Université de Montréal

Une librairie orientée-objet pour la simulation des réseaux stochastiques dynamiques

par

Jocelyn Demers

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique et recherche opérationnelle

mai, 1999

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Une librairie orientée-objet pour la simulation
des réseaux stochastiques dynamiques

présenté par :

Jocelyn Demers

a été évalué par un jury composé des personnes suivantes :

Jean-Yves Potvin	président du jury
Teodor Gabriel Crainic	directeur de recherche
Michel Gendreau	codirecteur de recherche
Bernard Gendron	membre du jury

Mémoire accepté le : 8 septembre 1999

Sommaire

La solution des problèmes d'allocation de ressources prend souvent la forme d'un algorithme d'optimisation opérant sur un réseau stochastique dynamique. En raison des pouvoirs décisionnels qui sont conférés à ces algorithmes, il est essentiel d'évaluer empiriquement la performance des solutions qu'ils proposent. Or, ces expériences supposent une évolution stochastique dynamique implicite du système étudié, hypothèse uniquement vérifiée dans le système réel. L'acquisition d'un environnement expérimental virtuel qui permet de reproduire cette dimension des problèmes étudiés est donc indispensable à la conduite de ces expériences.

Un tel environnement nous est offert par la simulation. En effet, elle procure à ses utilisateurs tous les bénéfices d'un environnement expérimental parfaitement contrôlable. Une utilisation novatrice du paradigme de programmation orientée-objet, tel que défini par le langage C++, permet d'implémenter une librairie de simulation pour les réseaux stochastiques dynamiques qui facilite la programmation inhérente à son utilisation tout en permettant la création d'un environnement expérimental ouvert, éloquent et extensible. Pour ce faire, les mécanismes d'héritage et les fonctions virtuelles coopèrent afin de permettre l'évolution des classes de la librairie ainsi que la

redéfinition ponctuelle de leurs composantes. De plus, un autre mécanisme orienté-objet, les patrons, autorisent la création de classes modulaires dont les attributs sont interchangeables. Ces mécanismes sont néanmoins assujettis à la création d'un champ type dans les classes de la librairie qui permet d'identifier dynamiquement la classe effective d'un objet.

La pertinence de concevoir de cette manière une librairie pour la simulation des réseaux stochastiques dynamiques est prouvée en simulant le problème de l'allocation des conteneurs vides. La simulation réalisée reproduit les offres et demandes de conteneurs vides des clients d'une compagnie maritime pendant un an. À cette simulation est adjoint un algorithme d'optimisation externe afin de démontrer les capacités de la librairie dans une application réelle. Parallèlement, les mécanismes d'extensibilité de la librairie sont explorés plus en détail dans des exemples moins ambitieux concernant le changement dynamique du niveau de détails d'une simulation et l'utilisation des nombres aléatoires communs.

En somme, ce mémoire établit le bien-fondé d'utiliser conjointement l'approche orientée-objet et la simulation pour réaliser un environnement expérimental ouvert qui permet d'évaluer la performance des algorithmes d'optimisation opérant sur des réseaux stochastiques dynamiques.

Table des matières

Liste des figures

Remerciements

L'auteur tient à remercier les professeurs Teodor Crainic et Michel Gendreau pour leur support financier et leurs recommandations tout au long de l'élaboration de ce mémoire. Remerciements également au professeur Pierre L'Écuyer pour ses précieux conseils concernant l'utilisation de son générateur aléatoire.

INTRODUCTION

Depuis la nuit des temps, le domaine des transports est motif ou composante essentielle aux plus grandioses réalisations de l'humanité. La construction des pyramides, les voies romaines, la route de la soie, les canaux de Suez et de Panama ainsi que le tunnel sous la Manche sont tous des exemples historiques qui témoignent de l'importance passée et présente des transports dans les activités quotidiennes de notre civilisation.

Paradoxalement, les problèmes de transports n'ont reçu que très tardivement une attention scientifique rigoureuse. La deuxième guerre mondiale, première guerre de mouvements disputée sur mer, air et terre, a rapidement confronté les stratèges des deux camps à des problèmes logistiques d'une complexité sans précédent. La victoire étant tributaire de la résolution de ces problèmes, de nombreux scientifiques ont été affectés à leur analyse. La recherche opérationnelle était née.

La recherche opérationnelle est la science des décisions. Plus précisément, la recherche opérationnelle appliquée au domaine des transports a pour but de procurer aux agents de décision les informations pertinentes pour établir des politiques d'opération

optimales. Ces informations peuvent prendre plusieurs formes, dont les plus évidentes sont des trajets et des horaires. La démarche utilisée par les chercheurs opérationnels pour parvenir à des recommandations est divisée en deux volets : modélisation du problème à l'étude et résolution du modèle conçu.

