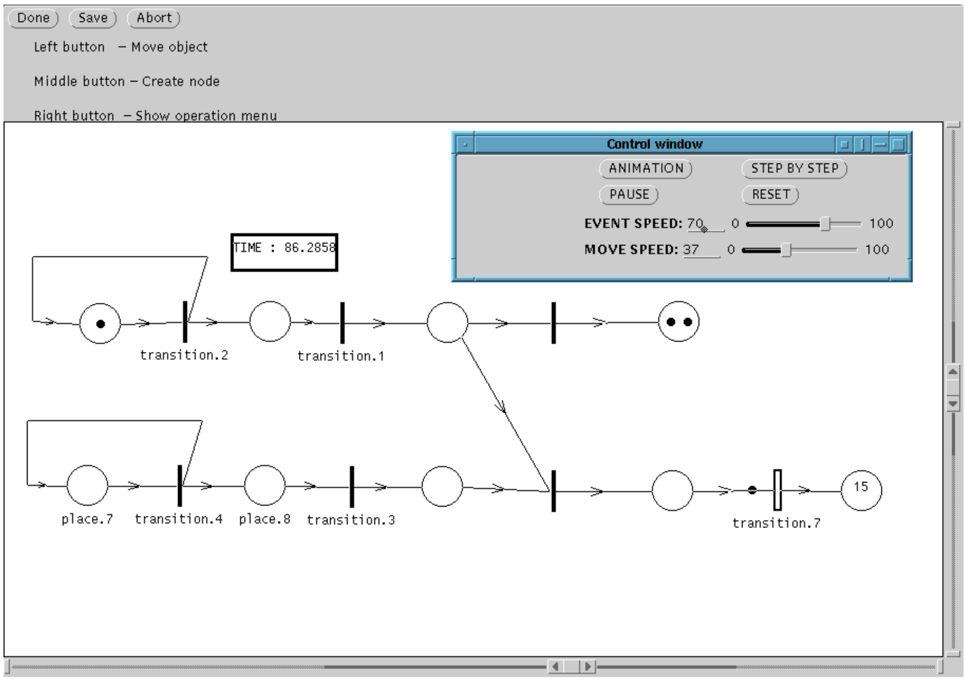


FIG. 13.6 – *Animation de la simulation d'un réseau de Petri*

Suivi de trajectoire Agat [Arsac *et al.*, 1993a, Arsac *et al.*, 1993b] est un outil d'animation d'algorithmes développé par le projet Safir de l'INRIA Sophia-Antipolis. Agat peut être vu comme un processeur de flux recevant des données du simulateur et gérant leur affichage sous formes graphiques. Le mode d'affichage est spécifié par le programmeur à l'aide d'un fichier script, l'affichage pouvant être direct ou post-mortem. Agat est utilisé pour visualiser l'évolution de variables (cf. Fig. 13.7). Les flux Agat peuvent être générés, soit par le programmeur, soit automatiquement. La génération automatique est réalisée en utilisant une interface spécifique à l'outil statistique STAT-TOOL (utilisation de la classe `Trace_Stat` au lieu de la classe `Module_Statistique`). Cette classe, pour chaque variable analysée, crée un flux et se charge d'émettre les données au module Agat.

Monitoring d'une exécution distribuée Nous avons développé un outil, INSPECTOR, capable de récupérer diverses informations d'une simulation distribuée en cours d'exécution, et de les afficher (cf. Fig. 13.8). Dans la version actuelle, les informations accessibles pour chaque cluster sont : le mode de synchronisation et la date courante du noyau, diverses informations sur les objets actifs (identifiant, nom, date, etc.), diverses informations sur les proxy (localisation de l'objet, etc.), et pour la synchronisation conservative, diverses informations sur les canaux (provenance, date, messages en attente, etc.).

La partie graphique utilise la librairie LEDA [Näher, 1995], les communications utilisent les messages actifs de SCHOONER.

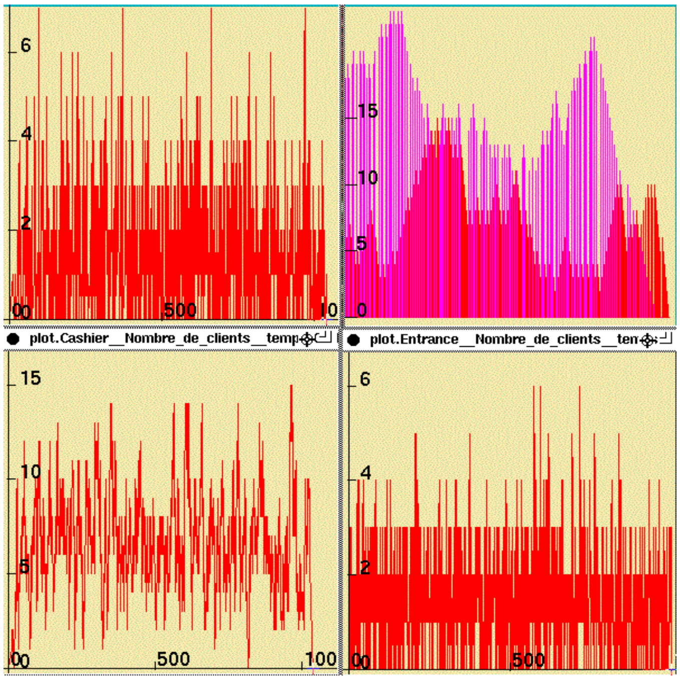


FIG. 13.7 – *Exemple de visualisation à l'aide d'Agat*

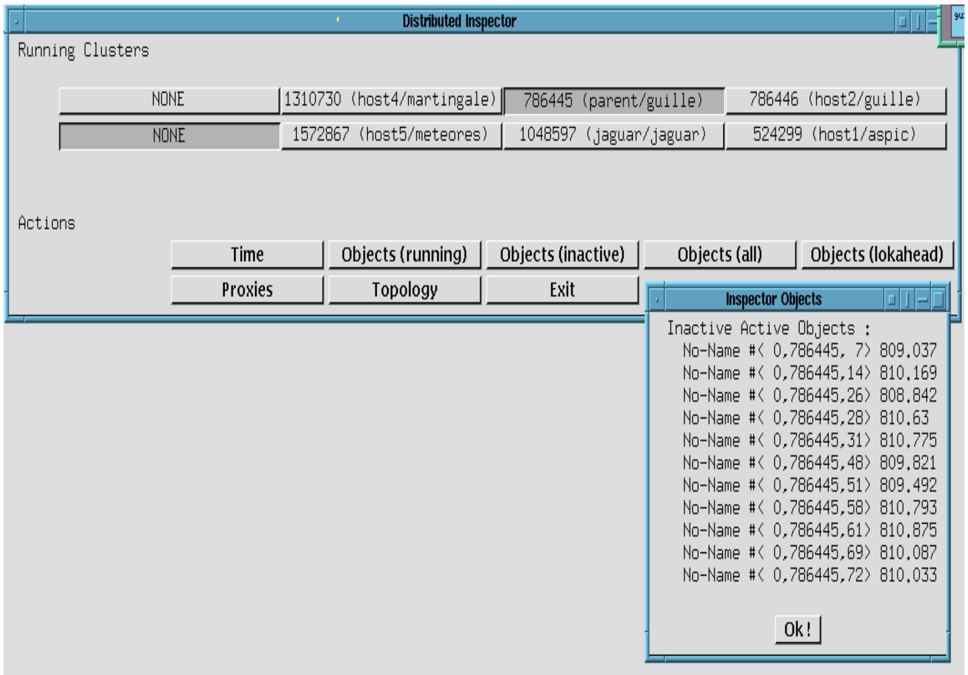


FIG. 13.8 – *L'interface graphique* INSPECTOR *permettant le suivi d'une simulation distribuée*

14. EVALUATION DU PROTOTYPE

Dans les chapitres précédents, nous avons présenté le modèle de programmation PROSIT et décrit sa mise en œuvre. Nous allons maintenant mener une évaluation détaillée du prototype. Une évaluation uniquement quantitative ignorerait de nombreux aspects important de nos travaux, comme la facilité de codage et de distribution des simulations, la réutilisabilité des composantes, etc. C'est pourquoi nous commencerons par présenter une évaluation qualitative du framework. Mais l'étude quantitative détaillée est nécessaire car elle permet de répondre aux questions suivantes :

- quel est l'impact, sur les performances du noyau, des choix retenus pour le modèle de programmation PROSIT et sa mise en œuvre ?
- quel est l'impact des caractéristiques du modèle de simulation sur les performances ? Pour quels types de modèles, le noyau se comporte-t-il le mieux ?
- quel est l'impact du support d'exécution (processeur, réseau, etc.) sur les performances ?

Nous avons commencé par évaluer les mécanismes de base du noyau en environnement séquentiel et distribué. Nous avons ensuite cherché à vérifier ces résultats partiels sur des simulations de réseaux de files d'attente. Parallèlement à l'évaluation de la librairie files d'attente, nous avons cherché à comparer PROSIT au simulateur QNAP2 [Simulog, 1995]. QNAP2 est un logiciel de modélisation et de simulation commercialisé par la société Simulog. Les modèles QNAP2 sont décrits selon le formalisme files d'attente. Cette comparaison [1] était motivée par le fait que QNAP2 est l'outil de simulation le plus utilisé dans notre projet, et par le fait que nous souhaitions vérifier que les mécanismes à la base de PROSIT (objet actif et réification) induisaient un coût acceptable. Nous avons ensuite mesuré les temps d'exécution de ces modèles en environnement distribué. Ce second groupe d'expériences nous a permis de calculer les accélérations obtenues avec les protocoles de synchronisation implémentés.

Signalons qu'une autre étude [Badel et Toulzat, 1997], moins exhaustive et ne portant que sur la librairie files d'attente, a été réalisée par la société Simulog.

1. Les résultats de la comparaison PROSIT/QNAP2 doivent être interprétés avec précaution. En effet, on compare un prototype de recherche à un produit commercial, constamment amélioré depuis 10 ans. Ces deux simulateurs sont aussi radicalement différents au niveau de l'implémentation. QNAP2 est écrit en Fortran et *interprète* ses modèles alors qu'une simulation PROSIT est un programme C++ compilé. Le modèle d'exécution PROSIT fait aussi appel à des mécanismes nouveaux comme la réification et l'utilisation de processus légers. Une autre évaluation qui sera très prochainement entreprise, consistera à comparer PROSIT à un simulateur de même génération.

14.1 Evaluation qualitative du framework

L'évaluation qualitative d'un environnement de développement par son *concepteur-réalisateur* est une tâche difficilement objective. Nous allons néanmoins chercher à évaluer le framework Prosit selon les critères de facilité d'utilisation, de puissance d'expression et enfin de facilité de distribution d'une simulation.

Nous considérons que le framework fournit un modèle de programmation naturel et très puissant, tout en étant relativement simple. La simplicité d'utilisation provient à la fois de l'approche objet, de la réification, de l'architecture adoptée et enfin du modèle de programmation et d'exécution développé. Mais cette simplicité est relative car à chaque niveau d'utilisation correspond un niveau de compétence :

1. *Utilisateur de base.* Le prérequis pour cette catégorie d'utilisateur est la connaissance du domaine de la simulation à événements discrets et une bonne maîtrise du langage C++ [2]. L'utilisateur devra ensuite se familiariser avec les concepts de base du modèle de programmation Prosit : objet actif, demande de service, etc. S'il utilise une librairie, il devra aussi assimiler les fonctionnalités disponibles. Cette phase d'apprentissage correspond à la partie (A) de la figure 14.1. Prosit a déjà été utilisé dans un tel cadre avec des résultats très positifs.
2. *Programmeur de librairie.* Le développement d'une librairie nécessite une connaissance avancée du langage et des techniques de développement en C++. Le programmeur doit aussi maîtriser les mécanismes avancées de Prosit comme la multi-activité, la simulation hiérarchique, etc. Cette phase d'apprentissage correspond à la partie (B) de la figure 14.1. Le développement de la librairie réseaux de Petri (cf. §13.1) entre dans ce cadre et a permis d'évaluer les bases nécessaires au développement d'une librairie de modélisation.
3. *Programmeur avancé.* Cette dernière catégorie intervient lors de la spécialisation du framework (nouvelle politique de synchronisation, nouvel échéancier, etc.). Ces utilisateurs ont une connaissance détaillée des mécanismes à la base du framework ce qui requière un apprentissage plus long (partie (C) de la figure 14.1). Les nombreuses expériences réalisées tout au long de cette thèse ont permis de vérifier la souplesse et la facilité de spécialisation du framework. Mais il est actuellement prématuré de généraliser à un autre programmeur. Le déroulement du projet Hipertrans (cf. §15.2) qui utilise Prosit, apportera les premières réponses.

La puissance d'expression du modèle de programmation provient aussi de l'approche objet, de l'utilisation de la réification, des choix de conception et de l'architecture d'exécution adoptée. La mise en place, sans extension du modèle, de fonctionnalités avancées (cf. Chap. 9) et les nombreuses réalisations effectuées (cf. Chap. 13 et 15) constituent une première illustration de la puissance d'expression du framework. La sélection de Prosit comme noyau de simulation pour le projet Européen Hipertrans (cf. §15.2) est aussi un élément allant dans ce sens. Mais seule une utilisation à plus grande échelle de Prosit permettra de vérifier que tout système à événements discrets peut en effet être simulé et que le modèle *objet actif* permet de représenter naturellement la dynamique de ces systèmes.

2. Nous avons présenté à la section 13.2.3 des interfaces graphiques permettant de se libérer de la phase de programmation C++.

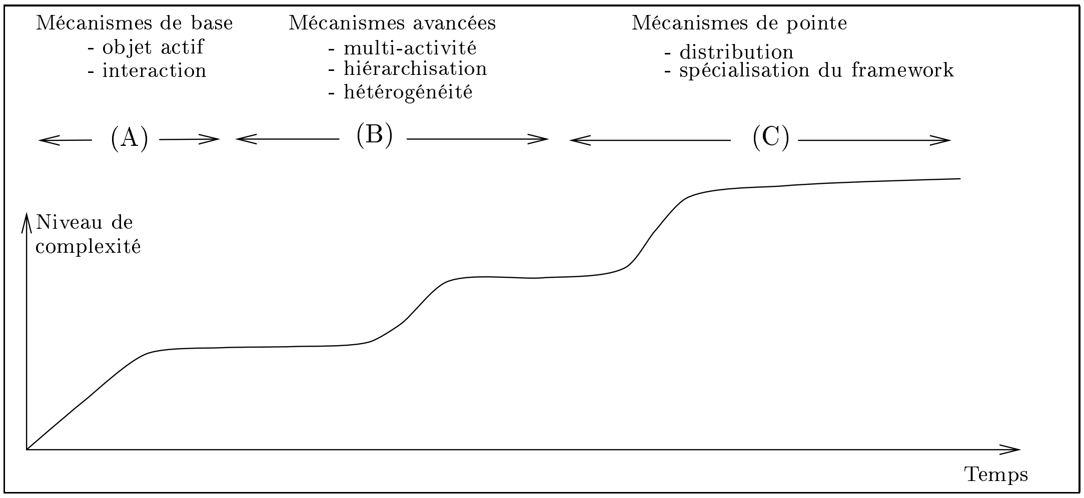


Fig. 14.1 – *Courbe d'apprentissage de l'utilisation du framework* Prosit

La distribution d'une simulation est une tâche intrinsèquement complexe qui requière : (1) la sélection du protocole de synchronisation et (2) le codage des classes distribuées et du programme principal. Le premier point est indépendant de notre approche et nous avons présenté à la section 3.3 les principaux éléments à prendre en compte lors de la sélection du protocole de synchronisation. Le codage des classes distribuées a été détaillé à la section 8.5 et la difficulté potentielle réside dans le codage du protocole d'interaction avec le noyau de simulation. Mais le véritable obstacle à la distribution tient généralement dans les faibles performances, voire les contre-performances, obtenues. L'optimisation d'une exécution passe par une gestion dynamique : de la migration des objets actifs, du protocole de synchronisation, etc. Or, le framework offre actuellement peu de mécanismes de suivi d'exécution[3] et aucun mécanisme adaptatif.

14.2 Configurations logicielles et matérielles utilisées

Nous avons effectué les expériences sur différentes architectures de processeurs et en utilisant différents réseaux de communication. Deux raisons expliquent cette diversité. Tout d'abord, notre collaboration avec la société Simulog nécessitait l'utilisation de la dernière version de Qnap2, la 9.3, qui n'était disponible que pour Solaris. Ensuite, nous désirions évaluer l'impact du processeur et du réseau de communication sur les performances des exécutions. Cette étude devait permettre de sélectionner une plate-forme cible pour Prosit et d'évaluer, du point de vue de l'utilisateur, le rapport entre l'accélération apportée et le coût induit par l'achat du nouveau matériel.

La configuration logicielle est la suivante :

– Tests séquentiels : Prosit V1.2.0, compilé en `-O -DNDEBUG`, sans les modules de traces et de statistiques. La version de Qnap2 est la 9.3 pour Solaris.

3. Les mécanismes disponibles sont les traces (cf. §13.2.1) et les interfaces graphiques de monitoring (cf. §13.2.3).

– Tests distribués : PROSIT V1.2.0 pour les classes du framework séquentiel et V0.3.0 pour les classes du framework distribué, compilé en `-O -DNDEBUG`, sans les modules de traces et de statistiques. La version de SCHOONER utilisée est la V1, aussi compilé en `-O -DNDEBUG`.

La configuration matérielle :

– Tests séquentiels. Nous avons utilisé trois configurations :

1. Sun Ultra2 - 188 Mo - SunOS 5.5 (i.e. Solaris 2.5),
2. Alphastation 500 - 256 Mo - 333 Mhz et Digital UNIX V3.2D-2,
3. PC Pentium Pro (i686) à 200Mhz et noyau Linux 2.0.18.

– Tests distribués. Nous avons utilisé deux configurations :

1. Alphastations 500, reliés par Ethernet 10Mb/s ou 100Mb/s,
2. PC Pentium Pro reliés par Ethernet 100Mb/s ou Myrinet [4].

Les expériences ont été réalisées dans un environnement dédié ce qui limite les facteurs extérieurs de perturbation. Néanmoins, les mesures données dans la suite sont une moyenne sur au moins 40 expériences.

14.3 Evaluation des mécanismes de base

14.3.1 Mécanismes séquentiels

Nous étudions dans cette section les mécanismes de base du noyau séquentiel. Nous mesurons les coûts de création d'un objet actif et d'une activité, d'un changement de contexte et celui des interactions entre les objets actifs. Nous étudions enfin le comportement du système en présence d'un grand nombre d'objets actifs.

Créations d'objets actifs et d'activités

Nous voyons ici (cf. table 14.1) que lors de la création d'un objet actif plus de 70% du coût est associé à la création de l'activité principale. Les 30% restants correspondent à la création et à l'initialisation des structures de données et à l'enregistrement auprès du noyau. Le coût de création d'une activité comprend la création et l'initialisation de l'objet `Stack` (33%), la création de l'objet `Activity` (5%) et l'initialisation de l'activité (42%).

La création d'une activité étant relativement coûteuse, le programmeur de librairie devra limiter au maximum cette opération. Pour cela, plusieurs techniques sont à sa disposition : il peut coder des comportements capables de servir directement les requêtes sans créer d'activités annexes, il peut recycler les activités annexes, etc.

4. Myrinet [Myricom, 1995] est un nouveau réseau local développé par la société américaine Myricom et qui est basé sur la technologie classiquement utilisée dans les machines massivement parallèles (communication bidirectionnelle, réseau switché, re-configuration automatique, programmation du routage possible, etc.).

Création	Temps en μs		
	Config 1	Config 2	Config 3
Un objet actif	15	7	5
Une activité	11	5	3

Tab. 14.1 – *Temps de création d'un objet actif et d'une activité*

A titre de comparaison (chiffres tirés de [Namist, 1996]), la création d'un processus léger sur Sparc5/ Solaris 85 Mhz est de 37 μs avec la librairie "marcel" [Namist, 1996], 45 μs avec la librairie de Mueller [Mueller, 1993], de 26 μs avec la librairie de Provenzano [Provenzano, 1996] et de 86 μs avec la librairie native Solaris. Sur cette architecture la création d'une activité Prosit est de 38 μs. Cette très bonne performance de Prosit est essentiellement due à la librairie de processus légers utilisée (cf. section 11.3.1).

Changement de contexte

Cette expérience (cf. table 14.2) montre qu'il y a peu de différences entre un changement de contexte objet-actif/objet-actif (exemple : noyau/station) et un changement de contexte objet-actif/activité (activité principale/activité annexe). C'est bien le résultat attendu dans la mesure où les opérations mises en œuvre sont sensiblement les mêmes.

Changement de contexte	Temps en μs		
	Config 1	Config 2	Config 3
objet actif/objet actif	2.2	0.3	0.11
objet actif/activité	1.9	0.24	0.1

Tab. 14.2 – *Evaluation des temps de changements de contexte*

Le coût d'un changement de contexte est très variable d'un processeur à l'autre et dépend de la quantité de données sauvegardées et restaurées (nombre et taille des registres) et de la disponibilité d'instructions spécialisées. Mais l'ordre de grandeur est toujours de plusieurs dizaines de cycles. Il est donc important qu'à chaque réactivation l'objet effectue le maximum d'opérations.