Il existe deux types de modèles en recherche opérationnelle. Le premier est le modèle dit déterministe. Ces modèles s'appliquent à des situations où les effets d'une action quelconque peuvent être déterminés avec certitude. Par opposition, les modèles dits probabilistes s'appliquent à des situations où l'état du système est relié à différentes lois de probabilités. Il existe donc une incertitude quant aux réponses du système à des actions spécifiques. Les modèles déterministes sont par conséquent généralement plus simples à résoudre que les modèles probabilistes. Cependant, en raison des composantes stochastiques inhérentes à tout problème de transport, les modèles probabilistes sont plus réalistes que les modèles déterministes pour les modéliser.

Un modèle probabiliste d'un problème de transport se nomme réseau stochastique dynamique (RSD). Il se compose d'une collection d'entités interdépendantes dont l'état évolue selon certaines lois de probabilité en fonction du temps. Les problèmes de transport impliquent généralement un grand nombre d'entités dont les comportements spécifiques sont bien définis et dont les interactions entre elles sont comprises. Cependant, la diversité de ces interactions ainsi que leur nombre rendent impossible de cerner précisément les comportements collectifs des entités qui composent le problème. La modélisation en RSD fait l'hypothèse que seules les interactions de base entre les entités sont suffisantes pour décrire les comportements

d'un système. Selon cette approche, la reproduction d'un nombre raisonnable de ces interactions de base recréera les comportements collectifs observés dans le système réel. Sous réserve de cette hypothèse, les RSD sont donc parfaitement adaptés pour représenter de façon réaliste et complète les problèmes de transport modernes.

Les RSD auxquels ce mémoire s'intéresse particulièrement sont les problèmes d'allocations de ressources. Ils consistent à disposer et à affecter de façon optimale une flotte de ressources de façon à satisfaire un ensemble de demandes. La complexité de ces problèmes a cependant pour effet de nécessiter la création de RSD que seuls les ordinateurs peuvent résoudre. Pour ce faire, on a recours à des algorithmes d'optimisation, ensemble de règles qui permet de déterminer la solution d'un RSD. Habituellement, ces algorithmes d'optimisation sont invoqués pour guider des opérateurs humains dans un choix complexe. Cette dépendance sur la machine crée le besoin de vérifier empiriquement la performance des solutions qu'elle propose. Or, ces expériences sont difficiles à conduire puisqu'il faut reproduire la mécanique complexe d'un RSD. C'est dans le contexte de cette problématique que ce mémoire propose un outil de simulation ayant pour but de promouvoir l'étude statistique rigoureuse de la validité des solutions apportées à des RSD.

La simulation consiste à animer un modèle à l'aide de l'informatique. Le programme de simulation utilise les lois de probabilités contenues dans le modèle pour générer une instance possible du problème étudié. Cette dernière peut ensuite être soumise à la solution proposée au modèle pour quantifier ses performances. La répétition de cette démarche un nombre suffisant de fois permet

d'établir la performance de la solution proposée selon une marge d'erreur prédéterminée.

L'obstacle majeur à une utilisation plus répandue de la simulation est la difficulté de la programmation qui lui est associée. En effet, le modèle doit premièrement être traduit de sa représentation symbolique en représentation digitale sans perte de détails. Par la suite, il faut programmer un interpréteur pour générer des instances du problème à partir de son modèle. Finalement, il faut recueillir des métriques sur certains paramètres du système pour quantifier la performance de la solution. L'outil de simulation présenté dans ce mémoire facilite chacune de ces étapes.

L'approche utilisée dans la réalisation de l'outil de simulation présenté dans ce mémoire peut être résumée en une seule phrase : la simulation est indissociable de la programmation. Or, plusieurs tentatives pour faciliter l'utilisation de la simulation ont consisté à isoler les usagers du langage de programmation. Si on prend en considération que l'essence même d'une simulation est de reproduire de la façon la plus exacte qui soit les comportements d'un système réel, il est indéniable qu'un mécanisme qui permet la modélisation sur mesure des spécificités qui leur sont propres est nécessaire. C'est pourquoi l'outil de simulation présenté dans ce mémoire prend la forme d'une librairie, c'est-à-dire une collection d'éléments dont les services sont connexes et destinés à augmenter les capacités d'un langage de programmation dans un domaine spécifique. De cette façon, on s'assure de disposer d'un outil de simulation flexible.

Flexibilité n'est cependant pas synonyme d'extensibilité. La contribution principale de ce mémoire à la simulation des RSD et à la

recherche opérationnelle en général sera de démontrer que l'utilisation judicieuse des mécanismes orientés-objet, tels l'héritage et les patrons (*templates*), permet de créer une librairie de simulation facilement extensible qui correspond aux besoins de la simulation des RSD. Ce mémoire démontrera également les bénéfices conceptuels de l'approche orientée-objet pour définir des RSD facilement et rapidement à l'aide de structures prédéfinies.

Le premier chapitre de ce mémoire introduit les problèmes d'allocation de ressources et notre problématique. Par la suite, le deuxième chapitre motive l'utilisation de la simulation en recherche opérationnelle et précise les objectifs de la librairie présentée. Le troisième chapitre présente une revue de littérature sur les concepts reliés à la création d'un outil de simulation et le chapitre quatre discute des contributions au domaine de ce mémoire. Le cinquième chapitre est consacré à la description détaillée des mécanismes de la librairie et le chapitre six présente des exemples d'utilisations. Pour terminer, les conclusions qui peuvent être tirées de cet exercice sont exposées.

CHAPITRE 1 Problématique

Ce chapitre est dédié à la description de la problématique motivant les travaux présentés dans ce mémoire. Tout d'abord, le contexte de notre problématique sera illustré avec des problèmes de transport réels empruntés à la littérature. Par la suite, la nécessité de posséder un environnement expérimental qui permet d'analyser la performance des algorithmes d'optimisation conçus pour résoudre ces problèmes sera mise en évidence. Pour conclure, les modalités souhaitables d'un tel environnement seront exposées.

1.1 Contexte

Les applications de la recherche opérationnelle au domaine des transports sont nombreuses et variées. Entre autres, des chercheurs opérationnels s'intéressent aux problèmes de tournées, problèmes qui consistent à déterminer un itinéraire optimal pour établir un circuit fermé entre plusieurs sites, tandis que d'autres modélisent des systèmes de transport en commun afin d'estimer la meilleure cadence de service sur les lignes d'autobus. Les problèmes auxquels ce mémoire s'intéresse particulièrement sont les problèmes d'allocation de ressources, c'est-à-dire les problèmes qui ont pour objet la découverte d'une façon optimale d'affecter une flotte de ressources afin de satisfaire un ensemble de demandes. Parallèlement, ces

problèmes sont aussi concernés par la relocalisation appropriée des ressources disponibles afin de pouvoir satisfaire les demandes futures le plus efficacement possible.

Un premier exemple de problème d'allocation de ressources est la gestion d'une flotte de camions tracteurs pour les voyages longues distances (Powell et al. 1988). Dans cette instance du problème, les clients demandent la prise en charge par la compagnie de camionnage d'une ou plusieurs remorques situées à leur site afin qu'elles soient acheminées à destination. Les destinations desservies couvrent la totalité de l'Amérique du nord et ce avec une équipe d'environ 2000 camionneurs. Le problème consiste à évaluer les chargements offerts pour ensuite les assigner aux camionneurs disponibles de façon à maximiser les opportunités de profits. Ceci implique le service en priorité des voyages à destination des villes offrant de bonnes possibilités de devenir les points de départ de nouveaux voyages et la création de cycles débutant et se terminant près des villes d'origine des camionneurs. De plus, cette planification doit être effectuée en connaissant environ 35% des demandes totales pour la journée et aussi peu que 10% des demandes pour le jour suivant en raison des besoins irréguliers des clients de la compagnie.

Un deuxième exemple de problème d'allocation de ressources, toujours dans le domaine du camionnage, est l'optimisation et la révision des opérations de chargement d'une compagnie de courrier spécialisée dans le transport de petits paquets (Braklow et al. 1992). La manière usuelle de procéder pour la compagnie de courrier est de rassembler tous les paquets provenant d'une région à son dépôt local afin de les consolider selon leur destination dans les camions-remorques qui parcourent les arcs de son réseau de distribution. À

chaque dépôt, les paquets sont consolidés selon leur destination dans une nouvelle remorque pour éventuellement atteindre leur destination finale. Cette compagnie couvre l'ensemble des États-Unis, possède une flotte de 45 000 camions et remorques, recueille environ 60 000 paquets par jour et doit gérer de façon journalière les déplacements de 10 000 camions entre ses différents dépôts. Le problème consiste à déterminer à quel moment une remorque doit quitter un dépôt et à décider, lorsque le volume de paquets ou la situation l'exige, l'établissement de liaisons directes temporaires entre deux dépôts ne communiquant pas habituellement afin de satisfaire les délais de livraisons assurés au client tout en minimisant les coûts pour la compagnie.