A titre de comparaison (chiffres tirés de [Namist, 1996]), un changement de contexte entre processus légers sur Sparc5/Solaris 85 Mhz est de 8 μs avec la librairie "marcel", 220 μs avec la librairie de Mueller, de 34 μs avec la librairie de Provenzano et de 48 μs avec la librairie native Solaris. Sur cette architecture, un changement de contexte entre objets actifs Prosit est de 6 μs. Là encore cette très bonne performance de Prosit est essentiellement due à la librairie de processus légers utilisée (cf. section 11.3.1).

Interactions entre objets actifs

L'interaction entre les objets actifs fait intervenir deux mécanismes : la réification des appels et l'ordonnancement des objets actifs. Il est donc naturel qu'un appel de service soit plus coûteux qu'un appel de méthode C++. Rappelons les opérations intervenant lors d'une demande bloquante de service :

1. construction et initialisation de l'objet requête (il y a notamment détermination dynamique de la fonction-service demandée),
2. émission de la requête, qui est reçue et stockée par l'appelé,
3. suspension de l'appelant,
4. activation de l'appelé qui récupère l'objet requête et exécute directement la fonction service,
5. construction et émission de l'objet réponse qui est reçu et stocké par l'appelant,
6. suspension de l'appelé et réactivation de l'appelant qui traite l'objet réponse et poursuit.

Les mesures montrent (cf. table 14.3) qu'un appel de service bloquant est de 20 à 70 fois plus coûteux qu'un appel de méthode virtuelle C++. Ce coût se répartit comme suit : 44% pour l'ordonnancement et les opérations effectuées par le noyau, 16% pour la réification (création et livraison des objets requête et réponse), 30% pour la gestion mémoire et 5% pour le comportement et le service des objets actifs.

La mesure, coté client, d'un appel non bloquant permet d'évaluer le coût de la réification de l'appel et de la livraison de la requête au serveur. Ces opérations coûtent environ quatre appels virtuels, ce qui correspond au nombre de fonctions nécessaires (construction/initialisation de la requête, émission réifiée, réception réifiée et enfin stockage). L'implémentation réalisée est donc optimale.

	Temps en μs		
Type d'appel	Config 1	Config 2	Config 3
Fonction membre virtuelle	0.55	0.28	0.3
Demande bloquante avec service directe	40	15	7
Demande non bloquante (mesure coté client)	2.5	1.2	1

Tab. 14.3 – *Temps pour les différentes catégories d'appel de fonction*

Dans le cas de l'appel de service bloquant, il y a quatre changements de contexte[5]. Une optimisation consisterait à fournir une version de Prosit permettant les changements de contexte directs d'objet actif à objet actif, sans passer par le noyau. On gagnerait ainsi deux changements de contexte et quelques appels de fonction. Une autre optimisation consisterait à transformer les appels de services instantanés (i.e. code algorithmique sans demande de service ni consommation de temps) en appels C++ classiques. Cette optimisation n'est néanmoins possible que dans le cadre séquentiel.

5. Ces changements de contexte sont : demandeur => noyau, noyau => serveur, serveur => noyau, noyau => demandeur.

Comportement en présence d'un très grand nombre d'objets actifs

Dans cette expérience, nous étudions le comportement de PROSIT en présence d'un grand nombre d'objets actifs. Chaque objet actif effectue le code donné à l'exemple 14.1. La consommation de temps est soit constante (i.e. `wait(1)`), soit le résultat d'un tirage selon une distribution uniforme entre 0 et 2.

Exemple de code 14.1 Comportement des objets actifs

```
void behave() {
  for (int i=0;i<200;i++) {
    wait(...); // consommation de temps
  }
}
```

Comme le montre la Fig. 14.2, le temps d'exécution est linéaire par rapport au nombre d'objets actifs. C'est un bon comportement qui montre que les performances du noyau ne s'écroulent pas lors d'une charge importante. La simulation avec consommation de temps uniforme est plus coûteuse que celle avec consommation de temps constante. Ce coût est essentiellement dû à la génération des temps de service et à l'utilisation différente de l'échéancier.

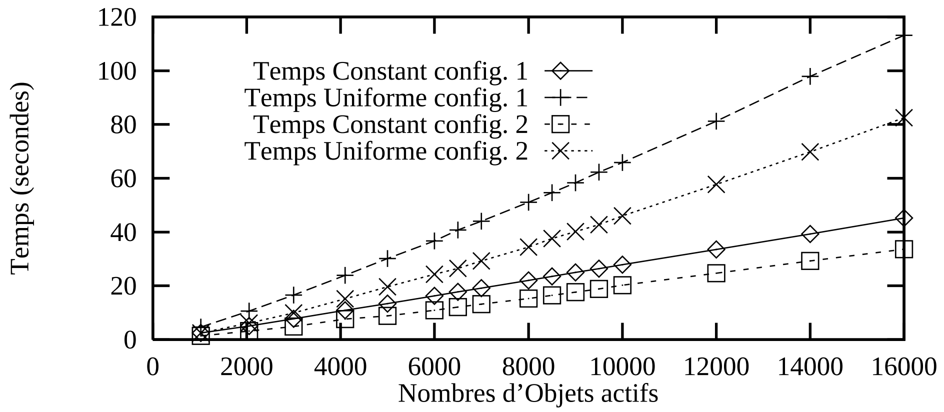


FIG. 14.2 – *Comportement en présence d'un grand nombre d'objets actifs*

14.3.2 Mécanismes distribués

Nous étudions dans cette section les mécanismes propres à la version distribuée du framework : création d'objets actifs, interactions entre objets distants et migration. Nous cherchons notamment à isoler et à caractériser les coûts issus du réseau de communication, de la librairie SCHOONER et de l'implémentation PROSIT.

Evaluation du support de communication

Préalablement à toute analyse des performances, il est nécessaire de mesurer les temps de communication en distinguant les délais apportés par chacune des couches constituant le support de communication. Ces mesures permettent d'isoler les coûts propres à l'implémentation et de recenser les points non optimisables par la programmation.

Trois réseaux ont été utilisés durant les tests : Ethernet 10Mb/s, Ethernet 100Mb/s et Myrinet. Nous donnons au tableau 14.4 leurs principales caractéristiques. Plutôt que de rapporter les performances théoriques, rarement atteintes, nous donnons les mesures effectuées sur notre configuration (ici la configuration 2) en utilisant les protocoles UDP/IP. Nous noterons ainsi que la latence de Myrinet est de 190 μS, alors qu'elle est d'environ 9 μS quand on utilise l'API constructeur. Nous donnons aussi à la figure 14.3, les temps de communication en fonction de la taille des messages émis, pour chacune des couches constituant le support de communication et chacun des réseaux. Ces mesures ont été réalisées en collaboration avec N. Furmento et O. Dalle du projet Sloop.

Réseau	Latence μS	Débit Mb/S	Caractéristiques principales
Ethernet	150	10	Faible débit et latence importante. Problème de contension d'accès au média.
Fast Ethernet	100	80	Le débit est plus important, mais la latence est sensiblement la même que pour Ethernet. Le gain ne sera significatif que pour de grands messages. Problème de contension d'accès au média.
Myrinet	190	160	Communications bidirectionnelles switchées. Interface IP disponible, mais seule l'utilisation de l'API constructeur permet d'obtenir les performances optimales

Tab. 14.4 – *Principales caractéristiques des réseaux utilisés (UDP/IP)*

De ces mesures, nous pouvons conclure que PVM induit un coût très important au niveau des communications. Ce surcoût est particulièrement pénalisant lors du transfert de petits messages. Schooner utilisant directement PVM pâtit de ce manque de performances. De plus Schooner introduit aussi un coût de communication non négligeable. Ce sur-coût à PVM est de l'ordre de 10 à 25% pour les petits messages (de 0 à 1 Ko) et décroît quand la taille des messages augmente. L'évaluation de Myrinet montre que les bonnes performances de l'API constructeur ne se retrouvent pas au dessus des couches TCP et UDP. Nous n'avons donc pas retenu cette architecture pour la suite de nos tests. Néanmoins, une version de PVM ou de Schooner utilisant directement l'API constructeur devrait offrir d'excellentes performances. Prosit bénéficiera directement de ces futur développements.

Création d'objets actifs

La création (cf. table 14.5) locale d'un objet actif est 4 fois plus coûteuse qu'une création en environnement séquentiel. Ce coût plus important est dû aux opérations supplémentaires du mécanisme de proxy. Le coût important de création d'un objet distant (50 fois plus sur la même machine et 100 fois plus sur une autre machine, pour la Config. 2) est essentiellement dû à la mise-à-jour, dans le proxy, de l'identifiant de l'objet.

Une optimisation à venir consistera à ne plus attendre la confirmation de la création

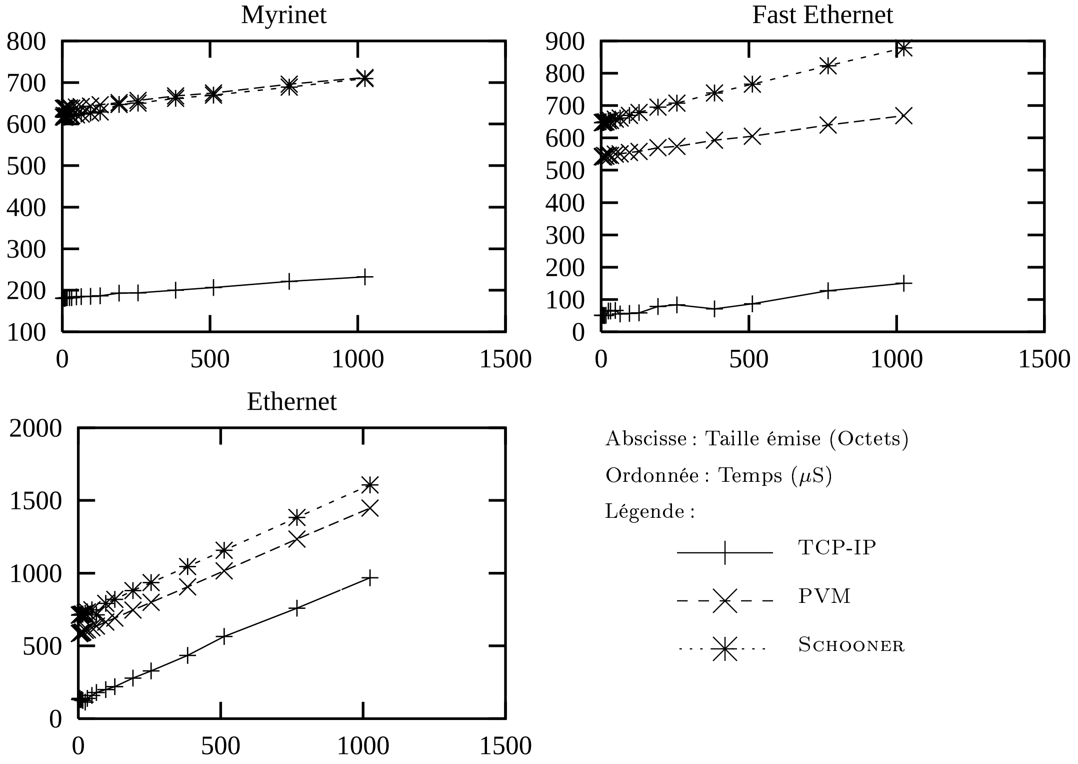


FIG. 14.3 – *Temps de communication pour les différents réseaux*

distante (voir §12.3.3). Tout accès à l'objet, alors que la confirmation n'est pas arrivée, sera bloquant. Nous n'avons pas retenu cette approche dans un premier temps, car l'échec d'une création ne sera signalé qu'au moment de l'accès et non plus au moment de la création. Ce qui pose des problèmes de robustesse du code et de conformité avec une exécution séquentielle.

Création d'un objet actif	Temps en μs			
	Config 1 10 Mb/s	Config 1 100 Mb/s	Config 2 100 Mb/s	Config 2 Myrinet
Création locale sur le même cluster	39	39	20	20
Création sur un cluster situé sur le même computer	1952	1952	1857	1846
Création sur un cluster situé sur un autre computer	3904	2928	2533	2400

TAB. 14.5 – *Temps de création d'un objet actif en environnement distribué*

Interactions entre objets actifs

Nous voyons (cf. table 14.6) ici qu'une demande de service à un objet situé sur une autre machine est environ 150 fois plus coûteux qu'un appel local. Ce coût important (deux ordres de grandeur) est en partie dû au support de communication. En effet près de 50% du temps est passé en communication. A titre de comparaison, en PM^2 [Namist, 1996], le coût d'un appel distant asynchrone est de 550 μS, et celui d'un appel synchrone est de 1300 μS. Nous avons aussi mesuré le coût d'un appel RPC sans paramètre : 700 μS.

Type d'appel	Temps en μs			
	Config 1 10 Mb/s	Config 1 100 Mb/s	Config 2 100 Mb/s	Config 2 Myrinet
Appel réifié local	19	19	19	19
Appel réifié distant (même computer)	1952	1952	833	909
Appel réifié distant (autre computer)	2928	2928	1250	1250

Tab. 14.6 – *Interaction entre objets actifs*

Binding

Le mécanisme de binding est une opération peu coûteuse (cf. table 14.7) quand le proxy existe localement (il y a simple recherche dans une table) ou quand l'objet actif est local (deux recherches dans des tables et création de l'objet proxy). Dans le cas contraire, c'est la diffusion (et la réception du résultat) du message de résolution d'adresse qui est coûteux.

Mode	Temps en μs			
	Config 1 10 Mb/s	Config 1 100 Mb/s	Config 2 100 Mb/s	Config 2 Myrinet
Proxy déjà en local	1	1	1	1
Objet local sans proxy local	16	16	7	7
Objet distant sans proxy local même computer	2928	2928	2000	1666
Objet distant sans proxy local autre computer	3904	3904	2666	2666

Tab. 14.7 – *Evaluation du mécanisme de binding*

Migration d'objets actifs

Nous mesurons ici le temps de migration d'un objet actif en fonction de la quantité de données transférées. Dans la version actuelle du prototype, la quantité minimale de données transférée est de 1900 octets. Ce minimum est dû aux informations système transférées et à la portion de pile déjà utilisée. Afin d'évaluer les performances, nous avons mesuré le temps d'exécution d'une simulation avec un unique objet actif effectuant un *ping-pong* entre deux clusters (cf. l'exemple de code 14.2). Nous donnons à la figure 14.4 le temps de migration en fonction de la quantité de données transférées (la configuration 1 est utilisée). A titre de comparaison, nous donnons aussi le temps d'une opération d'envoi de message Schooner déplaçant la même quantité de données. En PM^2, le temps de migration d'un *thread* entre deux processus situés sur la même machine est de 700 μS (2Ko transféré).

Comme on peut le constater, le mécanisme de migration d'un objet actif tire profit du réseau plus rapide à 100 Mb/s. En effet, nous avons ici des transferts de données nettement plus importants que lors des mesures précédentes. Nous voyons aussi que le coût des traitements

Exemple de code 14.2 Comportement d'un objet actif *Ping-Pong*

```
Cluster *cl;
for (int i=0; i<200; i++) {
  // Main Cluster ==>Remote Cluster
  cl = Cluster::others()->get_any();
  migrate(cl);
  // Remote Cluster ==>Main Cluster
  cl = Runtime::conf_manager()->get_parent_cluster();
  migrate(cl);
}
```

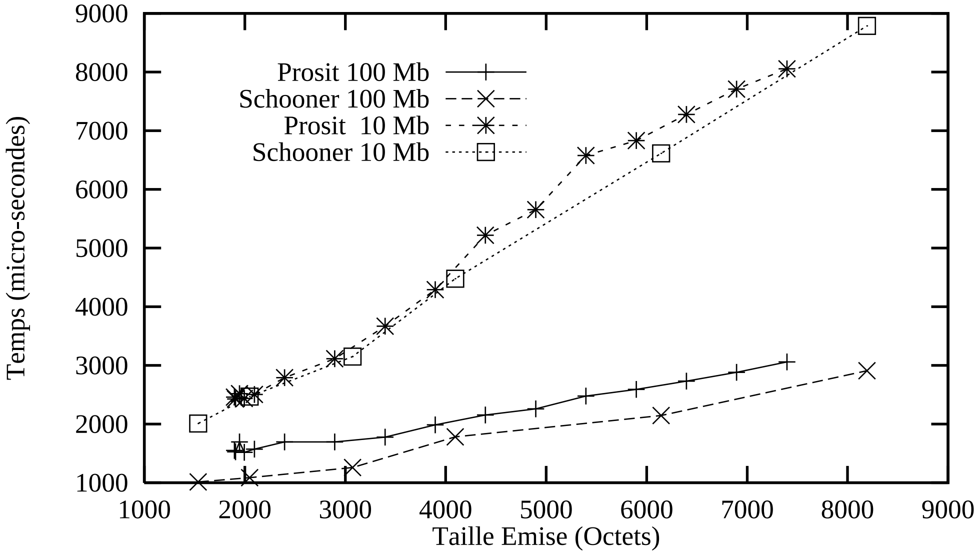


FIG. 14.4 – *Etude des temps de migration et de communication*

associés à la migration est faible par rapport à celui de la communication. L'overhead est de l'ordre de 20%.

Si nous comparons le coût d'un appel de service distant (2928 μS) avec le coût d'une migration, deux situations sont favorables à la migration :

- les objets sont de petites tailles (moins de 100 octets de données), le réseau peut alors être lent (10 Mb/s) ;
- les objets sont de grandes tailles (jusqu'à 5000 octets de données), le réseau doit être rapide (100Mb/s).

Ce rapport des coûts s'explique par le fait qu'un appel distant nécessite deux communications, contre une pour la migration. Mais nous n'avons pas pris en compte le phénomène de recouvrement calcul-communication qui peut avoir lieu dans une simulation, ni les coûts associés à la synchronisation.

La même expérience réalisée avec la seconde forme de migration (migration avec création d'une nouvelle activité cf. §12.4.2) est de 20 à 40% plus rapide et il y a 1800 octets de données transférées en moins (ces données correspondaient à la partie utilisée de la pile). Le gain de cette approche est largement réduit par la latence importante du réseau.

14.3.3 Conclusions sur l'évaluation des mécanismes de base

Cette première étude nous a permis d'évaluer les coûts associés aux mécanismes supportant le framework. Ces coûts ne sont pas négligeables, mais sont du même ordre de grandeur que d'autres réalisations. Nous pouvons alors considérer qu'ils sont acceptables au vu des fonctionnalités apportées. La mise en œuvre d'une simulation devant avoir des temps d'exécution minimaux, demande du programmeur une attention particulière au nombre d'objets actifs et d'activités, aux interactions et à la quantité de travail effectué à chaque activation. Nous présentons à la section 14.6 diverses techniques d'optimisations.

Pour la composante distribuée, nous avons vu que l'utilisation de SCHOONER et de PVM induisaient un coût très important lors du transfert de petits messages. Or, la plupart des communications en PROSIT font intervenir de petits messages (message nul 36 octets, appel de service distant environ 180 octets, création d'un objet à distance environ 100 octets). Les simulations distribuées seront pénalisées par cet overhead. Deux approches sont alors possibles, utiliser une autre librairie de passage de messages que PVM (ce qui ne devrait pas poser de problème avec SCHOONER), ou faire appel aux techniques d'agrégation de messages. Nous avons aussi évalué le coût de la migration d'un objet actif et conclu que cette technique était une alternative performante par rapport à l'appel de service distant. Néanmoins, l'utilisation d'un réseau rapide (100 Mb/s) s'avère nécessaire. Il nous reste à valider cette approche et à étudier les interactions entre la migration, la synchronisation et le recouvrement calcul-communication.

14.4 Evaluation de la version séquentielle de la librairie files d'attente

14.4.1 Station unique

Dans cette première expérience, nous cherchons à évaluer le coût apporté par la librairie files d'attente. Pour le mesurer, nous simulons un serveur avec un unique client passif rebouclant. Le modèle est codé à partir de la librairie files d'attente, puis directement à l'aide des classes de simulation. Nous codons aussi le même modèle avec un client actif, afin de comparer les deux approches. Le temps de service est d'une unité de temps et la simulation s'arrête à 10^6 unités de temps. La configuration 1 est utilisée.

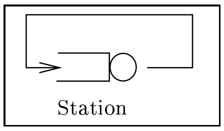


FIG. 14.5 – *Le modèle station unique*

Simulation	Temps en Secondes
PROSIT clients passifs classes de simulation	32
PROSIT clients passifs librairie files d'attente	45
PROSIT clients actifs	73
QNAP2	16

TAB. 14.8 – *Temps de simulation du modèle "station unique"*

De cette expérience, nous pouvons conclure qu'une simulation au-dessus de la librairie files d'attente aura un temps d'exécution plus important (40% ici) qu'une simulation directement codée à l'aide des classes actives. Ce coût supplémentaire a deux origines. La première est que cette librairie possède de nombreuses fonctionnalités (statistique, routage, etc.) qui induisent des traitements supplémentaires. La seconde vient du fait que l'on utilise ici une librairie prototype, non encore optimisée.
Comme indiqué à la section 9.1.1, nous voyons que l'utilisation du modèle client-actif est plus coûteux (60% ici) que l'utilisation du modèle client-passif. En effet, les opérations d'ordonnancement et de réification sont alors plus nombreuses. Par exemple, entre une fin de service et une nouvelle demande de service par ce client, il y a :

- dans le cas client passif, une demande de service asynchrone avec le client en paramètre;
- dans le cas client actif, livraison de l'objet réponse au client, réactivation du client qui fait une nouvelle demande synchrone à la station suivante.

La comparaison avec QNAP2 n'est pas favorable à PROSIT (50% à 180% plus coûteux). Nous payons ici l'aspect généraliste de PROSIT et la souplesse de ses mécanismes. De plus, la simulation d'une simple file d'attente est défavorable à PROSIT dans la mesure où à chaque activation, très peu de travail est réalisé (simple tirage d'une distribution). Il nous reste à voir si ce rapport se généralise lors de la simulation de grands modèles ou lors de la simulation de stations possédant un code algorithmique important.

14.4.2 Modèle linéaire

Nous cherchons ici à vérifier la stabilité du système en présence d'un grand nombre de stations. Nous comparons aussi les temps de simulation lors d'un service direct de la requête, et lors d'un service via une activité annexe. Nous effectuons enfin la simulation selon le formalisme client actif afin de comparer les deux approches.

Le modèle simulé est une série de files en tandem. La station source génère 10000 clients selon une distribution uniforme(1,2). Chaque station comprend 1 serveur et sert les clients de manière FIFO, avec un service composé de 3 consommations de temps selon la distribution uniforme(2,10). La simulation s'arrête quand il n'y a plus d'objets à réactiver. La configuration 1 est utilisée.

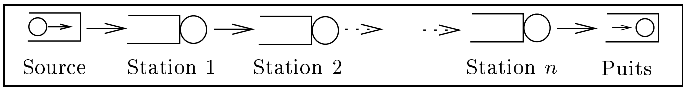


FIG. 14.6 – *Modèle linéaire*

Nous donnons à la figure 14.7, les temps de simulation en fonction du nombre de stations. Nous pouvons remarquer que les temps d'exécution PROSIT sont linéaires en fonction du nombre de stations et du nombre d'événements. Nous retrouvons donc le même comportement que celui de la section 14.3.

Nous obtenons aussi un rapport de 2, lors de la comparaison entre une simulation où les objets actifs servent directement les requêtes et celle où les requêtes sont servies par des activités annexes. Ce rapport est conforme aux mesures de la section précédente. En cas de service algorithmique important, ce rapport sera nettement plus faible.

La comparaison avec QNAP2 montre qu'en présence de services simples les performances de PROSIT se rapprochent de celles de QNAP2, voire les dépassent, quand le nombre d'objets à gérer est important. Dans le cas de service via une activité annexe, PROSIT est de 1.5 à 3 fois plus coûteux que QNAP2 jusqu'à 450 stations, puis ses performances se rapprochent de celles de QNAP2 en restant toujours légèrement inférieures. Dans le cas de service direct, dès 200 stations, PROSIT n'est plus que 1.5 fois plus coûteux que QNAP2 et à partir de 450 stations, il est plus performant. Les bonnes performances de PROSIT viennent de la structure en *splay-tree* de l'échéancier gérant les objets actifs et de la hiérarchisation des échéanciers (l'échéancier principal gère les objets actifs et chaque objet possède un échéancier pour les traitements internes).

La simulation avec clients actifs, si elle apporte de nombreux avantages, restera, dans la plupart des cas, plus coûteuse qu'une exécution avec clients passifs. Elle aura aussi une image mémoire plus importante. A la figure 14.8, nous donnons le rapport des temps de simulation pour les deux approches. Cette courbe montre que pour ce modèle l'approche clients actifs est 25% à 35% plus coûteuse.

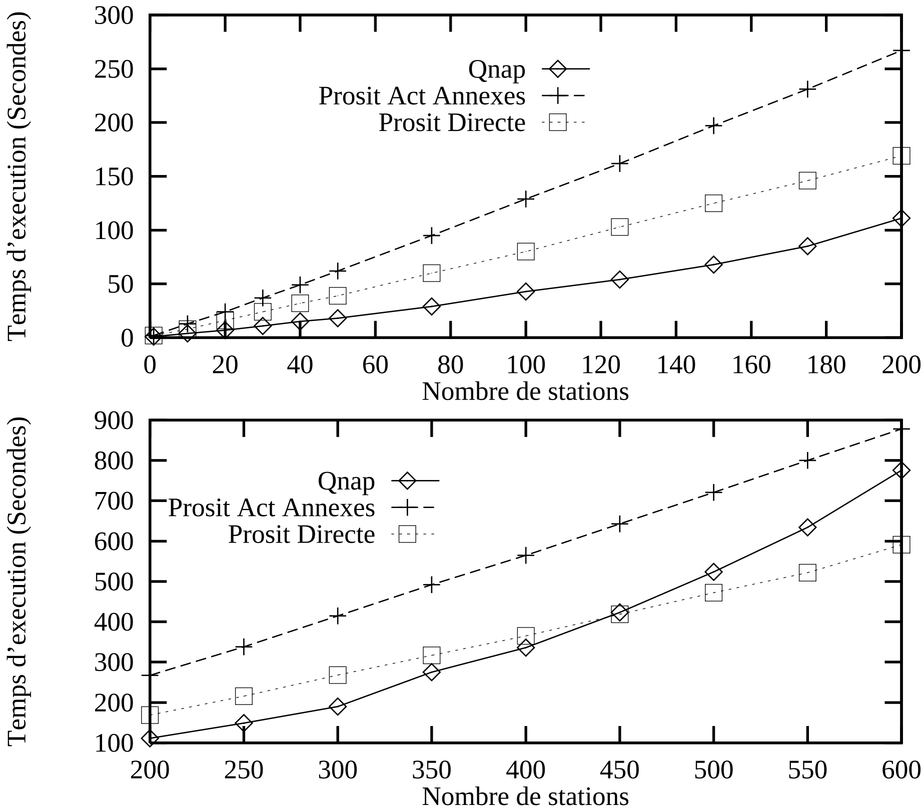


FIG. 14.7 – *Temps de simulation du modèle* linéaire *en fonction du nombre de stations*

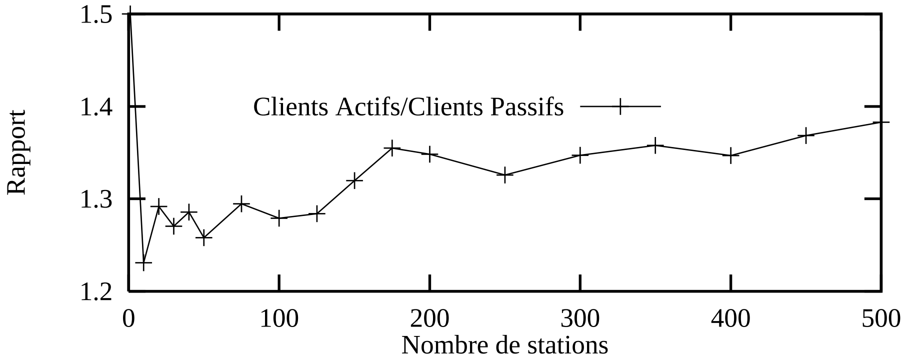


FIG. 14.8 – *Rapport des simulations avec clients actifs et clients passifs*

14.4.3 Code algorithmique

Nous avons vu précédemment que la création et la gestion des activités services possédaient un coût fixe. Ce coût est pénalisant lors de services simples (comme c'est le cas avec les files d'attente), mais peut être négligé lors de services algorithmiques complexes. De plus, le code algorithmique en Prosit étant compilé, nous aurons un net avantage par rapport à Qnap2[6].

Nous simulons le modèle présenté à la figure 14.9. La source génère les clients selon une distribution uniforme(0.2,0.4) et les route de manière équiprobable vers une des stations du premier niveau. Les stations du premier niveau servent les clients selon la politique FIFO et avec un temps de service selon une distribution uniforme(1,2). Le client est ensuite routé vers la station du second niveau ayant la file la moins pleine. Les stations du second niveau servent les clients selon la politique FIFO et avec un temps de service uniforme(1,2). La station du dernier niveau sert les clients selon la politique FIFO et avec un temps de service uniforme(0,0.3). Toutes les stations effectuent à chaque service une boucle de n fois le code suivant $c+ = a * b/3.14$ (a,b,c réels). Les requêtes sont servies via une activité annexe. Arrêt de la simulation à 10000 unités de temps. La configuration 1 est utilisée.

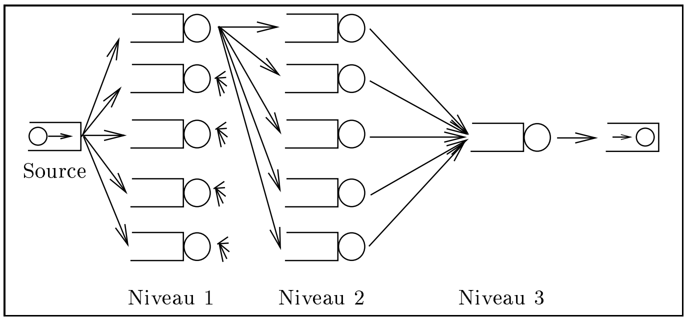


Fig. 14.9 – *Modèle pour la simulation avec code algorithmique*

Nous voyons à la figure 14.10 que le temps de simulation Prosit est peu sensible au nombre d'opérations effectuées à chaque service. Ce qui n'est par exemple pas le cas de la simulation Qnap2.

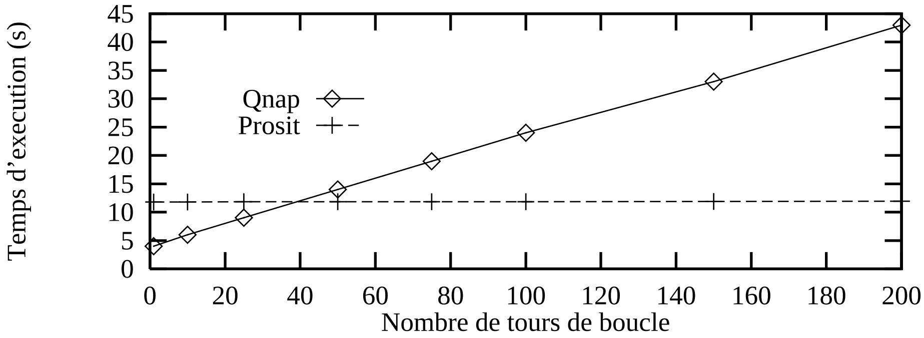


Fig. 14.10 – *Temps de simulation du modèle avec code algorithmique*

6. Notons qu'il est possible d'interfacer du code Fortran avec Qnap2.

14.4.4 Conclusions sur l'évaluation de la librairie séquentielle

Cette étude de la simulation séquentielle de files d'attente confirme et complète les conclusions de la section 14.3.3. Quand le seul critère est la performance, PROSIT est un bon candidat pour la simulation de grands modèles ou la simulation de modèles possédant beaucoup de code algorithmique. Dans les autres cas, les performances de PROSIT sont moyennes, tout en restant acceptables.

Comme prévu, la modélisation et le codage selon l'architecture client actif donnent lieu à des simulations plus coûteuses en ressources CPU et mémoire. Au vu de ces résultats numériques, nous pouvons nous poser deux questions :

- La simulation orientée processus est-elle trop coûteuse?
- L'architecture client actif est-elle trop coûteuse?

Ces deux questions ne sont pas distinctes car l'architecture client actif peut être vue comme une extension du modèle orienté processus. Tous deux ont les même objectifs et cherchent à regrouper les actions composant une séquence logique (ensemble des événements d'un service, ensemble des événements composant le contrôle d'un objet, ensemble des demandes de service d'un client). Il est difficile de répondre dans l'absolu à ces deux questions car la notion de coût est relative et dépend à la fois des objectifs de la simulation, de la machine disponible, etc. Nous considérons que les fonctionnalités apportées (facilité de modélisation, de codage, de mise au point et de maintenance, réutilisabilité des classes, facilité de distribution) compensent très largement les performances moyennes lors de la simulation de certains modèles. Dans le cadre de projets où les temps d'exécutions sont critiques, le modélisateur devra sélectionner l'architecture de ses modèles (clients actifs ou passifs) en fonction des performances attendues. Nous montrerons aussi à la section 14.6 comment se passer de l'approche orientée processus en PROSIT.

L'utilisation de la librairie files d'attente, encore au stade de prototype, est aussi pénalisant. Mais, même après une optimisation importante, les simulations utilisant une librairie générique auront généralement des temps d'exécution plus importants que celles faisant appel à des classes dédiées.

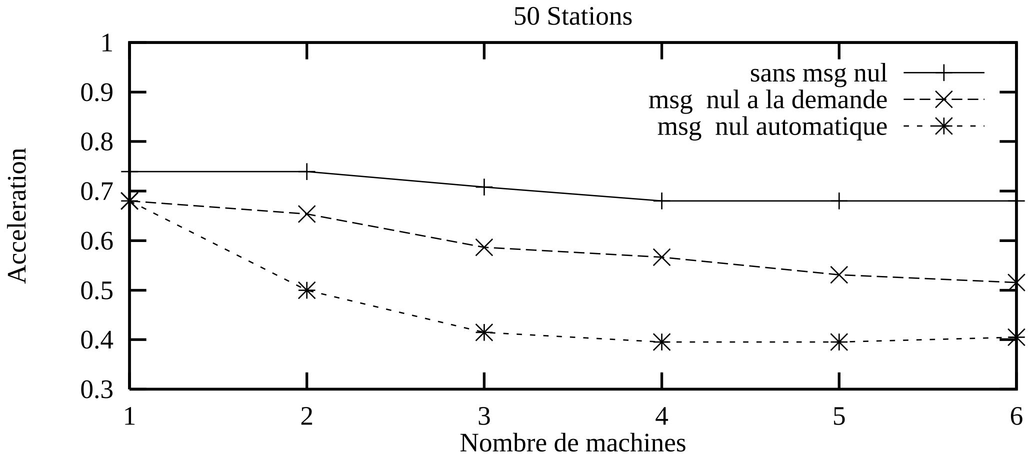
14.5 Evaluation de la version distribuée de la librairie files d'attente

Nous évaluons dans cette section le comportement du noyau en environnement distribué et avec le protocole conservatif. Nous donnons pour chacune des expériences l'accélération qui est calculée à partir du temps d'exécution d'une simulation séquentielle sur la même machine. Ce calcul est possible grâce à la disponibilité de la version séquentielle du framework. Nous considérons que l'évaluation de l'accélération est ainsi *plus juste*. En effet, on rencontre très souvent dans la littérature un calcul réalisé à partir du temps d'exécution d'une simulation distribuée effectuée sur une seule machine (exécution bien moins performante qu'une version séquentielle). Nous donnons aussi dans certains cas une mesure de *l'efficacité* de la distribution. Cette valeur s'obtient en divisant l'accélération par le nombre de machines utilisées.

14.5.1 Modèle linéaire

Nous reprenons ici le modèle de la section 14.4.2, mais avec les paramètres suivants: génération des clients selon une distribution uniforme(0,5) et service via une activité annexe selon la distribution uniforme(1,5). La simulation est arrêtée à 20 000 unités de temps. Nous simulons les deux cas extrêmes, 50 et 500 stations, en faisant varier le nombre de clusters. Les stations sont réparties de manière équilibrée sur les clusters. La configuration 1 et le réseau Fast Ethernet sont utilisés.

Architecture client passif La topologie entre clusters étant acyclique et linéaire, l'approche conservatrice sans messages nuls peut être utilisée. Elle permet d'obtenir les meilleures performances. Lors de l'utilisation de messages nuls, l'approche à la demande est plus performante que celle où les messages nuls sont émis automatiquement. Cette situation vient du fait que la version actuelle du noyau émet beaucoup de messages nuls (dix fois plus que lors de l'émission à la demande), et comme nous l'avons vu l'émission d'un message est une opération particulièrement coûteuse. Les meilleures performances de ce protocole peuvent aussi, en partie, s'expliquer par le nombre important de messages transitant par les canaux logiques. Ces messages arrivant suffisament *tôt*, les clusters sont rarement en attente et émettent peu de messages de synchronisation. Ce mécanisme a été étudié par Fujimoto dans [Fujimoto, 1989]. Dans cet article, l'auteur montre qu'à partir d'un certain nombre de messages circulant dans le réseau (il parle de *seuil d'avalanche*) le protocole à la demande est plus efficace que le protocole automatique.
Cette expérience confirme bien les conclusions présentées à la section 14.3.3. La distribution du modèle de 50 stations n'est source d'aucune accélération, dans la mesure où le rapport calcul/communication est faible. Nous avons par contre une accélération dans le cas 500 stations du fait de l'augmentation de ce ratio.



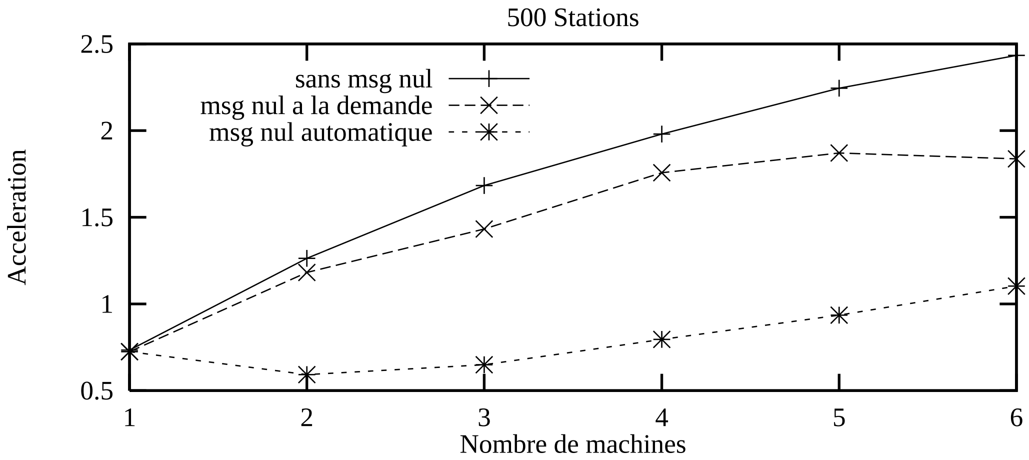


Fig. 14.11 – *Accélération du temps de simulation du modèle* linéaire *en fonction du nombre de clusters*

Architecture client actif Nous simulons maintenant le modèle en utilisant l'architecture client actif. Nous utilisons 5 clusters et nous faisons varier le nombre de stations, le nombre de clients et la date de fin de simulation. Grâce à la migration des clients, nous avons toujours une topologie acyclique et linéaire, et comme précédemment c'est la simulation sans messages nuls qui donne la meilleure accélération. Dans tous les cas de figure, plus il y a de clients et de stations (plus le ratio calcul/communication est important), plus l'accélération est grande (cf. Fig. 14.13).

Nous avons aussi cherché à simuler ce modèle sans faire migrer les clients. Ces derniers sont alors regroupés sur le premier cluster et nous avons alors une topologie cyclique comme indiquée à la figure 14.12. Les performances de la simulation sont alors catastrophiques, et nous obtenons des temps d'exécution 100 fois plus importants que lors des exécutions avec migration des clients. Ces mauvaises performances s'expliquent par le nombre considérable de demandes de service distants et par l'importance de la synchronisation due à la topologie.