Un troisième et dernier exemple d'une instance de notre classe de problèmes est l'allocation et la gestion d'une flotte de conteneurs vides (Crainic et al. 1993). Ce problème s'intéresse à la récupération et à l'allocation des conteneurs vides résultants ou nécessaires aux opérations d'importation et d'exportation des clients européens d'une compagnie maritime. Ces clients se situent principalement en France et au Benelux. Les conteneurs vides peuvent emprunter les routes, les voies ferrées ou une combinaison de ces deux moyens de transport pour transiter entre les 138 dépôts et les 717 clients de la compagnie. Une caractéristique intéressante de ce problème est la possibilité de substituer un des 21 types de conteneurs dans la demande d'un client par un autre. En effet, lorsque le contenu prévu des conteneurs le permet, il est par exemple possible de substituer deux conteneurs de 20 pieds pour un de 40 pieds et vice et versa. Le défi posé par ce problème consiste à gérer la flotte de conteneurs vides de façon à toujours pouvoir satisfaire les demandes des clients dans les délais

prescrits et à déterminer où acheminer les conteneurs vides récupérés chez les clients.

Avant de débuter une analyse plus approfondie des problèmes d'allocation de ressources, inspirée des développements de Powell (1998), deux caractéristiques de cette classe de problèmes méritent d'être mentionnées immédiatement. En effet, elles sont facilement identifiables dans les trois exemples que nous avons présentés. La première est l'ampleur des systèmes étudiés. L'envergure des réseaux de transport utilisés ainsi que le nombre et la taille des inventaires de ressources à administrer est telle que seule l'informatique permet d'en gérer la complexité d'une façon efficace et cohérente. La solution des problèmes d'allocation de ressources passe obligatoirement par des algorithmes d'optimisation ou des systèmes experts. Une deuxième caractéristique de ces systèmes est la poursuite du profit qui les anime. En conséquence, il est désirable de pouvoir vérifier l'impact de tout changement aux opérations du système afin de ne pas provoquer d'éventuelles pertes de rentabilité. Par analogie, le chercheur opérationnel a la même responsabilité envers un système qu'il veut améliorer que l'ingénieur envers les structures qu'il construit.

On peut formaliser les problèmes d'allocation de ressources en distinguant leur trois composantes fondamentales : les ressources, les requêtes et les actions. Les ressources sont des commodités en quantité limitée dont l'usage doit être rentabilisé. Des exemples de ressources provenant des problèmes de la section précédente sont des camions, des remorques et des conteneurs. Les requêtes quant à elles sont des demandes relatives à l'attribution ou à la récupération de ressources. Un client qui réclame le transport de ses remorques constitue une requête. Pour leur part, les actions sont des tâches dont

l'accomplissement requiert des ressources. Le repositionnement stratégique d'équipement, la consolidation de chargement et la satisfaction d'une requête sont toutes des actions. En conséquence, la solution à un problème d'allocation de ressources consiste à orchestrer les actions qui permettent de disposer au moment et à l'endroit opportun des unités de ressources aptes à satisfaire les modalités physiques et temporelles d'un ensemble de requêtes.

Notre effort de formalisation peut encore être enrichi en analysant conjointement les exemples décrits précédement. En effet, on discerne alors les attributs types des problèmes d'allocation de ressources. Le plus évident est le nombre de catégories de ressources différentes à gérer simultanément. Il ne faut cependant pas penser que la complexité des problèmes d'allocation de ressources est uniquement fonction de cet attribut. D'ailleurs, nos exemples permettent de distinguer d'autres attributs déterminants dans la difficulté de ces problèmes tels les ressources homogènes et hétérogènes. Les systèmes à ressources homogènes possèdent un seul type de ressources apte à satisfaire les requêtes d'un type particulier, tandis que dans les systèmes à ressources hétérogènes, il existe plus d'un type de ressources pouvant satisfaire la même requête. Le transport de conteneurs par une compagnie qui gère une flotte de camions et de trains est un exemple de problème avec ressources hétérogènes. On distingue également les systèmes à ressources simples ou composées. Dans les systèmes à ressources simples, chaque requête nécessite l'allocation d'une seule et unique unité de ressources tandis que dans les systèmes à ressources composées, il faut coordonner plusieurs ressources pour satisfaire une seule requête. Le problème de la compagnie de courrier spécialisée dans le transport de petits paquets est un problème à ressources

composées puisqu'il faut affecter un chauffeur, un camion et une remorque pour chaque transit des paquets sur le réseau. Les systèmes les plus complexes à optimiser sont les systèmes à ressources hétérogènes composées en raison du grand nombre d'options à évaluer.