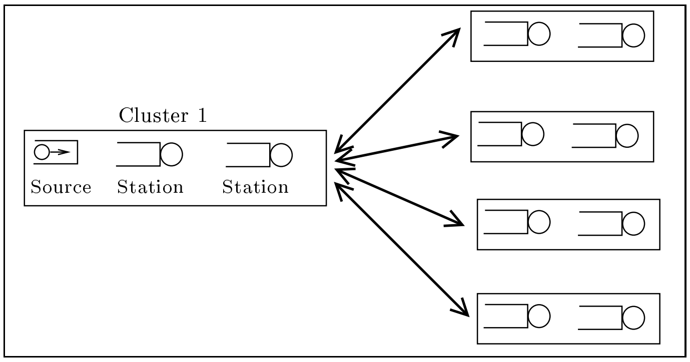


Fig. 14.12 – *Topologie pour le modèle linéaire sans migration de clients*

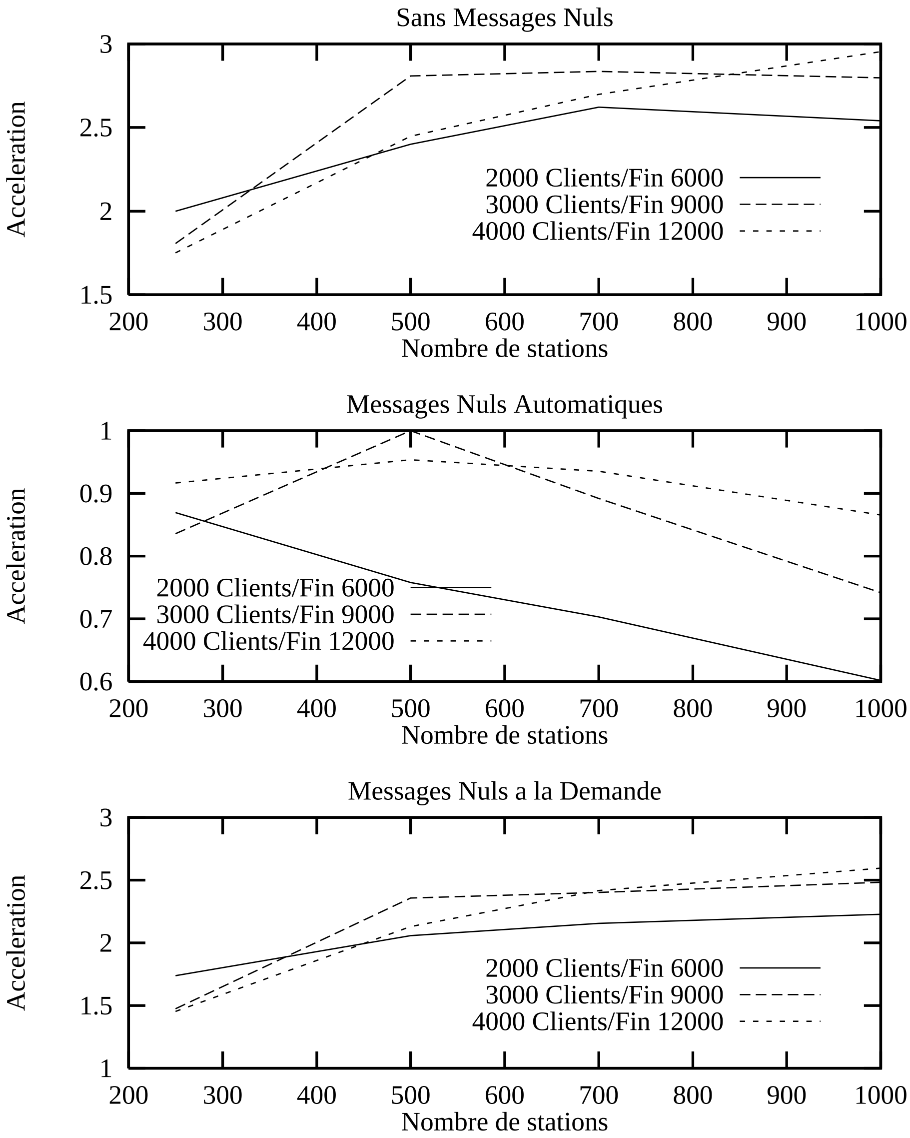


Fig. 14.13 – *Accélération du temps de simulation du modèle* linéaire *avec migration de clients actifs (5 clusters)*

14.5.2 Modèle atelier

Nous cherchons ici à étudier le comportement du système lors de la simulation de modèles à topologie cyclique. Nous avons pour cela mis en place la simulation d'un hypothétique atelier de fabrication. Le modèle est présenté à la figure 14.14 et comprend quatre sous systèmes de fabrication symétriques composés d'une source et de 200 stations. Chaque source génère 10000 clients selon une distribution uniforme(1,3). Une fois généré, le client est servi par la première station de son système. A la sortie de la première station, chaque client peut, soit passer au système suivant avec la probabilité λ, soit poursuivre avec la probabilité $1 - \lambda$. Un client qui poursuit sera servi par les 199 stations du sous-système et à nouveau par la première station.

Le dernier sous-système est un cas particulier, car à la sortie de la première file, le client peut soit terminer avec une probabilité de λ, soit partir vers le premier sous-système avec une probabilité de β soit boucler une probabilité de $1 - \lambda - \beta$. Les stations servent les clients via une activité annexe selon trois distributions uniforme(0.25,0.5). La simulation termine à 10000 unités de temps. Nous simulons ce modèle en faisant varier les paramètres λ et β, ce qui revient à faire varier le ratio calcul/communication. Nous utilisons 4 clusters, un par sous-système.

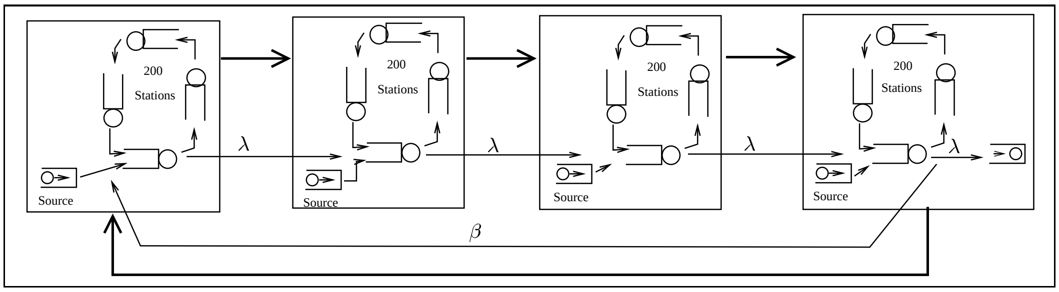


Fig. 14.14 – *Topologie pour le modèle atelier*

Architecture client passif Nous donnons à la table 14.9, le temps d'exécution du modèle simulé avec les deux protocoles conservatifs et pour différentes valeurs de λ et β. Dans quasiment tous les cas de figure, nous obtenons une accélération de la simulation. Les performances sont moins bonnes pour les valeurs de λ grandes et de β petites, car les clients quittent rapidement les sous-systèmes et le ratio calcul/communication diminue. La synchronisation avec messages nuls automatiques est ici la plus performante. Le nombre de messages circulant sur les canaux étant moins important que dans le modèle précédent.

β / λ		0	0.1	0.2	0.3
0	A la demande	296/1.7/0.42			
	Auto	253/2/0.5			
0.1	A la demande	296/1.8/0.45	297/1.7/0.42	287/1.76/0.44	283/1.7/0.42
	Auto	257/2/0.5	254/2/0.5	248/2/0.5	241/2/0.5
0.3	A la demande	279/1.6/0.4		231/1.75/0.44	273/1.4/0.28
	Auto	253/1.75/0.44		200/2/0.5	250/1.53/0.38
0.5	A la demande	236/1.3/0.32	215/1.36/0.34	206/1.34/0.33	200/1.31/0.32
	Auto	211/1.47/0.36	185/1.59/0.4	180/1.54/0.39	174/1.5/0.37
0.7	A la demande	162/1.06/0.26		140/1.05/0.26	133/0.96/0.24
	Auto	199/0.86/0.21		121/1.21/0.3	114/1.13/0.28

Tab. 14.9 – *Simulation distribuée du modèle Atelier (durée en S/accélération/efficacité) – clients passifs*

Architecture client actif La distribution de la simulation selon l'architecture client actif nous donne sensiblement les mêmes performances que pour le cas client passif. Néanmoins, nous avons ici un coût des communications plus élevé (transfert d'un objet actif comparé au transfert d'une requête asynchrone) et donc une plus grande sensibilité au ratio calcul/communication. Ce qui explique la dégradation plus rapide des performances quand les valeurs de λ grandissent et celles de β diminuent.

β / λ		0	0.1	0.2	0.3
0	A la demande	466/1.6/0.4			
	Auto	394/1.9/0.5			
0.1	A la demande	443/1.6/0.4	432/1.6/0.4	432/1.6/0.4	417/1.5/0.37
	Auto	382/1.9/0.5	378/1.9/0.5	374/1.8/0.45	392/1.6/0.4
0.3	A la demande	358/1.5/0.37		364/1.5/0.37	352/1.4/0.35
	Auto	337/1.6/0.4		312/1.7/0.42	336/1.5/0.37
0.5	A la demande	269/1.3/0.32	273/1.3/0.3	280/1.3/0.3	277/1.25/0.3
	Auto	261/1.3/0.32	281/1.3/0.3	283/1.3/0.3	260/1.3/0.3
0.7	A la demande	179/0.9/0.22		182/0.9/0.22	184/0.85/0.21
	Auto	192/0.9/0.22		203/0.8/0.2	194/0.8/0.2

Tab. 14.10 – *Simulation distribuée du modèle Atelier (durée en S/accélération/efficacité) – clients actifs*

14.5.3 Conclusions sur l'évaluation de la librairie distribuée

L'évaluation du framework distribué, dans le cadre de la simulation de files d'attente, donne des résultats très encourageants. Nous obtenons des accélérations significatives aussi bien pour l'architecture client actif que pour l'architecture client passif. De plus, la migration des objets actifs en réduisant les coûts de communication, de synchronisation et en équilibrant la charge s'est avérée être une solution particulièrement performante. Bien entendu, l'accélération n'est obtenue que si le ratio calcul/communication est suffisament important. Mais ce ratio est difficile à déterminer a priori et dépend de nombreux critères (nombre d'objets actifs par cluster, schémas d'interaction, complexité des comportements et des services, etc.). Le développement d'outils permettant d'analyser une simulation séquentielle s'avère donc nécessaire (cf §8.5). Dans le même ordre d'idées, nous devons aussi instrumenter le framework, afin d'être en mesure de déterminer précisément où se situent les goulots d'étranglement.

14.6 Bilan et techniques d'optimisation de simulations

Cette première étude des performances du framework Prosit, nous permet de tirer les conclusions suivantes :

- L'approche orientée processus induit un surcoût à l'exécution. Mais ce surcoût peut être négligé lors de la simulation de grands modèles ou de modèles à forte composante algorithmique. De plus, l'architecture hiérarchique du contrôle rend la simulation de grands modèles particulièrement performante.
- L'utilisation de la réification est aussi un mécanisme coûteux. Là encore, ce coût peut être négligé quand la granularité des services augmente.
- Dans le cadre de la distribution, les résultats obtenus sont très encourageants, tout en restant similaires à ceux d'autres simulateurs. En effet, Prosit ne propose pas de nouvelles techniques de synchronisation, et les algorithmes standards définissent les classes de modèles pour lesquelles il y aura accélération.
  Au niveau fonctionnalité, la demande bloquante d'un service distant s'avère être pénalisante du point de vue des performances, car elle introduit un cycle de dépendance entre les deux clusters. Par contre, la migration d'objets actifs, en réduisant les coûts de communication, de synchronisation et en équilibrant la charge, est une approche particulièrement performante.
  Les exécutions distribuées se sont aussi révélées être très pénalisées par la latence importante des supports de communication. C'est pourquoi nous envisageons d'expérimenter d'autres supports et librairies de communication.

Nous présentons à la table 14.12 une synthèse des performances de Prosit en fonction du nombre d'objets actifs et de la complexité de ces derniers. Nous présentons aussi à la table 14.11 une comparaison multi-critères des architectures client actif et client passif.

	Client actif	Client passif
Réutilisabilité	Importante réutilisation des classes serveurs.	Faible réutilisation des classes serveurs (gestion du routage).
Performances	Bonnes si le rapport : coût du service / coût de demande est élevé. Bonnes performances distribuées grâce à la migration.	Bonnes performances séquentielles. Bonnes performances distribuées car les services sont non bloquants.
Abstraction et facilité de codage	Très importante.	Moyenne en général, mais peut être plus importante si elle est adaptée à l'architecture du modèle.
Dimention des modèles	Un objet actif occupe une place dans l'échéancier et beaucoup de mémoire (4 Ko).	Un client passif occupe peu de mémoire et est toujours stocké dans un objet actif.

Tab. 14.11 – *Comparaison multi-critères des architectures client actif et client passif*

Malgré les bonnes performances du système Prosit, il est souvent nécessaire d'optimiser une simulation afin de réduire les temps d'exécution. La technique classiquement rencontrée, mais non satisfaisante, consiste à simplifier le modèle jusqu'à obtenir les temps d'exécution

Complexité Nombre d'OA	Faible	Moyenne	Importante
Faible < 200	Performances séquentielles faibles. Performances distribuées inexistantes.	Performances séquentielles moyennes. Très faibles performances distribuées.	Bonnes performances séquentielles dues au code compilé. Très faibles performances distribuées.
Moyen $[200, 500]$	Performances séquentielles moyennes. Faibles performances distribuées	Bonnes performances séquentielles et distribuées.	Bonnes performances séquentielles et distribuées.
Important > 500	Bonnes performances séquentielles dues aux structures internes. Performances distribuées moyennes.	Bonnes performances séquentielles et distribuées.	Très bonnes performances séquentielles et distribuées.

Tab. 14.12 – *Performance de* Prosit *en fonction du nombre d'objets actifs et de la complexité de chaque objet*

désirés. Mais grâce à l'architecture framework, il est aussi possible de spécialiser certaines composantes afin de tirer profit d'une spécificité du modèle. Nous avons choisi de présenter ici deux exemples de spécialisation : la surcharge des points de réification et le codage d'automates. Ces deux exemples illustrent bien les idées et les techniques à la base de nos propositions.

14.6.1 Surcharge des points de réification

Une première optimisation possible consiste à limiter le nombre de réactivations des objets actifs. La réactivation d'un objet est par exemple demandée par le point de réification `get_request()` à la réception d'une requête[7], afin que le comportement puisse la traiter dès son arrivée. Or dans certains cas, l'objet réactivé ne peut servir la requête et ne fait que la stocker. D'où l'idée de spécialiser ce point de réification en y intégrant la connaissance de la dynamique de l'objet. La réactivation n'est alors demandée que si l'objet peut servir la requête. Nous illustrons à l'exemple de code 14.3, l'utilisation de cette technique dans le cadre de la programmation d'un serveur doté d'une politique FIFO et sans préemption. Signalons que cette approche serait plus difficile à implémenter si le service autorisait la préemption. Elle nécessite aussi un plus grand soin pour le calcul de statistiques et pour la génération de traces dans la mesure où la date d'arrivée de la requête est antérieure à sa date de prise en compte.

Evaluation du gain Nous simulons un système comportant une source routant des clients passifs vers une station. La source génère 10^5 clients selon une distribution uniforme(0,1). La station possède un unique serveur, et sert directement les clients de manière FIFO, sans préemption, et selon une distribution uniforme(1,2). A la fin du service, les clients sont routés

7. De manière symétrique, la réactivation est aussi demandée par le point de réification `get_reply()` dès réception d'une réponse. La technique présentée dans cette section s'applique donc aussi.

Exemple de code 14.3 Surcharge du point de réification afin de limiter le nombre de réactivations

```
void Server_Normal::get_request(Request *rq) {
  request_in->add(rq);
  switch(status) {
    case RUNNING:
      if(serveur_libre()||(is_express(rq))) {
        noyau->unschedule_obj(this);
        set_time(rq->get_time());
        noyau->schedule_obj(this);
      }
      break;
      ....
  }
}
```

vers une station puits. La surcharge du point de réification permet une exécution plus rapide de 30% . Signalons que cette optimisation est implémentée dans la librairie files d'attente et à donc été utilisée lors des évaluations présentées aux sections 14.4 et 14.5.

14.6.2 Utilisation d'automates

Toujours dans l'objectif de réduire le nombre d'activations, nous avons expérimenté une autre technique d'optimisation qui consiste à mettre en œuvre des objets actifs dépourvus d'activité principale. Cette approche consiste à dériver la classe `Active_Obj` en surchargeant les primitives de gestion de l'ordonnancement (`sleep()`, `wait()`, etc.) et à ne plus créer d'activité principale. L'activation de l'objet consiste alors à exécuter entièrement la méthode `behave()`. Cette technique est entièrement transparente à l'ordonnanceur et permet de faire coexister des objets actifs avec et sans activité principale. Avant d'analyser les gains de cette technique, passons en revue ses limitations.

Le désavantage majeur de cette approche provient du fait que pour tout objet actif ne possédant pas d'activité, le mécanisme de point d'arrêt (cf. §6.4.1 page 49) n'est plus disponible. L'objet doit alors se comporter comme un automate en stockant son état dans ses attributs, l'activation consistant à exécuter les traitements courants et à éventuellement changer d'état. L'objet ne peut pas non plus créer et contrôler d'activités annexes. Le programmeur doit se charger de transformer le comportement utilisant les points d'arrêt en un comportement sous forme d'automate. Selon la dynamique de l'entité modélisée, cette tâche peut aller d'une simple transformation du `behave()` à sa complète re-conception et ré-écriture.

Evaluation du gain Afin d'évaluer le gain de cette approche, nous avons repris la simulation du modèle linéaire avec clients passifs présentée à la section 14.4.2. Mais nous remplaçons ici l'ensemble des objets actifs composant le modèle par des objets automates. La figure 14.15 présente le temps d'exécution des simulations avec objets actifs et avec automates. Nous pouvons voir que l'utilisation des automates apporte un gain d'exécution de l'ordre de 20 à

30%.

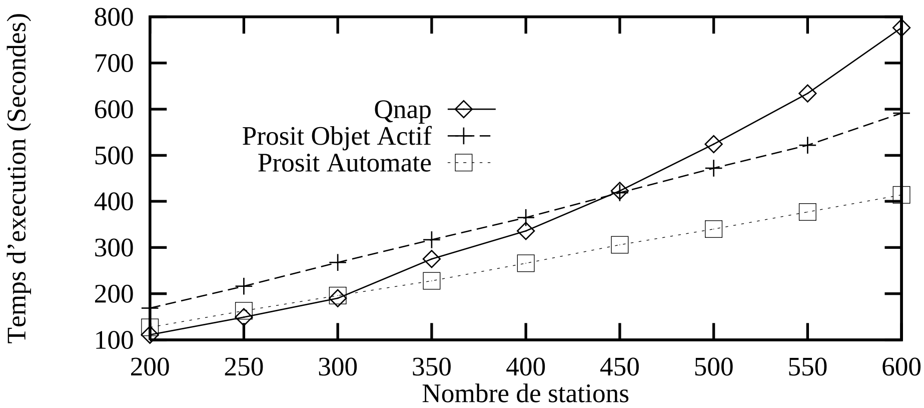


Fig. 14.15 – *Temps de simulation du modèle linéaire avec automates*

A titre indicatif, nous donnons à la table 14.13 le pourcentage du temps d'exécution, groupé par fonctionnalités, pour la simulation avec automate et celle avec objet actif. Cette étude montre que trois composantes pourraient faire l'objet d'une optimisation : la génération de nombres aléatoires, la gestion mémoire et la gestion de l'échéancier.

Fonctionnalité	Automate	Objet actif
Génération de nombres aléatoires	30	22.8
Gestion mémoire	21	21
Gestion de l'échéancier	20	16.5
Mécanisme de réification	13	13.5
Gestion de l'ordonnancement	5	6.5
Comportement des objets	5	6
Calcul du routage	3	2.7
Changement de contexte	0	11
Divers	3	0

Tab. 14.13 – *Répartition fonctionnelle du temps d'exécution (en pourcentage)*

15. EXEMPLES D'APPLICATIONS

Nous présentons dans ce chapitre deux exemples d'applications concrètes de Prosit. Chacune de ces réalisations exploite la flexibilité de l'environnement en spécialisant le framework.

- L'évolution actuelle des besoins en communication nécessite la mise en place de réseaux haut débit. La complexité de ces réseaux, la variété du trafic (sessions interactives, transfert de fichiers, données multimédia, etc.) et le nombre important de paramètres à prendre en compte (qualité du service, débit, etc.) rendent leur étude analytique impossible. C'est pourquoi la mise en place d'environnements de simulation, conviviaux et très performants, est plus que jamais une nécessité. Nous montrons dans la première section comment l'environnement Prosit est utilisé pour atteindre ces objectifs dans le cadre de la simulation de réseaux de commutateurs ATM.
- La simulation de trafic routier est un domaine en pleine expansion qui est motivée par de multiples facteurs : croissance continue du nombre de voitures, congestion quasi permanente des centres urbains, pollution, évaluation des systèmes dynamiques d'information, etc. La simulation intervient à la fois comme outil d'étude, de prévision, de formation et de validation. La direction transport de la communauté européenne a lancé en 1997 un nouveau projet de simulation du trafic routier baptisé Hipertrans, et Prosit a été retenu comme noyau de simulation pour ce projet. Dans la seconde section, après avoir présenté le projet Hipertrans, nous montrerons comment le framework Prosit y est utilisé.

15.1 Simulation mixte de réseaux de commutateurs ATM

Le développement récent de réseaux ATM a motivé la réalisation de nombreux outils de simulation ([Golmie *et al.*, 1995], [Marsan *et al.*, 1995], [Pham, 1996]). Mais tous ces environnements, en adoptant l'approche à événements discrets classique, ne permettent pas de simuler à une vitesse acceptable (de l'ordre de l'heure et pas de la semaine) et au niveau de détail désiré (celui de la cellule) l'interconnexion de plusieurs commutateurs. Afin de remplir ces deux exigences, niveau de détail et vitesse d'exécution, nous avons fait appel aux techniques de la simulation équationnelle (cf. §2.2.3). La convivialité et l'extensibilité de l'outil de simulation sont obtenus en intégrant les modules équationnels à Prosit. Prosit permet aussi de simuler les composantes non-équationnelles et l'intégration est réalisée suivant la technique présentée à la section 9.2.1. Nous présentons dans cette section, le modèle du commutateur ATM, puis la mise en œuvre de la simulation au sein du framework Prosit et enfin les premiers résultats obtenus.

Ce travail a été réalisé en collaboration avec A. Jean-Marie et B. Gaujal. Il fait suite à l'étude de l'intégration de moteurs équationnels dans PROSIT (cf. §9.2.1). Cette application constitue une validation de nos propositions. Le lecteur intéressé peut consulter [Gaujal *et al.*, 1997b, Gaujal *et al.*, 1997c].

15.1.1 Description du modèle

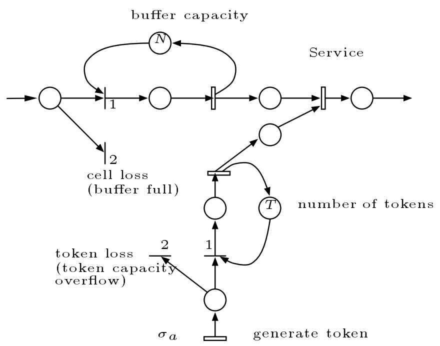
Le commutateur ATM constitue la composante la plus complexe du réseau. Il se comporte comme un système synchrone traitant des paquets de tailles constantes et se prête donc bien à une simulation équationnelle. Par contre, d'autres composantes sont difficiles voire impossibles à modéliser par la technique équationnelle. C'est par exemple le cas des sources de trafic qui sont généralement asynchrones et modélisées par une superposition de flux on-off. Nous avons alors décidé de représenter le commutateur par un module équationnel et les autres composantes par des éléments classiques du modèle files d'attente.

Il n'existe pas d'architecture standardisée pour les commutateurs (voir par exemple [Black, 1995] Chap. 8) et les informations propres à chaque constructeur sont confidentielles. Néanmoins, les commutateurs sont généralement bâtis à l'aide de composants standards (*leaky bucket*, contrôle de flux, etc.). Seuls l'assemblage, le nombre et les caractéristiques de ces éléments de base varient. C'est pourquoi nous avons modélisé le commutateur de façon modulaire, tout en respectant les contraintes de la technique équationnelle. L'assemblage d'un commutateur complet se fait à l'aide d'un éditeur graphique hiérarchique. Nous présentons brièvement les principaux éléments entrant dans la composition du commutateur (cf. Fig. 15.1) :

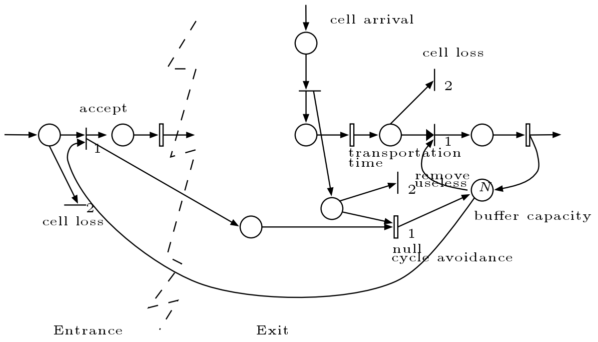
- La figure 15.1.a donne le réseau de Petri correspondant au *leaky bucket*. Les jetons sont générés selon un taux $1/\sigma_a$. Une cellule entrant dans le commutateur quand il n'y a pas de jetons disponibles est bufférisée. Si le buffer est plein, la cellule est perdue.
- La figure 15.1.b donne le réseau de Petri correspondant au mécanisme de contrôle d'admission. La ligne brisée représente l'entrée du commutateur, une cellule ne peut entrer tant que le buffer de sortie est plein.
- La figure 15.1.c donne le réseau de Petri correspondant au mécanisme de contrôle de flux *back pressure*. Deux politiques sont possibles, blocage avant ou après le service.
- La figure 15.1.d illustre le mécanisme de priorité des trafics.
- La figure 15.1.e présente le mécanisme de *time-out*. Une cellule arrivant *time-out* unités de temps après la précédente, est perdue.

15.1.2 Simulation en PROSIT

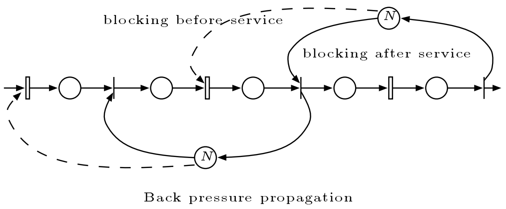
La mise en œuvre de la simulation au sein de l'environnement PROSIT suit la technique présentée à la section 9.2.1. A chaque modèle de commutateur est associé une classe équationnelle. L'ensemble des classes est ensuite regroupé dans une librairie. La génération de la classe équationnelle est un processus entièrement automatisé. A l'aide d'une l'interface graphique, le modélisateur construit son commutateur en utilisant les composants de base. Une fois le modèle achevé et sauvegardé au format PNL, un programme permet de générer les équations



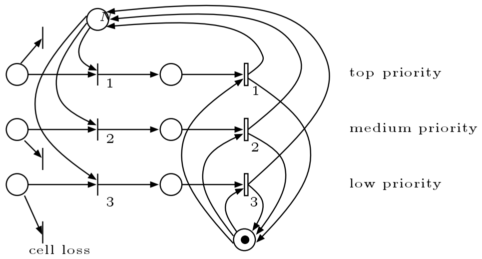
a) Leacky bucket



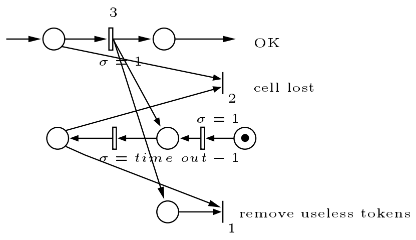
b) Admission control based on the capacity
overflow threshold of output buffers



c) Back pressure mechanism



d) 3 different traffic in a single finite
queue with 3 levels of priority



e) jiter control

Fig. 15.1 – *Composants de base d'un commutateur ATM*

et le code de la classe équationnelle. Le codage de la simulation peut ensuite être manuel, en codant directement le programme C++, ou alors graphique en utilisant l'interface présentée à la section 13.2.3.

15.1.3 Expériences et résultats

L'expérience réalisée consiste à simuler un réseau comportant trois commutateurs (cf. Fig. 15.2). Chaque commutateur possède six ports d'entrées et autant de ports de sortie. Nous avons aussi inclus les mécanismes suivants :

- deux classes de trafic sont superposées (le trafic CBR et le trafic ABR),
- les flux d'entrée sont régulés par un mécanisme de *leaky buckets*,
- l'entrée du commutateur est contrôlée par un mécanisme de *back pressure*,
- les buffers de sortie sont dotés d'un mécanisme de *time out*.

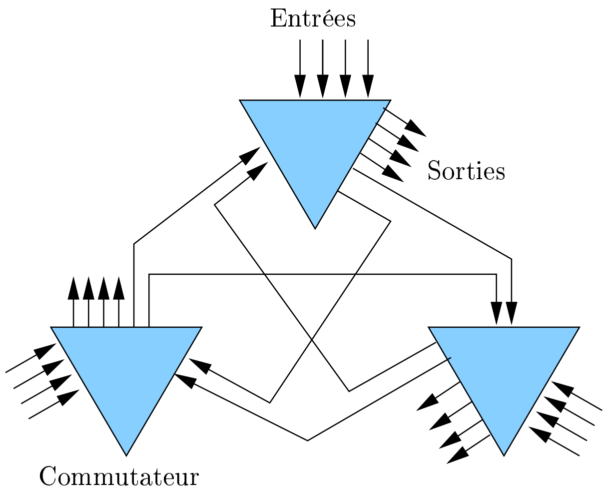


FIG. 15.2 – *Réseau ATM simulé*

Nous avons réalisé trois expériences en faisant varier le taux d'arrivée des cellules. Ces dernières sont générées selon des distributions exponentielles de moyenne 0.5, 1, 10 et 100. On simule 10000 unités de temps.

1. La première expérience consista à mettre le réseau à plat et à le simuler de manière séquentielle en utilisant la librairie de simulation de réseaux de Petri. La mise à plat du réseau correspond à environ 1500 places et transitions. Les résultats sont présentés à la table 15.1.
2. La seconde expérience consista à regrouper l'ensemble des trois commutateurs dans un seul module équationnel et à simuler de manière séquentielle le réseau. L'objectif étant alors de tirer profit au maximum de l'efficacité de la technique équationnelle. Le nombre d'objets actifs étant alors ici de 115. Les résultats sont présentés à la table 15.1.

3. La dernière expérience consista à distribuer le réseau sur 3 clusters. Chaque cluster se chargeant de simuler un commutateur, ses entrées et ses sorties locales. La synchronisation conservatrice avec messages nuls automatiques est utilisée. Chaque cluster gère 38 objets actifs (un cluster en gère 39). Les résultats sont présentés à la table 15.2.

Pour les expériences séquentielles nous avons utilisé la configuration 2 (cf. §14.2) et pour les expériences distribuées la configuration 1 et le réseau Ethernet à 10 Mb (cf. §14.2). Les classes client passif de la librairie files d'attente ainsi que la librairie de simulation de réseaux de Petri sont utilisées.

Arrivée	Simulation mixte	Réseau de Petri
Exp(0.5)	875	12490
Exp(1)	140	9580
Exp(10)	26	4098
Exp(100)	14	599

Tab. 15.1 – *Temps d'exécution pour la simulation séquentielle (Secondes)*

Les résultats donnés à la table 15.1 montrent que l'utilisation de la technique équationnelle réduit les temps d'exécution de un à deux ordres de grandeur. Signalons que la librairie de simulation de réseaux de Petri pourrait faire l'objet d'optimisations importantes. De plus, cette librairie associe à chaque place et à chaque transition un objet actif, ce qui est une approche particulièrement coûteuse. Malgré tout, l'utilisation de la technique équationnelle restera plus performante. S'il est vrai qu'une partie du gain provient de la réduction du nombre d'objets actifs, l'optimisation la plus importante reste l'absence d'échéancier et de traitement séquentiel des événements.

Taux d'arrivée	Exp(100)	Exp(10)	Exp(1)	Exp(0.5)
Simulation séquentielle mixte	14	26	140	875
Simulation distribuée mixte	44	68	125	387
Simulation distribuée mixte avec bufférisation	44	64	97	359

Tab. 15.2 – *Temps d'exécution pour la simulation distribuée (Secondes)*

Les résultats présentés à la table 15.2 correspondent bien aux conclusions de la section 14.5.3 : la distribution n'est valable que si le ratio calcul/communication est important. Ce ratio augmente quand le réseau simulé est chargé (taux d'arrivée important des cellules). Le cas Exp(0.5) donne la meilleure accélération, mais dans cette configuration le système est surchargé et la plupart des cellules sont perdues au niveau des commutateurs. Le cas Exp(0.5) génère le même nombre de messages inter-clusters que le cas Exp(1). La bufférisation améliore ici légèrement l'accélération.

15.2 Simulation de trafic routier

15.2.1 Le projet Hipertrans

Hipertrans est un projet financé par la communauté européenne qui a pour objectif de développer en deux ans un atelier de simulation de trafic routier ayant les caractéristiques suivantes :

- **Modélisation**. Le trafic devra être modélisé au niveau microscopique (niveau du véhicule) et intégrer le comportement des différentes classes de conducteurs. La modélisation devra aussi tenir compte de la signalisation (statique et dynamique), des capteurs et du réseau d'information aux usagers. L'approche sera aussi multimodale car elle considérera plusieurs types de transports (véhicules particuliers, transports publics, etc.) ainsi que les interactions entre eux.
- **Réalisation**. Le simulateur devra être capable de simuler une ville moyenne plus vite que le temps réel. Il pourra ainsi être utilisé comme outil de prédiction au sein des centres de contrôle. En mode de fonctionnement temps réel, il sera couplé avec un système de contrôle du trafic. La simulation en cours sera alors considérée par le contrôleur comme étant la réalité. Afin d'obtenir ces performances, le simulateur pourra faire appel aux techniques d'exécution distribuée.
- **Utilisation**. L'environnement sera utilisé à la fois comme outil d'entraînement, de validation et de prédiction. Il devra posséder une interface graphique conviviale et fournir l'ensemble des fonctionnalités haut de gamme des interfaces hommes-machines (interactivité, graphisme temps réel, etc.).

Nous présentons dans la section suivante l'architecture logicielle en cours de développement au sein du projet Hipertrans. Comme nous le verrons, le framework et l'environnement Prosit y occupent une place centrale. Ce travail est le fruit d'une collaboration avec plusieurs membres du projet (Simulog, Université de Westminster et Université Notre Dame de la Paix) et plus particulièrement avec T. Jiménez en thèse à l'INRIA.

15.2.2 Mise en œuvre d'une librairie pour la simulation de trafic routier

L'architecture retenue pour la première version de l'atelier de simulation (version sans la contrainte du temps réel) est basée sur celle déjà expérimentée et validée pour la librairie Prosit (cf. la section 13.2.3) et est présentée à la figure 15.3. L'interface graphique sera basée sur l'outil Graphmod de la société Simulog et génèrera une description de la simulation dans un format standard. Cette description comprendra à la fois la description du réseau physique, la description de la signalisation et des règles de priorités, celle du trafic ainsi que les paramètres de la simulation. Ces fichiers de configuration pourront ensuite soit être utilisés pour générer le code C++ correspondant à la simulation, soit être directement utilisés par un simulateur Prosit générique capable de construire dynamiquement le modèle et de l'exécuter.

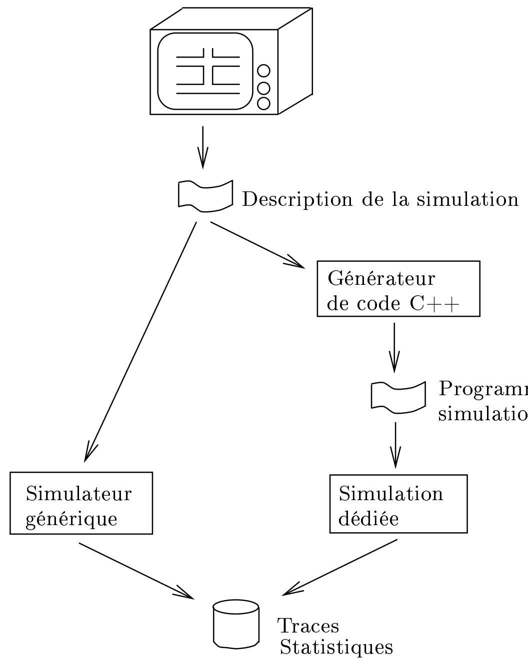


Fig. 15.3 – *Construction d'une simulation dans le cadre d'*Hipertrans

Cette architecture nécessite la conception et la réalisation d'une librairie Prosit dédiée à la simulation de trafic routier. Mais, pour des raisons de performances, la librairie sera codée directement au-dessus de la couche objet actif. Nous aurions pu construire une modélisation en utilisant les files d'attente ou les réseaux de Petri, mais au prix d'une multiplication du nombre d'objets actifs et d'une plus grande difficulté à optimiser le code. Nous aurons une classe d'objet actif par élément du modèle (classe segment de route, carrefour, et.). De plus, afin d'avoir un système le plus ouvert et évolutif possible, nous avons décidé de mettre en place deux niveaux de librairie (cf. Fig. 15.4). Le premier niveau constituera une abstraction des classes de base de Prosit et sera lui-même un framework dédié à la simulation de trafic routier. Le second niveau utilisera ce framework pour mettre en œuvre une modélisation spécifique du trafic routier. Dans le projet, cette modélisation suivra le modèle PACSIM [Cornelis, 1997, Toint, 1994].

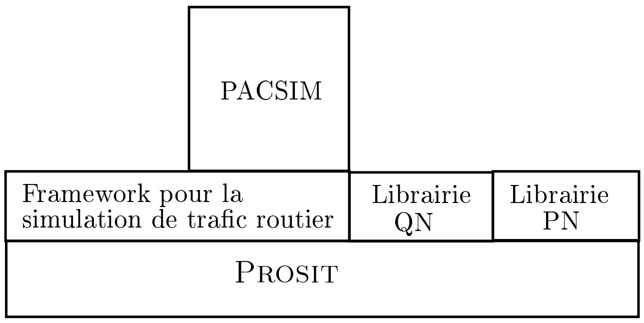


Fig. 15.4 – *Intégration des librairies de simulation de trafic dans le framework* Prosit

Cette architecture et plus particulièrement le framework de base pour la simulation du trafic routier font l'objet des recherches de T. Jiménez [Jiménez, 1997].

Quatrième partie

CONCLUSIONS ET PERSPECTIVES

16. CONCLUSIONS ET PERSPECTIVES

16.1 Contributions et travaux réalisés

Nous avons proposé dans cette thèse, un modèle de conception et de programmation pour la mise en œuvre de simulateurs à événements discrets. Nos contributions comprennent aussi la réalisation d'un support d'exécution, portable et efficace, pour l'expérimentation de ce modèle. Ce support a également été enrichi de librairies de modélisation et d'outils d'aide à la programmation. Le modèle de programmation et son environnement de développement forment une boîte à outils, appelée Prosit, permettant la réalisation de simulateurs séquentiels et distribués.

16.1.1 Modèle de conception et de programmation

Dans le modèle proposé, une simulation correspond à l'interaction dans le temps d'un ensemble d'objets actifs. Le temps est géré de manière automatique et les objets possèdent diverses primitives de manipulation (accès à la date courante, attente d'un laps de temps, etc.). Les objets actifs interagissent par le biais de demandes de service et le modèle distingue un mode bloquant et un mode non-bloquant. Le programmeur dispose d'un double niveau de paramètrage, il peut spécifier les paramètres du service et ceux associés au contrôle de l'exécution du service. Au cours de la simulation, chaque objet actif exécute de manière autonome son comportement qui est spécifié, par le programmeur, dans la routine `behave()`. Nous avons retenu un contrôle explicite et centralisé. Le comportement de l'objet comprend la gestion des requêtes et des services, ainsi que les interactions avec les autres objets actifs. Un objet actif peut aussi créer et contrôler des activités annexes. Ces dernières se déroulent de manière concurrente dans le temps de simulation et permettent par exemple d'exécuter les services.

La manipulation des demandes de service ainsi que le contrôle explicite et centralisé permettent le développement de classes de comportement réutilisables par héritage. La modélisation d'un système puis la construction de sa simulation à l'aide du mécanisme d'objet actif s'est avéré être une approche à la fois simple et directe pour l'utilisateur, tout en restant très puissante et générique. Nous avons aussi montré dans cette thèse, comment les abstractions introduites permettaient de construire très simplement des simulations hiérarchiques et hétérogènes.

Dans le cadre d'une simulation distribuée, les objets actifs composant le modèle sont répartis sur un ensemble de machines. Pour cela, le modèle de programmation offre une vision structurée et abstraite du parc en introduisant la notion de *computer* et de *cluster*. Un computer représente une machine logique et le cluster est l'entité abstraite que manipule le programmeur pour désigner un contexte (mémoire, CPU, etc.). Cette architecture permet de contrôler et de spécifier, au moment de l'exécution, le nombre et la nature des machines physiques utilisées. Pour la distribution de son modèle, le programmeur dispose de mécanismes de placement et de migration des objets actifs, ainsi que de primitives de localisation. La synchronisation entre clusters est réalisée par les ordonnanceurs d'objets actifs et comporte un niveau intra-cluster et un niveau inter-clusters. Les objets actifs participent à la synchronisation en fournissant des informations à l'ordonnanceur et en exécutant les actions demandées par ce dernier.

La distribution d'une simulation se fait sans modification de la conception et avec un minimum de modifications du code. L'architecture adoptée permet d'intégrer la programmation séquentielle et la programmation distribuée au sein d'une même plate-forme de développement.

16.1.2 Support d'exécution

Le support d'exécution développé se présente sous la forme d'une boîte à outils, un framework, permettant la simulation de toute classe de systèmes à événements discrets. Cet environnement possède plusieurs niveaux d'utilisation et chaque composante peut être spécialisée ou redéfinie. Les protocoles conservateurs avec messages nuls automatiques et à la demande sont déjà supportés. L'architecture framework permettra d'incorporer d'autres protocoles et Prosit pourra alors être utilisé comme un *banc d'essai*.

La mise en œuvre du modèle, utilise des techniques réflexives. Une demande de service est automatiquement réifiée en un objet requête, la terminaison d'un service correspond aussi à la construction et à l'émission d'un objet réponse. Le mécanisme mis en place comporte une modélisation des requêtes et de leurs services, une liste des requêtes accessible par le programmeur et une modélisation des routines. Nous avons montré, au travers du framework et des nombreux exemples développés tout au long de cette thèse, l'intérêt de la réification dans le cadre de la simulation à événements discrets. Cette technique, associée au concept d'objet actif, facilite le développement de simulations complexes, favorise le codage et la réutilisation de classes de comportement, et enfin, permet d'intégrer les modèles de programmation séquentielle et distribuée.

16.1.3 Librairies de modèles et outils d'aide à la programmation

L 'objectif de notre travail consistant aussi à fournir un environnement de développement le plus complet possible, nous avons conçu et participé aux développements de diverses librairies de modélisation et d'outils d'aide à la programmation.

Bibliothèques de modélisation Les classes de modélisation étendent les classes de base en définissant un paradigme de modélisation. Elles constituent un élément essentiel d'un environnement généraliste. Nous avons développé une librairie de modélisation pour la simulation de réseaux de files d'attente et nous avons collaboré au développement d'une librairie pour la simulation de réseaux de Petri.

Outils d'analyse et de suivi Les traitements statistiques sont un objectif majeur, voire l'objectif principal, de nombreuses simulations. C'est pourquoi, nous avons participé au développement puis intégré à PROSIT un outil statistique extensible et réutilisable. Nous avons aussi conçut et intégré à PROSIT des fonctionnalités de traces.

Interfaces graphiques et générateur de code Nous avons adapté ou développé puis intégré à l'environnement PROSIT divers logiciels permettant l'édition graphique de modèles, l'animation du déroulement d'une simulation et le suivi de l'exécution. Ces outils facilitent la mise en œuvre de simulations, notamment pour les utilisateurs ne sachant pas programmer. Ils constituent aussi une plate-forme de formation et de démonstration.

16.2 Perspectives

Les travaux entrepris dans le cadre de cette thèse avaient pour objectifs de définir et de réaliser un environnement de programmation de simulateurs, séquentiels et distribués, à événements discrets. Nos contributions, concrétisées par l'environnement PROSIT, ouvrent trois axes de recherche et de développement : la diffusion de nos propositions, le développement d'outils et de techniques d'aide à la distribution de simulations, et enfin, la poursuite de la recherche et de l'expérimentation dans les domaines de la simulation hiérarchique et de la simulation mixte.

16.2.1 Diffusion de nos propositions

Le transfert technologique des résultats de recherche vers l'industrie constitue un objectif majeur d'une thèse [1] et l'aboutissement [2] valorisant de trois années de travail. Nous avons évoqué précédemment notre collaboration avec la société Simulog. Cette collaboration a donné lieu à la mise à disposition d'une version prototype de PROSIT pour évaluation. Cette évaluation est actuellement en cours et les premiers résultats sont très positifs. Nous espérons qu'elle aboutira bientôt sous la forme d'un contrat entre l'INRIA et la société Simulog.

Mais l'industrialisation de nos propositions ne constitue pas l'unique objectif. La mise à disposition, à plus large échelle, du framework à des fins de recherches et de formation est aussi particulièrement important et répondrait à de nombreuses sollicitations. De même, la spécialisation du framework à d'autres domaines, comme c'est le cas avec Hipertrans, est aussi un objectif.

16.2.2 Outils et techniques d'aide à la distribution

Nous avons été confrontés, lors de nos expériences, à la grande difficulté de l'analyse des performances d'une simulation distribuée et à celle de la sélection de l'algorithme de synchronisation. Si ces problèmes sont connus depuis longtemps, ils prennent une plus grande importance dans le cadre de l'environnement PROSIT. En effet, l'abstraction des mécanismes de simulation et la réutilisation à grande échelle de code et de sous-modèles privent l'utilisateur

1. Cet objectif est d'avantage mis en avant dans le cadre d'une thèse au sein de l'INRIA.
2. La phase d'industrialisation peut aussi constituer le début d'une nouvelle expérience si le thésard y participe de manière active.

d'une connaissance fine du comportement dynamique de son modèle. De plus, la diffusion des techniques de la simulation passe par une utilisation efficace de l'environnement par des non spécialistes. C'est pourquoi, nous envisageons deux axes de développement dans un avenir proche :

– Le premier axe consiste à concevoir puis à intégrer à l'environnement Prosit, un ensemble d'outils d'analyse de la simulation séquentielle. Ces outils pourront exploiter soit une trace d'exécution, soit être directement intégrés au noyau séquentiel. Ils devront être en mesure d'effectuer une analyse du parallélisme intrinsèque d'un modèle et de proposer l'algorithme de synchronisation le mieux adapté ainsi qu'une évaluation multi-critères (occupation mémoire, nombre de messages, etc.) de son utilisation.

– La seconde approche, complémentaire de la précédente, consiste à instrumenter le framework distribué. Cette instrumentation est nécessaire et permettra de comprendre et de détailler les différents coûts liés à la distribution de la simulation.

16.2.3 Simulation mixte et simulation hiérarchique

Nous avons présenté et motivé au chapitre 9 les techniques de simulation mixte et de simulation hiérarchique. Nous avons aussi montré que le modèle de programmation Prosit supporte naturellement ces deux approches. Mais ce travail de recherche et d'expérimentation n'en est qu'à ses débuts.

Nous avons déjà acquis une première expérience dans le domaine de la réalisation de simulations hétérogènes. Mais la revendication de la capacité de Prosit à intégrer des moteurs de simulations hétérogènes, reste à compléter par d'autres réalisations. Une autre approche non encore exploitée est celle qui consiste non plus à intégrer dans Prosit un nouveau moteur de simulation, mais au contraire, à intégrer Prosit au sein d'un autre environnement de simulation. Ptolemy constituerait pour cela un premier candidat.

Dans le cadre de la simulation hiérarchique, nous avons vérifié que Prosit supportait de manière naturelle cette technique. Il nous reste encore à la valider par des simulations de taille réelle. Nous devons aussi nous intéresser à de nombreux problèmes annexes, mais très importants, comme la validation des modèles, la mise au point, etc.

ANNEXE

A. LISTE DES PRINCIPALES RÉFÉRENCES PORTANT SUR PROSIT ET SON ENVIRONNEMENT

Rapports de recherche

P. Mussi and G. Siegel. Sequential Simulation in PROSIT: Programming Model and Implementation. RR-2713, INRIA, Novembre 1995.

Rapports de stages

T. de Pretto. Conception et Implémentation d'un outil réalisant des traitements statistiques pour simulateurs parallèles. Rapport de fin de stage. DESS-ISI, Université de Nice–Sophia Antipolis. 1995.

D. Finzi. Conception et Implémentation d'une Bibliothèque pour la Simulation de Réseaux de Petri Temporisés. Rapport de fin de stage IUP Génie Mathématique et Informatique Dauphine, Septembre 1996.

F. Gioanni. Mécanisme de Trace dans Prosit. Rapport du projet de fin d'étude ESSI, Avril 1996.

T. Jiménez. Simulation de trafic routier : Etude des méthodologies de modélisation pour la micro-simulation et mise en œuvre, Rapport de DEA, Université de Nice–Sophia Antipolis, ESSI, 1997.

Articles

M. Badel, T. de Pretto, P. Mussi, and G. Siegel. Stat-Tool: An Extensible and Distributed Object Oriented Statistic Tool for Discrete Event Simulation. Dans *Object Oriented Simulation Conference*, 1996. Présentation orale, à paraître dans les proceedings de 1997.

P. Mussi and G. Siegel. The PROSIT Sequential Simulator: A Test-bed for Object Oriented Discrete Event Simulation. Dans *European Simulation Symposium*, pages 297–301, Erlangen, Germany, Octobre 1995.

P. Mussi and G. Siegel. Extending the Prosit System to Support Distributed Simulation. Dans *European Simulation Symposium*, Genoa–Italy, Octobre 1996.

B. Gaujal and A. Jean-Marie and P. Mussi and G. Siegel, High Speed Simulation of Discrete Event Systems by Mixing Process Oriented and Equational Approaches, Parallel Computing – special issue: environment and tools for parallel scientific computing, P. 219–234, Vol. 23(1–2), Avril 1997,

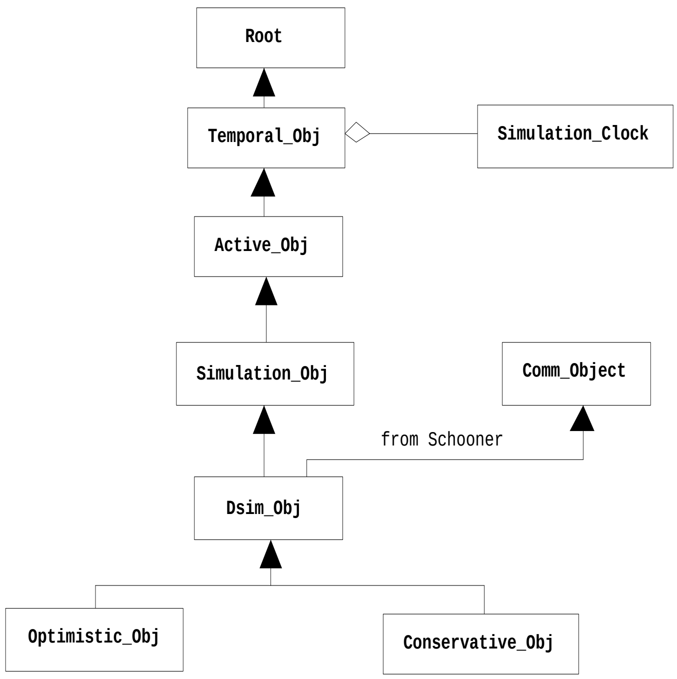
Documents électroniques

Site Prosit : http://www.inria.fr/sloop/prosit

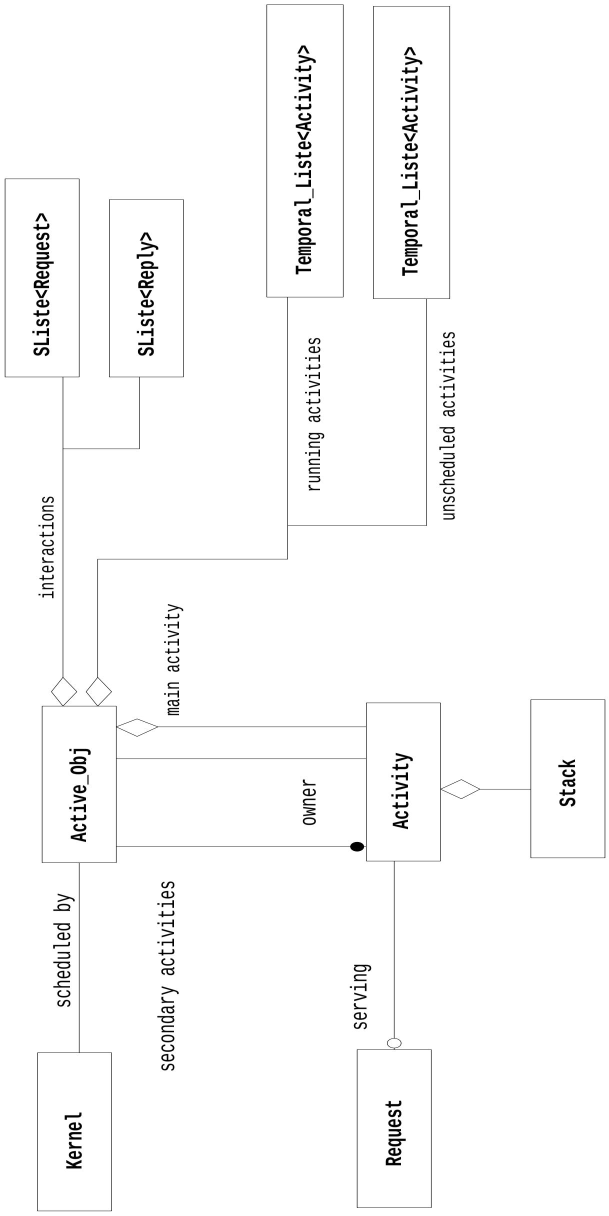
Site Schooner : http://www.inria.fr/sloop/schooner

B. LE MODÈLE OBJET DU FRAMEWORK PROSIT

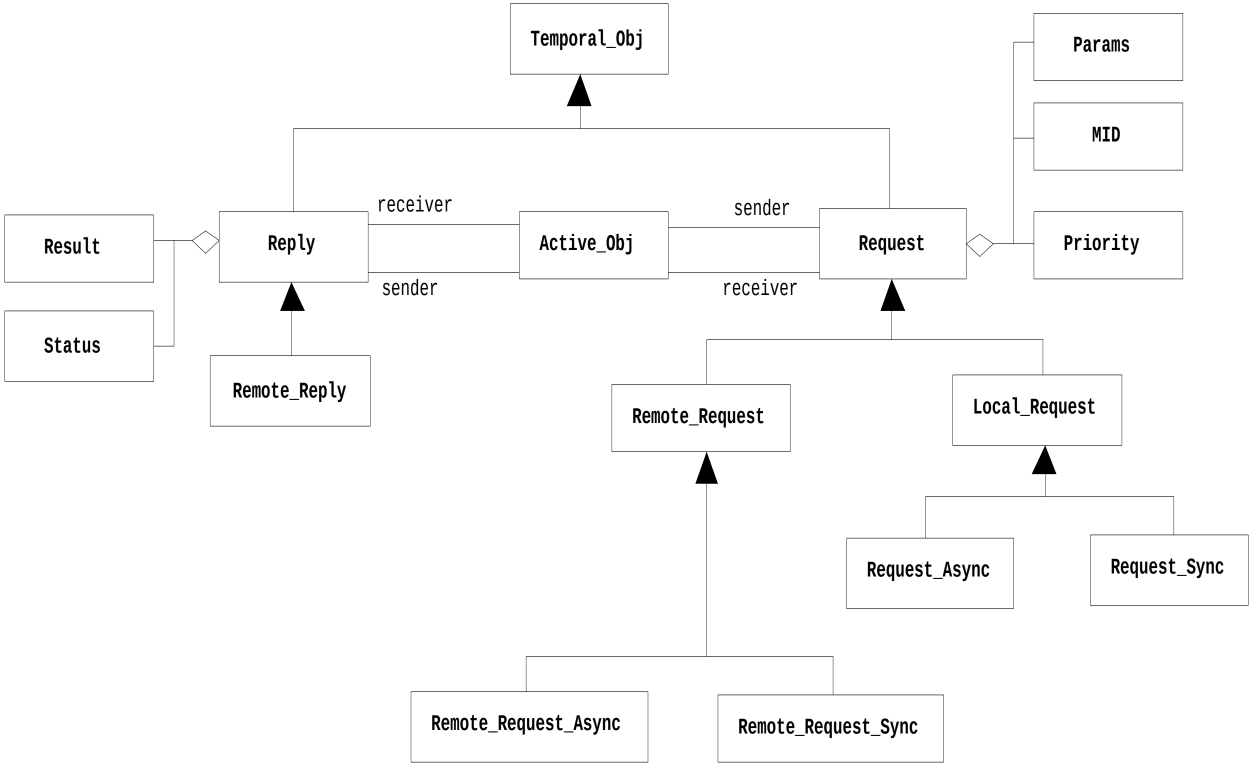
Hiérarchie de base des classes actives



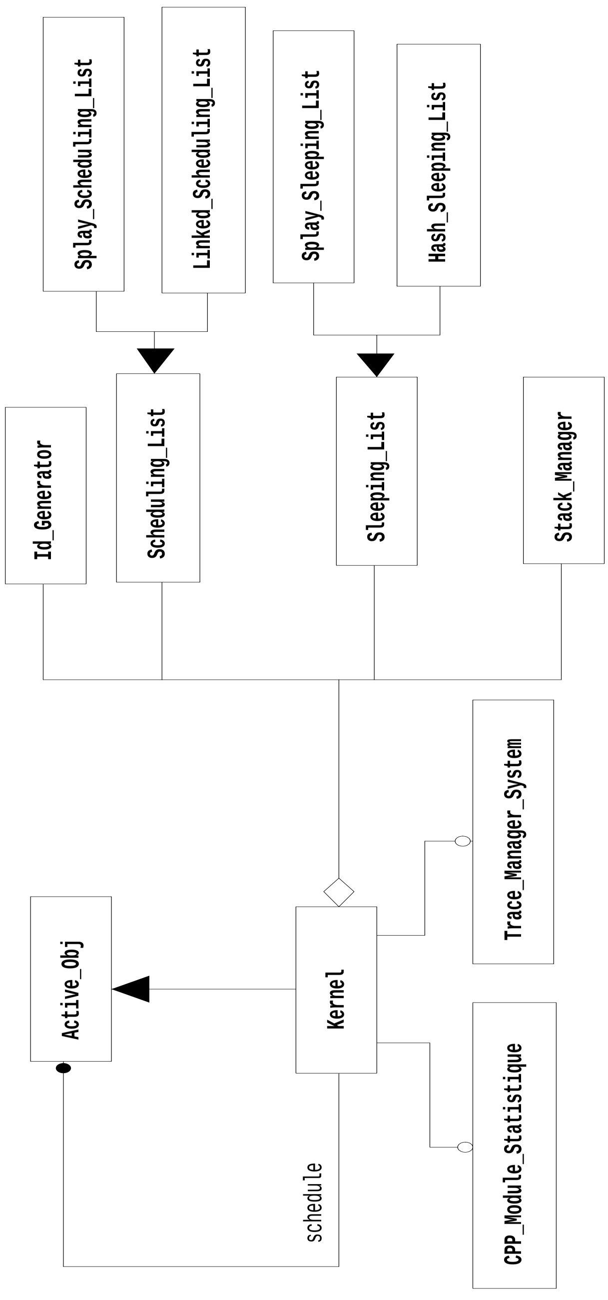
Modèle objet de la classe `Active_Obj`



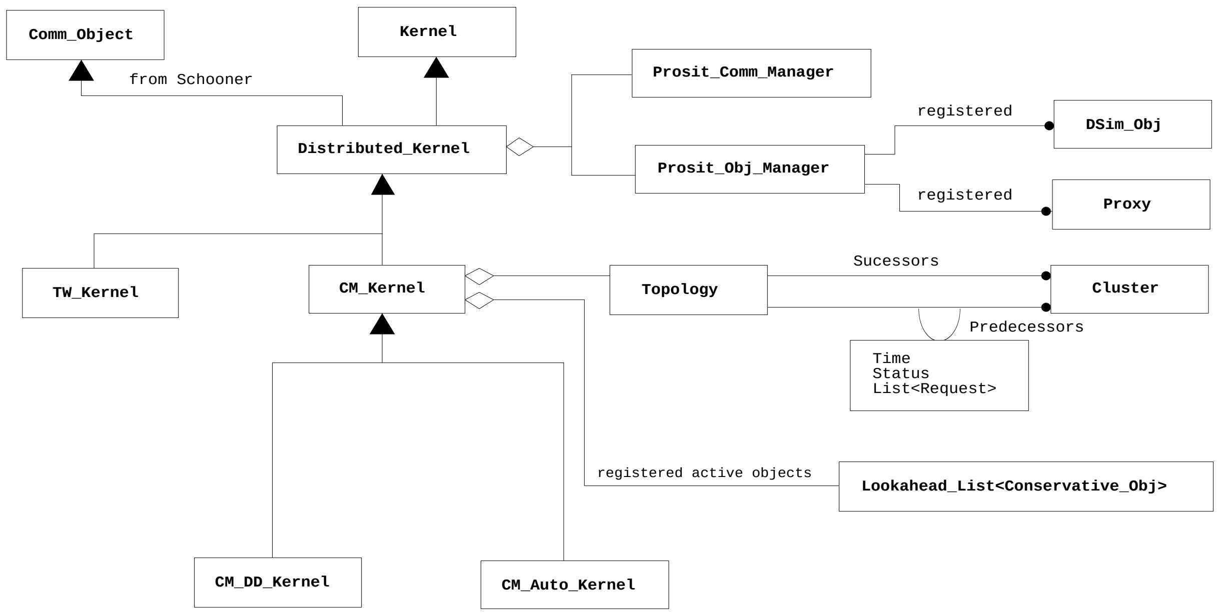
Modèle objet du mécanisme d'interaction



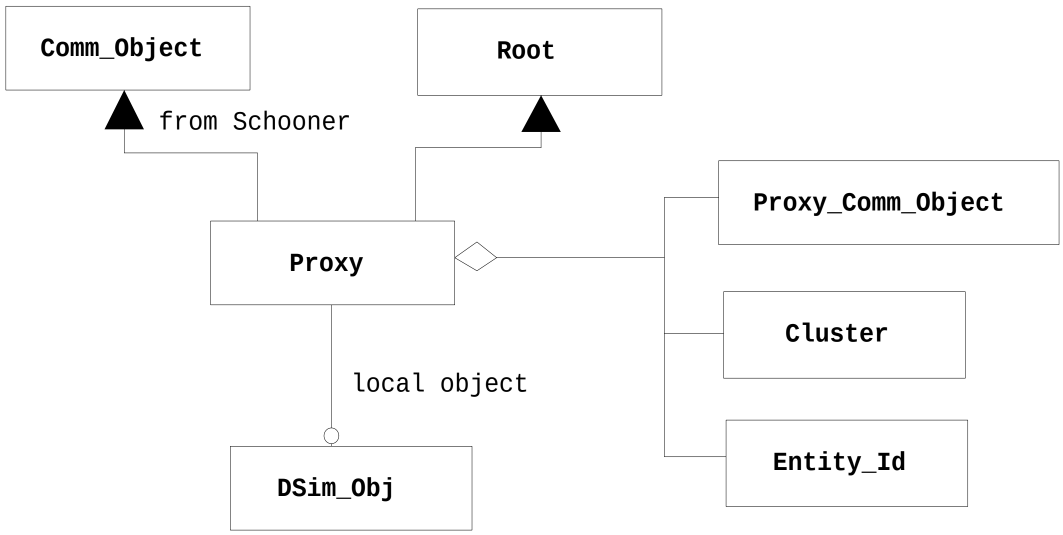
Modèle objet du noyau séquentiel



Modèle objet du noyau distribué



Modèle objet du mécanisme de proxy



INDEX

BIBLIOGRAPHIE

[Agha et Hewitt, 1987] G. Agha et C. Hewitt. *Concurrent Programming Using Actors*. MIT PRESS, 1987.

[Altmann *et al.*, 1995] M. Altmann, B. Wee, M. Ong, et C. Ong. *SIMEX Reference Manual*. University of Minesota – Academic Health Center, Janvier 1995. `http://www.nmsr.labmed.umn.edu` .

[Arsac *et al.*, 1993a] O. Arsac, S. Dalmas, et M. Gaëtano. Algorithm Animation with AGAT. INRIA – Projet Safir `http://www.inria.fr/safir/SAM/Agat/agat.html`, 1993.

[Arsac *et al.*, 1993b] O. Arsac, S. Dalmas, M. Gaëtano, et S. Lavirotte. *The AGAT manual*. INRIA - Projet Safir `http://www.inria.fr/safir/SAM/Agat/agat.html`, 1993.

[Aydt, 1992] R. Aydt. The Pablo Self-Defining Data Format. University of Illinois Urbana, Mars 1992. Revised April 11, 1995. `http://www-pablo.cs.uiuc.edu/` .

[Baccelli *et al.*, 1992] F. Baccelli, G. Cohen, G. Olsder, et J. Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[Baccelli *et al.*, 1996] F. Baccelli, S. Foss, et B. Gaujal. Structural, temporal and stochastic properties of free-choice Petri nets. *IEEE Trans. Automatic Control*, 41(12), Décembre 1996.

[Baccelli et Canales, 1993] F. Baccelli et M. Canales. Parallel Simulation of Stochastic Petri Nets using Recurrence Equations. *ACM Trans. on Modeling and Computer Simulation*, 3(1):20–41, Janvier 1993.

[Badel *et al.*, 1996] M. Badel, T. de Pretto, P. Mussi, et G. Siegel. Stat-Tool: An Extensible and Distributed Object Oriented Statistic Tool for Discrete Event Simulation. In *Object Oriented Simulation Conference*, 1996. Oral Presentation, will appear in the 1997 proceedings.

[Badel et Toulzat, 1997] M. Badel et F. Toulzat. PROSIT–Evaluation report V1.0. Technical Report RAP.SQ.97.INF.0009.V1.0/FT.FT, Simulog, Janvier 1997. Simulog, 1, Rue James Joule, 78286 Guyancourt, France.

[Baezner *et al.*, 1990] D. Baezner, G. Lomow, et W. Unger. Sim++: The Transition to Distributed Simulation. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 022 Nr 2, pages 211–218, San Diego CA, 1990.

[Baezner et Lomow, 1989] D. Baezner et E. Lomow. A Tutorial Introduction to Object-Oriented Simulation and Sim++. In *Winter Simulation Conference*, pages 140–146, Washington, DC, Décembre 1989.

[Bagrodia et Liao, 1990] R. Bagrodia et W. Liao. MAISIE: A Language and Opimizing Environment for Distributed Simulation. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 22(2), pages 205–210, San Diego CA, Janvier 1990.

[Bain et Scott, 1988] W. Bain et D. Scott. An Algorithm for Time Synchronization in Distributed Discrete Event Simulation. In B. Unger et D. Jefferson, editors, *SCS Multiconference on Distributed Simulation*, volume 19(3), pages 30–33, San Diego California, Février 1988.

[Baldwin *et al.*, 1991] R. Baldwin, M. Chung, et Y. Chung. Overlapping Window Algorithm for Computing GVT in Time Warp. In IEEE Computer Society Press, editor, *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 534–541, Mai 1991.

[Ball et Hoyt, 1990] D. Ball et S. Hoyt. The Adaptative Time Warp Concurrency Control Algorithm. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 22(2), pages 174–177, San Diego CA, Janvier 1990.

[Baude *et al.*, 1996] F. Baude, F. Belloncle, D. Caromel, N. Furmento, P. Mussi, Y. Roudier, et G. Siegel. Parallel Object-Oriented Programming for Parallel Simulations. *Information Sciences, special issue on Parallel Object-Oriented Programming–Elsevier*, 1996.

[Bauer et Sporrer, 1992] H. Bauer et C. Sporrer. Distributed Logic Simulation and an Approach to Asyncronous GVT-Circulation. In *SCS western Simulation Multiconference on Parallel and Distributed Simulation*, pages 205–209, Janvier 1992.

[Bauer et Sporrer, 1993] H. Bauer et C. Sporrer. Reducing Rollback Overhead in Time Warp Based Distributed Simulation with Optimized Incremental State Saving. In *26th Annual Simulation Symposium*, pages 12–20, Arlington, Virginia, Mars 1993. SCS.

[Beaumariage *et al.*, 1991] T. Beaumariage, A. Chell, et A. Roberts. Object Oriented Modeling: Attempts at Improving Model Execution Speed. In R. Edge, editor, *Object Oriented Simulation Conference*, volume 23(3), pages 93–99, Anaheim, California, Janvier 1991.

[Beaumariage et Mize, 1990] T. Beaumariage et J. Mize. Object Oriented Modeling Concepts and Ongoing Research. In A. Guasch, editor, *Object Oriented Simulation Conference*, pages 7–12, San Diego, California, Janvier 1990. SCS.

[Beguelin *et al.*, 1994] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, et V. Sunderam. A Users' Guide to PVM Parallel Virtual Machine (version 3). Technical report, Oak Ridge National Laboratory, Mai 1994.

[Bellenot, 1990] S. Bellenot. Global Virtual Time Algorithms. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 22(2), pages 122–127, San Diego CA, Janvier 1990.

[Bellonot et Duty, 1995] S. Bellonot et L. Duty. XTracker, A Graphical Tool for Parallel Simulations. In *Parallel and Distributed Simulation*, pages 191–194, Lake placid, New-York, Juin 1995. IEEE-Catalog.

[Berry et Jefferson, 1985] O. Berry et D. Jefferson. Critical Path Analysis of Distributed Simulation. *Simulation Series, Publ by Soc for Computer Simulation*, 15:pages 57–60, 1985.

[Bezivin, 1987] J. Bezivin. Some Experiments In Object-Oriented Simulation. In N. Meyrowitz, editor, *OOPSLA*, volume 22(12), pages 394–405, Orlando, Florida, Octobre 1987.

[Bezivin, 1992] J. Bezivin. Simulation par Objets. Cours sur la Simulation par Objets, Université de Nantes, 1992.

[Birrel, 1989] A. Birrel. An Introduction to Programming with Threads. Technical report, Digital, Systems research Center, Janvier 1989.

[Black *et al.*, 1988] A. Black, N. Hutchinson, H. Levy, et E.Jul. Fine-grained Mobility in the Emerald System. In *ACM transaction on computer system*, volume 6-1, pages 109–33, Février 1988.

[Black, 1995] U. Black. *ATM: Foundation for Broadband Networks*. Prentice hall series, 1995.

[Blair et Selvaraj, 1989] E. Blair et S. Selvaraj. DISC++: a C++ Based Library for Object Oriented Simulation. In *Winter simulation Conference*, pages 301–307, Washington, DC, Décembre 1989.

[Booch, 1991] G. Booch. *Object-Oriented Design with Applications*. Addison-Wesley, 1991.

[Boussinot *et al.*, 1995] F. Boussinot, G. Doumec, et J. Stefani. Reactive Objects. RR 2664, INRIA, Octobre 1995.

[Boussinot et Doumec, 1992] F. Boussinot et G. Doumec. *RC Reference Manual*. Ecole des Mines– Centre de Mathématiques Appliquées, B.P. 207, 06904 Sophia Antipolis CDX, France, 1992.

[Boussinot, 1992] F. Boussinot. La Programmation Réactive et le Langage Réactif C. Technical report, Ecole des Mines, Centre de Mathématiques Appliqués, B.P. 207, 06904 Sophia Antipolis CDX, France, 1992.

[Boussinot, 1996] F. Boussinot. *La programmation réactive, Application aux systèmes communicants*. Collection technique et scientifique des télécommunications, 1996.

[Brasileiro *et al.*, 1993] M. Brasileiro, A. Moura, I. Cabral, et M. Dias. SIMILE : An Object Oriented Simulator for Local Area Network. In A. Pave, editor, *European Simulation Multiconference – Modeling and Simulation*, Lyon–France, Juin 1993.

[Briner, 1991] J. Briner. Parallel Mixed-level Simulation using Virtual Time. *CAD accelerators*, pages 273–285, 1991.

[Briscoe *et al.*, 1988] D. Briscoe, L. Sokol, et A. Wieland. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. In B. Unger et D. Jefferson, editors, *SCS Multiconference on Distributed Simulation*, volume 19(3), pages 34–42, San Diego California, Février 1988.

[Brown, 1988] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communication of the ACM*, 31(10):1220–1227, Octobre 1988.

[Browne et Chandak, 1983] J. Browne et A. Chandak. Vectorization of Discrete Events Simulation. In *International Conference on Parallel Processing*, pages 359–361, Août 1983.

[Bryant, 1977] R. Bryant. Simulation of Packet Communication Architecture Computer Systems. Technical Report MIT-LCS-TR-188, MIT, 1977.

[Buck *et al.*, 1992] J. Buck, S. Ha, A. Lee, et D. Messerschmidt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. Technical report, University of California Berkeley, Août 1992.

[Buhr et Stroobosscher, 1991] P. Buhr et R. Stroobosscher. uc++ annotated reference manual. Technical Report CS-91-53, University of Waterloo, Octobre 1991.

[Cahill *et al.*, 1993] V. Cahill, S. Baker, C. Horn, et G. Starovic. Amadeus Project. In *OOPSLA*, pages 144–161, 1993.

[Cai et Turner, 1990] Cai et Turner. An Algorithm for Distributed Discrete Event Simulation - the *Carrier Null Message* Approach. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 22(1), pages 3–8, San Diego CA, Janvier 1990.

[Caromel *et al.*, 1996] D. Caromel, F. Belloncle, et Y. Roudier. *Parallel Programming Using C++*, chapter The C++// System. G. Wilson ed., MIT Press, 1996. To appear.

[Caromel et *al.*, 1997] Caromel et *al.* EUROPA Parallel C++, V2.1. Europa working group on parallel C++, architecture SIG, Juin 1997.

[Caromel, 1991] D. Caromel. *Programmation Parallèle Asynchrone et Impératives Etude et Propositions (une extension parallèle du langage EIFFEL)*. PhD thesis, université de Nancy I centre de recherche en informatique de Nancy (INRIA Lorraine), 1991.

[Chandy *et al.*, 1983] K. Chandy, J. Misra, et L. Haas. Distributed Deadlock Detection. *ACM Transactions On Computer Systems*, 1(2):144–156, Mai 1983.

[Chandy et Misra, 1979] K. Chandy et J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. In *IEEE Trans. on Software Engineering*, volume SE5(5), pages 440–452, Septembre 1979.

[Chandy et Misra, 1981] K. Chandy et J. Misra. Asynchronous Distributed Simulation Via a Sequence of Parallel Computations. In *Communications of the ACM*, volume 24-1, pages 198–206, Avril 1981.

[Chandy et Sherman, 1989] K. Chandy et R. Sherman. Space-Time and Simulation. In B Unger et R. Fujimoto, editors, *SCS Multiconference on Distributed Simulation*, volume 21(2) of *Simulation series*, pages 53–57, Tampa, Florida, Mars 1989.

[Chorus, 1989] Chorus. CHORUS Distributed Operating Systems. Technical Report CS/TR-88-7.8, Chorus système, Février 1989.

[Chow et Zheng, 1993] P. Chow et Q. Zheng. EXsim: A General Purpose Object Oriented Environment for Discrete Event Simulations. In D. Jefferson R. Bagrodia, editor, *Parallel and Distributed Simulation*, San Diego, California, Mai 1993.

[Chung *et al.*, 1993] K. Chung, J. Sang, et V. Rego. A Performance Comparison of Event Calendar Algorithms: an Empirical Approach. *Software Practice and Experience*, 23(10):1108–1138, Octobre 1993.

[Cochran *et al.*, 1994] J. Cochran, G. Mackulak, et P. Savory. Ascertaining Important Features for Industrial Simulation Environments. *Simulation*, pages 211–221, Octobre 1994.

[Cohen *et al.*, 1985] G. Cohen, P. Moller, J. Quadrat, et M. Viot. Une Théorie Linéaire des Systèmes à Evénements Discrets. Technical Report 362, INRIA, Janvier 1985.

[Colnet *et al.*, 1989] D. Colnet, D. Leonard, K. Tombre, A. Napoli, et G. Masni. *Les langages à objets*. InterEditions, Paris, 1989.

[Comfort et Gopal, 1988] J. Comfort et R Gopal. Environment Partitioned Distributed Simulation with Transputers. In B. Unger et D. Jefferson, editors, *SCS Multiconference on Distributed Simulation*, volume 19(3), pages 103–108, San Diego California, Février 1988.

[Communication of the ACM, 1994] Communication of the ACM, editor. *Special issue on Object Oriented Software Testing*, volume 37(9). ACM, Septembre 1994.

[Concepcion et Kelly, 1991] A. Concepcion et S. Kelly. Computing Global Virtual Time Using the Multi-Level Token Passing Algorithm. In *SCS Multiconference on Distributed Simulation*, volume 23(1), pages 63–68. SCS, Janvier 1991.

[COO, 1991] Cool-2, an object oriented support platform build above the chorus microkernel. In *International Workshop on O.O. O.S.*, pages 68–72, 1991.

[Cornelis, 1997] E. Cornelis. PACSIM users' manual. Faculté Universitaire Notre-Dame de la Paix, Namur, Belgique – Transportation Research Group, Avril 1997.

[Courtney, 1995] A. Courtney. Phantom, A Language Overview. Technical report, Trinity College Dublin, Ireland, Mai 1995.

[Cubert et Fishwick, 1995] R. Cubert et P. Fishwick. *Sim++*. University of Florida, Juillet 1995. `http://www.cise.ufl.edu/~fishwick/simpack/simpack.html`.

[Dahl et Nygaard, 1966] O. Dahl et K. Nygaard. SIMULA–an ALGOL-Based Simulation Language. *Communication of the ACM*, 9(9):671–678, Septembre 1966.

[Damitio *et al.*, 1993] M. Damitio, L. Ford, J. Laulhé, R. Oizel, et S. Turner. GUIDES : An Interactive Graphical Representation of the Time Warp Mechanism. In A. Pave, editor, *European Simulation Multiconference – Modeling and Simulation*, pages 157–161, Lyon–France, Juin 1993.

[de Pretto, 1995] T. de Pretto. Conception et Implémentation d'un outil réalisant des traitements statistiques pour simulateurs parallèles. Technical report, DESS-ISI, Université de Nice-Sophia Antipolis, France, 1995.

[Delcambre et Pollacia, 1994] M. Delcambre et L. Pollacia. Object Oriented Conceptual Modeling of Discrete Event Systems. In Beaumariage Herring, Wallace et Roberts, editors, *Object Oriented Simulation Conference*, 1994.

[DIS, 1996] *DIS Workshop on Standard for the Interoperability of Distributed Simulation*, Orlando, Florida, Mars 1996.

[Doyle, 1990] R. Doyle. Object Oriented Simulation Programming. In *Object Oriented Simulation*, pages 1–6, San Diego CA, 1990.

[Duval et Vaucher, 1975] P. Duval et J. Vaucher. A Camparison of Simulation Event List Algorithms. *Communications of the ACM*, 18(4):223–230, Avril 1975.

[Egdorf et Painter, 1994] H. Egdorf et S. Painter. An Object Oriented Methodology for Discrete Event Simulation Tasks: Requirements, Funtional Specification, Design, Implementation. In Beaumariage Herring, Wallace et Roberts, editors, *Object Oriented Simulation Conference*, 1994.

[Eicken *et al.*, 1992] T. Eicken, D. Culler, S. Goldstein, et K. Schauser. Active messages: a mechanism for integrated communication and computation. In *19th int'l Symposium on computer architecture*, 1992.

[Ferber, 1989] J. Ferber. Computational Reflection Class based Object Oriented Languages. In *Object-Oriented Programming: Systems Languages and Applications*, volume 24(10), pages 317–326, Octobre 1989.

[Ferscha, 1995a] A. Ferscha. *Handbook of Parallel and Distributed Computing*, chapter Parallel and Distributed Simulation of Discrete Event Systems. McGraw-Hill, 1995.

[Ferscha, 1995b] A. Ferscha. Probabilistic Adaptive Direct Optimism Control in Time Warp. In *Parallel and Distributed Simulation*, pages 120–129, Lake Placid,New-York, Juin 1995.

[Finzi, 1996] D. Finzi. Conception et Implémentation d'une Bibliothèque pour la Simulation de Réseaux de Petri Temporisés. Rapport de fin de stage IUP Génie Mathématique et Informatique Dauphine, Septembre 1996.

[Flanagan, 1996] D. Flanagan. *Java in a nutshell.* O'Reilly international Thomson, 1996.

[Fleischmann et Wilsey, 1995] J. Fleischmann et P. Wilsey. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Parallel and Distributed Simulation*, pages 50–58, Lake Placid, New York, Juin 1995.

[Ford et Lepreau,] B. Ford et J. Lepreau. Evolving Mach 3.0 to Migrating Thread Model. Ftp `cs.utah.edu pub/`.

[Forum, 1993] The MPI Forum. MPI: A Message Passing Interface. Technical report, university of Tenesse, 1993.

[Fowler et Scott, 1997] M. Fowler et K. Scott. *UML Distilled, Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

[Franta et Maly, 1977] W. Franta et K. Maly. An Efficient Data Structure for the Simulation Event Set. *Communication of the ACM*, 80(8):596–602, Août 1977.

[Fujimoto *et al.*, 1992] R. Fujimoto, J. Tsai, et C. Gopalakrishnan. Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp. *IEEE Trans. on Computers*, 41(1):68–82, Janvier 1992.

[Fujimoto et Jand, 87] R. Fujimoto et J. Jand. The Rollback Chip: Hardware Support for Distributed Simulation using Time Warp. Technical Report UUCS-87-025, University of Utah, 87.

[Fujimoto et Nicol, 1992] R. Fujimoto et D. Nicol. State of the Art in Parallel Simulation. In Swain *et al.*, editor, *winter simulation conference*, pages 246–254, Arlington, Virginia, Décembre 1992.

[Fujimoto, 1989] R. Fujimoto. Performance Measurements of Distributed Simulation Strategies. *Transaction of the Society of Computer Simulation*, 6(2):89–132, Avril 1989.

[Fujimoto, 1990] R. Fujimoto. Parallel Discrete Event Simulation. *Communication of the ACM*, 33(10):30–53, Octobre 1990.

[Furmento *et al.*, 1995] N. Furmento, Y. Roudier, et G. Siegel. Parallélisme et Distribution en C++, Une Revue des Langages Existants. Technical report, I3S Laboratory, University of Nice – Sophia Antipolis, CNRS – URA 1376, Avril 1995. Une version résumée a été présentée lors de la Septième Rencontre Francophones du Parallélisme, RenPar'7.

[Furmento et Baude, 1996] Nathalie Furmento et Françoise Baude. Schooner: An Object-Oriented Run-time Support for Distributed Applications. In K. Yetongnon et S. Hariri, editors, *Proceedings of Parallel and Distributed Computing Systems (PDCS'96)*, volume 1, pages 31–36, Dijon (France), Septembre 1996. International Society for Computers and their Applications (ISCA). ISBN: 1-880843-17-X.

[Gafni, 1988] A. Gafni. Rollback Mechanisms for Optimistic Distributed Simulation Systems. In B. Unger et D. Jefferson, editors, *SCS Multiconference on Distributed Simulation*, volume 19(3), pages 61–67, San Diego California, Février 1988.

[Gamma *et al.*, 1995] E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gaudel *et al.*, 1996] M. Gaudel, B. Marre, F. Schlienger, et G. Bernot. *Précis de génie logiciel.* MASSON, 1996.

[Gaujal *et al.*, 1997a] B. Gaujal, A. Jean-Marie, P. Mussi, et G. Siegel. High Speed Simulation of Discrete Event Systems by Mixing Process Oriented and Equational Approaches. *Parallel Computing – special issue: environment and tools for parallel scientific computing*, 23(1–2):219–234, Avril 1997.

[Gaujal *et al.*, 1997b] B. Gaujal, A. Jean-Marie, et G. Siegel. Fast Simulation of an ATM Backbone Mixing Process Oriented and Equational Approaches. Personnal communication, 1997.

[Gaujal *et al.*, 1997c] B. Gaujal, A. Jean-Marie, et G. Siegel. Modeling of ATM Switches Using Deterministic Petri Nets and Queuing Networks. Personnal communication, 1997.

[Gavrilov et Vinogradov, 1995] D. Gavrilov et O. Vinogradov. Object-Oriented Programming and Object-Oriented Simulation. In *Summer Computer Conference*, pages 143–148, Ottawa - Canada, Juillet 1995.

[Gioanni, 1996] F. Gioanni. Mécanisme de Trace dans PROSIT. Rapport du projet de fin d'étude ESSI, Avril 1996.

[Goldberg et Robson, 1983] A. Goldberg et D. Robson. *Smalltalk-80 : the language and its implementation*. Addison Wesley, 1983.

[Golmie *et al.*, 1995] N. Golmie, A. Koenig, et D. Su. The NIST ATM Network Simulator. Technical report, U.S. Department of Commerce, Août 1995. `ftp://isdn.ncsl.nist.gov/atm-sim/README` .

[Greenberg *et al.*, 1990] A. Greenberg, B. Lubachevsky, et I. Mitrani. Unboundedly Parallel Simulations via Recurrence Relations. In *ACM SIGMETRICS Conference in Measurement and Modeling of Computer Systems*, 1990.

[Grunwald, 1991] D. Grunwald. User's Guide to AWESIME-2, A Widely Extensible Simulation Environment. Technical Report CU-CS-552-91, University of Colorado at Boulder, 1991.

[Harold et Malloy, 1993] J. Mc Gregor M. Harold et B. Malloy. The Implementation of a Simulation Language using Dynamic Binding. In D. Jefferson R. Bagrodia, editor, *Parallel and Distributed Simulation*, pages 3–8, San Diego, California, Mai 1993.

[Heidelberger, 1986] P. Heidelberger. Statistical Analysis of Parallel Simulations. In *Winter Simulation Conference*, pages 290–295, Washington, DC, Décembre 1986.

[Heidelberger, 1988] P. Heidelberger. Discrete-Event Simulations and Parallel Processing: Statistical Properties. *SIAM Journal on Statistical Computing*, 9(6):1114–1132, Novembre 1988.

[Hill *et al.*, 1993] D. Hill, M. Gourgand, et P Kellert. Object-Oriented Simulation Animation Builder. In *Annual Simulation Symposium*, volume 93TH0512-4, pages 237–246, Arlington, Virginia, 1993.

[Hill, 1994] D. Hill. *Analyse Orientée Objets et Modélisation par Simulation*. Addison–Wesley, 1994.

[Hu, 1993] J. Hu. CEDES - A C++ Engine for Discrete Event Simulation. In *Simulation Symposium*, volume 93TH0512-4, pages 189–195, Arlington, Virginia, Mars 1993.

[IEEE, 1993] IEEE. Proposed IEEE Standard Draft: Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications, Version 2.0, Third Draft. Technical Report No. IST-CR-93-15, Institute for Simulation and Training, University of Central Florida, Mai 1993.

[INRIA, 1990] INRIA, editor. *Ecole d'été sur la construction des systèmes d'exploitation répartis*. INRIA, 1990.

[Jackson et Zobel, 1993] T. Jackson et R. Zobel. Attachable Object Oriented Graphical Display System for Simulation and Instrumentation. In *Modelling and Simulation, FSM Lyon 1993*, pages 147–151, 1993.

[Jean-Marie, 1997] A. Jean-Marie. The SPADES Environment. to appear, INRIA, 1997.

[Jefferson et Reiher, 1991] D. Jefferson et P. Reiher. Supercritical Speedup. In IEEE Computer Society Press, editor, *24th Annual Simulation Symposium*, pages 159–168, Avril 1991.

[Jefferson, 1985] D. Jefferson. Virtual Time. In *ACM Trans. on Programming Languages and Systems*, volume 7(3), pages 404–425, Juillet 1985.

[Jefferson, 1990] D. Jefferson. Virtual Time II: Storage Management in Distributed Simulation. Technical report, UCLA, Janvier 1990.

[Jiménez, 1997] T. Jiménez. Simulation de trafic routier : Etude des méthodologies de modélisation pour la micro-simulation et mise en œuvre. Master's thesis, Université de Nice–Sophia Antipolis, ESSI, 1997.

[Jones *et al.*, 1989] D Jones, C. Chou, D. Renk, et C. Bruel. Experience with Concurrent Simulation. In *Winter Simulation Conference*, volume IEEE Catalog, 89CH2778-9, pages 756–764, Washington, DC, Décembre 1989.

[Jones, 1986] D. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communication of the ACM*, 29(4):300–311, Avril 1986.

[Keppel, 1993] D. Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Technical Report UWCSE 93-05-06, University of Washington, ftp `ftp.cs.washington.edu`, 1993.

[Kleinrock, 1975] L. Kleinrock. *Queueing Systems : I Theory.* Wiley interscience publication, 1975.

[Knuth, 1973] D. Knuth. *The Art of Computer Programming, Volume 1 / Fundamental Algorithms.* Eddison Wesley, 1973.

[Konas et Yew, 1995] P. Konas et P. Yew. Partitioning for Synchronous Parallel Simulation. In *Parallel and Distributed Simulation*, pages 181–184, Lake Placid, New York, Juin 1995.

[Krone *et al.*, 1995] O. Krone, M. Aguilar, et B. Hirsbrunner. PT-PVM: Using PVM in a multi-threaded environment. In *EURO-PVM*, 1995.

[Kuhl *et al.*, 1994] F. Kuhl, W. Neal, et H. Cohen. Object Request Broker: Foundation for Distributed Simulation. *Software Practice and Experience*, 24(12):1145–1157, Décembre 1994.

[Lamprecht, 1981] G. Lamprecht. *Introduction to Simula-67.* Braunschweig ; Wiesbaden : F. Vieweg und Sohn, 1981.

[Leroudier, 1980] J. Leroudier. *La Simulation à Evénements Discrets.* Edition des Hommes et Techniques, 1980.

[Lin *et al.*, 1990] Y. B. Lin, E. D. Lazowska, et J. L. Baer. Conservative Parallel Simulation for Systems with no Lookahead Prediction. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 22(2), pages 144–149, San Diego CA, Janvier 1990. SCS.

[Lin *et al.*, 1993] Y. Lin, W. Loucks, E. Lazowska, et B. Preiss. Selecting the Optimal Checkpoint Interval in TW Parallel Simulation. In D. Jefferson R. Bagrodia, editor, *Parallel and Distributed Simulation*, volume 23(1), pages 3–10, San Diego, California, Mai 1993.

[Lin et Lazowska, 1989] Y. Lin et E. Lazowska. The Optimal Checkpointing Interval in TW Parallel Simulation. Technical report, Uni. Of Washington, 1989.

[Lin et Lazowska, 1990] Lin et Lazowska. Determining the Global Virtual Time in a Distributed Simulation. In *Int. Conf. on Parallel Processing*, 1990.

[Lin et Schab, 1994] K. Lin et D. Schab. The Performance Assesment of Dead Reckoning Algorithms in DIS. *Simulation*, 63(5):318–325, Novembre 1994.

[Lin, 1992] Y. Lin. Memory Management Algorithms for Optimistic Parallel Simulation. In *Parallel and Distributed Simulation*, volume 24(3), pages 43–52, Newport Beach, California, Janvier 1992.

[Liskov *et al.*, 1986] B. Liskov, M. Herlihy, et L. Gilbert. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. In *13th ACM Symposium on Principles of Programming Languages*, pages 150–159, St Petersburg, Florida, 1986.

[Little et McCue, 1993] M. Little et D. McCue. Construction and Use of a Simulation Package in C++. Technical Report 437, University of Newcastle upon Tyne, Juillet 1993. also in: C User's Journal 12(3) March 1994- `http://ulgham.ncl.ac.uk/C++SIM/homepage.html`.

[Liu et Tropper, 1990] L. Liu et C. Tropper. Local Deadlock Detection in Distributed Simulation. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 22(2), pages 64–69, San Diego CA, Janvier 1990. SCS.

[Loucks et Nandy, 1992] W. Loucks et B. Nandy. An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers. In *Parallel and Distributed Simulation*, volume 24(3), pages 139–146, Newport Beach, California, Janvier 1992.

[Loucks et Nandy, 1993] W. Loucks et B. Nandy. On a parallel partitioning technique for use with conservative parallel simulation. In D. Jefferson R. Bagrodia, editor, *Parallel and Distributed Simulation*, pages 43–51, San Diego, California, Mai 1993.

[Loucks et Preiss, 1990] W. Loucks et B. Preiss. The Role of Knowledge in Distributed Simulation. In *SCS Multiconference on Distributed Simulation*, volume 22(1), pages 9–16, Janvier 1990.

[Lubachevsky *et al.*, 1989] B. Lubachevsky, A. Shwarts, et A. Weiss. Rollback Sometimes Works ... If Filtered. In *Winter Simulation Conference*, Washington, DC, Décembre 1989.

[Lubachevsky, 1988] B. Lubachevsky. Bounded Lag Distributed Discrete Event Simulation. In *Distributed Simulation*, pages 183–191, San Diego, CA, Février 1988.

[Madisetti *et al.*, 1988] V. Madisetti, J. Walrand, et D. Messerschmit. Wolf: A Rollback Algorithm for Optimistic Distributed Simulation Systems. In M. Abrams, P. Haigh, et J. Comfort, editors, *Winter Simulation Conference*, pages 296–305, San Diego, California, Décembre 1988. IEEE Catalog.

[Maes, 1987] P. Maes. Concepts and Experiments in Computational Reflection. In N. Meyrowitz, editor, *OOPSLA*, volume 22(12), pages 147–155, Orlando, Florida, Octobre 1987.

[Marsan *et al.*, 1995] M. Marsan, A. Bianco, T. Do, L. Jereb, R. Cigno, et M. Murafò. ATM simulation with CLASS. *Performance Evaluation*, 24:137–159, 1995.

[Marti *et al.*, 1996] D. Marti, P. Wilsey, et T. McBrayer. WARPED: A Time Warp Simulation Kernel for Analysis and Application Development. In *Hawaii International Conference on System Science*, 1996.

[Mascarenhas et Rego, 1995] E. Mascarenhas et V. Rego. Ariadne: Architecture of a portable threads system supporting mobil processes. Technical Report CSD-TR-95-017, Purdue University, Mars 1995.

[Matsumoto et Taki, 1993] Y. Matsumoto et K. Taki. Adaptative Time-Ceiling for Efficient Parallel Discrete Event Simulation. In T. Beaumariage et C. Roberts, editors, *Object Oriented Simulation Conference*, pages 101–106, La Jolla, California, 1993.

[Matsuoka et Yonezawa, 1993] S. Matsuoka et A. Yonezawa. *Research directions in concurrent object-oriented programming*, chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages, pages 107–150. MIT press, 1993.

[Mattern, 1987] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.

[Mattern, 1993] F. Mattern. Efficient Algorithms for Distributed Snapshot and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, 18(4):124–137, Août 1993.

[Mehl et Horst, 1991] Mehl et Horst. Speed-Up of Conservative Distributed Discrete Event Simulation Method by Speculative Computing. In *Advances in Parallel and Distributed Simulation*, pages 163–166, 1991.

[Meyer, 1988] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[Misra, 1986] J. Misra. Distributed Discrete-Event Simulation. In *ACM Computing Surveys*, volume 18(1), pages 39–65, 1986.

[Mohr, 1993] B. Mohr. Standardization of Event Traces Considerer Harmful or Is an Implementation of Object-Independant Event Trace monitoring and Analysis Systems Possible? In J.J. Dongarra et B. Tourancheau, editors, *Workshop "Environments and Tools For Scientific Parallel Computing"*, volume 6, pages 103–124. Elsevier, 1993.

[Mueller, 1993] Frank Mueller. A library implementation of POSIX threads under Unix. In *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, Janvier 1993.

[Murata, 1989] T. Murata. Petri nets: Properties, analysis and applications. In *IEEE*, pages 541–580, 1989.

[Mussi et Siegel, 1995a] P. Mussi et G. Siegel. Sequential Simulation in PROSIT: Programming Model and Implementation. Technical Report RR-2713, INRIA, Novembre 1995. Also in *European Simulation Symposium*, pages 297–301, Erlangen, Germany, October 1995.

[Mussi et Siegel, 1995b] P. Mussi et G. Siegel. The PROSIT Sequential Simulator: A Test-bed for Object Oriented Discrete Event Simulation. In *European Simulation Symposium*, pages 297–301, Erlangen, Germany, Octobre 1995.

[Mussi et Siegel, 1996] P. Mussi et G. Siegel. Extending the Prosit System to Support Distributed Simulation. In *European Simulation Symposium*, Genoa–Italy, Octobre 1996.

[Myricom, 1995] Myricom. MYRINET: A Gigabit per Second Local Area Network. *IEEE-Micro*, 15(1):29–36, Février 1995. `http://www.myri.com` .

[Näher, 1995] S. Näher. *The LEDA User Manual.* Max-Planck-Institut für Informatik, 3.1 edition, Janvier 1995.

[Namist, 1996] R. Namist. *PM² : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* PhD thesis, Université de Lille, 1996.

[Nicol, 1988] D. Nicol. Parallel Discrete Event Simulation of FCFS Stochastic Queuing Networks. *ACM SIGPLAN. Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 124–137, 1988.

[Nuttall, 1994] M. Nuttall. A brief survey of systems providing process or object migration facilities. *Operating System Review*, 28-4, Octobre 1994.

[OMG, 1991] OMG. The Common Object Request Broker: Architecture and Specification (CORBA). Technical report, Object Management Group, 1991. `http://www.omg.org` .

[OMG, 1992] OMG. *Object Management Architecture Guide (OMA Guide).* Number Second edition. OMG, Septembre 1992. `http://www.omg.org` .

[Orfali *et al.*, 1996] R. Orfali, D. Harkey, et J. Edwards. *The Essential distributed objects survival guide.* J Wiley and sons, 1996.

[Palaniswamy et Wilsey, 1993] A. Palaniswamy et P. Wilsey. An Analitycal Comparison of Periodic Checkpointing and Incremental State Saving. In D. Jefferson R. Bagrodia, editor, *Parallel and Distributed Simulation*, pages 127–134, San Diego, California, Mai 1993.

[Papathomas, 1990] M. Papathomas. Concurrency Issues in Object Oriented Programming Languages. Université de Genève, 1990.

[Peter et Milhau, 1996] Y. Peter et M. Milhau. Support de la mobilité dans CORBA. In *Ecole Placement et Répartition de Charge*, Juillet 1996.

[Pham, 1996] C. Pham. Simulations distribuées de réseaux ATM. *Calculateur Parallèle–Editions Hermès*, 8(2):229–243, 1996.

[Preiss et Loucks, 1995] B. Preiss et W. Loucks. Memory Management Technique for Time Warp on a Distributed Memory Machine. In *Parallel and Distributed Simulation*, pages 30–39, Juin 1995.

[Provenzano, 1996] C. Provenzano. Pthreads: an implementation of POSIX 1003.1.c. Technical report, MIT, 1996. `http://www.mit.edu/people/proven/pthreads.hml` .

[Pugh, 1990] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communication of the ACM*, 33(6):668–676, Juin 1990.

[Rakotoarisoa, 1994] H. Rakotoarisoa. *Simulation distribuée de réseaux de files d'attente*. PhD thesis, Université de Nice – Sophia Antipolis France, Février 1994.

[Reiher *et al.*, 1990] P. Reiher, R. Fujimoto, S. Bellenot, et D. Jefferson. Cancellation Strategies in Optimistic Execution Systems. In D. Nicol, editor, *SCS Multiconference on Distributed Simulation*, volume 22(2), pages 112–121, San Diego CA, Janvier 1990.

[Reiher et Jefferson, 1990] P. Reiher et D. Jefferson. Virtual Time Based Dynamic Load Management in the Time Warp Operating System. In D. Nicol, editor, *Multiconference on Distributed Simulation*, volume 22(2), pages 103–111, San Diego, CA, 1990.

[Reynolds, 1988] P. Reynolds. A Spectrum of Options for Parallel Simulation. In M. Abrams, P. Haigh, et J. Comfort, editors, *Winter Simulation Conference*, pages 325–332, San Diego, California, Décembre 1988. IEEE Catalog.

[Righter et Walrand, 1989] R. Righter et J. Walrand. Distributed Simulation of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):99–113, Janvier 1989.

[Rose, 1992] O. Rose. *SIM++ Part I- Sim An Object-Oriented Language for Process-Oriented Discrete-Event Simulation*. Institute of Telematics University of Karlsruhe, 1992.

[Rossum, 1994] G. Rossum. Python Tutorial. Technical report, CWI, Netherlands, Janvier 1994. `http://www.cit.ics.saitama-u.ac.jp/published/manual/python/Python.html`.

[Rumbauch, 1996] J. Rumbauch. *OMT, modélisation et conception orientées objet*. Masson, 1996.

[Sang *et al.*, 1994] J. Sang, K. Chung, et V. Rego. A Simulation Testbed based on Lightweight Processes. *Software Practice and Experience*, 24(5):485–505, Mai 1994.

[Sang *et al.*, 1996] J. Sang, E. Mascarenhas, et V. Rego. Mobile-Process-Based Simulation. *Journal of Parallel and Distributed Computing*, 33:12–23, 1996.

[Sargent, 1984] R. Sargent. A Tutorial on Verification and Validation of Simulation Models. In S. Sheppard, U. Pooch, et D. Pegden, editors, *Winter Simulation Conference*, Dallas, Novembre 1984.

[Shapiro *et al.*, 1989a] M. Shapiro, P. Gautron, et L Mosseri. Persistence and Migration for C++ Objects. In *ECOOP*, Juillet 1989.

[Shapiro *et al.*, 1989b] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, et C. Valot. SOS: An Object-Oriented Operating System – Assessment and Perspectives. *Computing System*, 2(4):287–337, 1989.

[Sheppard *et al.*, 1985] S. Sheppard, V. Chandredrasekaran, et K. Murray. Distributed Simulation using ADA. In P. Reynolds, editor, *Distributed Simulation*, volume 15(2) of *SCS Publication*, pages 27–31, 1985.

[Simulog, 1992a] Simulog. *Animator Programmer's Guide*, v 1.3 edition, Août 1992. Simulog, 1, Rue James Joule, 78286 Guyancourt, France.

[Simulog, 1992b] Simulog. *GSS4 User's Guide*, RR 2.2 Version 3 edition, Août 1992. Simulog, 1, Rue James Joule, 78286 Guyancourt, France.

[Simulog, 1995] Simulog. *QNAP2 Manual*. Simulog, Avril 1995. Simulog, 1, Rue James Joule, 78286 Guyancourt, France.

[Sleator et Tarjan, 1985] D. Sleator et R. Tarjan. Self-Adjusting Binary Search Trees. *JACM*, 32(3):652–686, Juillet 1985.

[Sokol et Stucky, 1990] L. Sokol et B. Stucky. MTW: Experimental Results for a Constrained Optimistic Scheduling Paradigm. *Multiconference on Distributed Simulation*, 22(2):169–173, Janvier 1990.

[Steinman, 1992] J. Steinman. SPEEDES: A Unified Approach to Parallel Simulation. In M Abrams et F. Reynolds, editors, *Parallel and Distributed Simulation*, volume 24(3), pages 75–83, Newport Beach, California, Janvier 1992.

[Steinman, 1993] J. Steinman. Breathing Time Warp. In R. Bagrodia et D. Jefferson, editors, *7th Workshop on Parallel and Distributed Simulation*, pages 109–118, San-Diego, California, Mai 1993.

[Stevens, 1990] W. Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[Stroustrup, 1993] B. Stroustrup. *Le Langage C++*. Addison Wesley, 1993.

[Stroustrup, 1994] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Toint, 1994] P. Toint. Introduction à PACSIM : un nouveau modèle dynamique comportemental pour la simulation du trafic. *Les cahiers du MET – collection trafics*, Novembre 1994. Ministère Wallon de l'équipement et des transports.

[Vaughan *et al.*, 1991] P. Vaughan, D. Newton, et R. Johns. PRISM: An Object-Oriented Modeling Environment in C++. In R. Edge, editor, *Object Oriented Simulation Conference*, volume 23(3), pages 32–39, Anaheim, California, Janvier 1991. SCS.

[Wegner, 1987] P. Wegner. Dimensions of Object-Based Language Design. In N. Meyrowitz, editor, *OOPSLA*, volume 22(12), pages 168–182, Orlando, Florida, Octobre 1987.

[West, 1998] D. West. Optimizing Time Warp: Lazy Rollback and Lazy Re-Evaluation. Master's thesis, University of Calgary, Alberta, 1998.

[Whitner et Balci, 1989] R. Whitner et O. Balci. Guidelines for Selecting and Using Simulation Model Verification Techniques. In *Winter Simulation Conference*, pages 559–568, Washington, DC, Décembre 1989.

[Worley, 1992] P. Worley. A new PICL Trace File Format. Technical report, Oak Ridge National Laboratory, Septembre 1992.

[Yu *et al.*, 1989] Q. Yu, D. Towsley, et P. Heidelberger. Time-driven Parallel Simulation of Multistage Interconnection Networks. In B. Unger et R. Fujimoto, editors, *SCS Multiconference on Distributed Simulation*, volume 021 Nr 2, pages 191–196, Tampa, Florida, 1989.

Résumé : Le travail de recherche présenté dans cette thèse porte sur la conception et la réalisation d'un environnement pour la programmation de simulations à événements discrets. Cet environnement, appelé Prosit, permet la mise en œuvre de simulateurs séquentiels et distribués. Prosit comprend aussi des librairies de modélisation ainsi que des outils d'aide à la programmation.

Dans un premier temps, nous présentons le modèle de programmation Prosit. Ce modèle est basé sur le concept d'objets actifs interagissants par le biais d'appels de méthodes réifiées. La réification permet un contrôle explicite et centralisé des objets actifs, et encourage la réutilisation, par héritage, de classes de comportements. L'extension distribuée du modèle introduit la migration d'objets actifs, et divers mécanismes de placement et de synchronisation.

Dans un second temps, nous donnons les grandes lignes de la mise en œuvre de l'environnement Prosit. Il se présente sous la forme d'un *canevas*, permettant la simulation de toute classe de systèmes à événements discrets. Ce canevas possède une version séquentielle et une autre distribuée, et chaque composante peut être spécialisée ou redéfinie. Nous présentons aussi dans cette partie les librairies et outils développés autour du canevas ainsi qu'une étude détaillée de ses performances.

Mots clés : Simulation à événements discrets – Simulation distribuée – Simulation objet – Environnement de programmation – Prosit.

Abstract: The main objective of this thesis is the conception and implementation of an environment for discrete event simulation programming. The environment, named Prosit, is based on a modeling process and a programming framework allowing to develop both sequential and distributed simulators. Modeling libraries and debugging tools have also been build around Prosit.

We first present the Prosit programming model. The model provides an active object architecture in which the simulated entities interact via reified function calls in the simulated time. Reification allows an explicit and centralized control of the active objects behavior. It is then possible to develop and reuse classes implementing specific or commonly used behaviors. Distributed simulation is achieved through remote service call and active objects migration. Different synchronization and mapping primitives are also available.

Afterwards, we detail the framework implementation. There is a sequential and distributed version and any discrete event system can be simulated. Each component of the framework can be customized either for optimization or for specialization. We also present the modeling libraries and tools build with the framework and, finally, we give a detailed evaluation of the performance.

Key words: Discrete event simulation – Distributed simulation – Objet oriented simulation – Programming framework – Prosit.