Il est possible d'améliorer notre classification des problèmes d'allocation de ressources en considérant les caractéristiques temporelles des requêtes. Les systèmes à requêtes planifiées sont les problèmes dans lesquels chaque requête est accompagnée d'un préavis assez long pour permettre une planification à moyen terme. On entend ici par moyen terme le temps nécessaire pour compléter des actions logistiques standards visant à satisfaire toutes les requêtes connues à condition qu'elles soient déclenchées sans délai. Les systèmes à requêtes rapides, quant à eux, permettent une planification à court terme, c'est-à-dire une planification tactique plutôt que stratégique. Pour leur part, les systèmes à requêtes immédiates nécessitent une action instantanée. Certains systèmes comportent des requêtes avec différents niveaux de préavis et on parle alors de problèmes à requêtes mixtes.

Un dernier moyen de distinguer deux problèmes d'allocation de ressources consiste à comparer les stratégies de service en usage. Dans les systèmes à service immédiat, les activités de service sont initiées par les requêtes. De leur coté, les systèmes à service planifié donnent suite aux différentes requêtes qui leur sont adressées selon un plan d'action prédéterminé. Les systèmes où des entorses au plan d'action sont permises lorsque la situation l'exige sont des systèmes à service mixte. La création de liaison temporaire entre des dépôts dans

l'exemple de la compagnie de courrier est un bon exemple de stratégie de service mixte.

Dans ce cadre de référence, les problèmes auxquels nous allons nous intéresser dans ce mémoire sont les problèmes relatifs à l'allocation de ressources dans des systèmes à ressources hétérogènes simples avec requêtes mixtes ou rapides et service mixte ou immédiat. Ces problèmes sont caractérisés par la flexibilité avec laquelle une requête particulière peut être satisfaite, avantage compensé par une certaine incertitude quant aux requêtes futures. De par nature, ces problèmes ont une forte composante stochastique qui implique une gestion conservatrice. En conséquence, les actions logistiques, et les coûts associés, ont une importance particulière dans ce type de problèmes.

1.2 Description de la problématique

La problématique qui nous intéresse est la difficulté d'évaluer de façon empirique la performance des algorithmes d'optimisation conçus pour résoudre les problèmes d'allocation de ressources. On entend ici par performance une mesure quantitative reliée à la qualité des solutions proposées par un algorithme d'optimisation et non une mesure de son efficacité à atteindre cette solution. Ces expérimentations sont importantes en raison des pouvoirs décisionnels qui sont conférés aux algorithmes d'optimisation. En effet, les systèmes qu'ils sont appelés à administrer ont un impact économique et social important et c'est pourquoi leur fiabilité doit être évaluée empiriquement avant de les laisser gérer les opérations d'un système réel de façon semi-autonome. La logistique des services d'urgence est

un exemple éclatant de problèmes d'allocation de ressource où aucune marge d'erreur n'est permise.

Une deuxième raison qui explique l'intérêt d'expérimenter avec un algorithme d'optimisation est l'amélioration de nos connaissances sur ses mécanismes et sur les problèmes qu'il permet de résoudre. En effet, une étude de la relation entre les performances d'un algorithme et ses paramètres permet d'isoler les composantes du système qui embarassent l'algorithme. La contribution de ces composantes au problème peut alors être étudiée plus en détails, pavant ainsi la voie à des recherches théoriques qui se traduiront par une meilleure compréhension du problème et d'éventuelles améliorations à l'algorithme. De plus, l'étude minutieuse d'une solution particulièrement disgracieuse produite par un algorithme permet d'identifier avec précision le point tournant où la logique utilisée pour évaluer le système a failli.

Il existe malheureusement un prérequis difficile à satisfaire pour pouvoir conduire une expérience visant à établir la performance de la solution produite par un algorithme d'optimisation. En effet, l'expérience suppose une évolution dynamique implicite du système étudié. Les systèmes que nous avons décrits précédement ne sont pas des objets statiques, leur état change avec le temps. De plus, la solution produite par un algorithme d'optimisation est une séquence de décisions qui s'étale sur une période de temps plus ou moins longue. Chacune de ces décisions modifie le système afin de créer le contexte particulier qui explique la prochaine décision dans la séquence. En conséquence, l'expérience doit reproduire et faire respecter les règles qui gouvernent les changements d'états dynamiques du système étudié.

Cette situation se généralise au système qui possède des composantes aléatoires. L'expérience doit alors contenir les règles qui permettent de déterminer les valeurs des paramètres stochastiques du système. Les règles dynamiques et stochastiques de l'expérience forment ce que nous appellerons son contexte stochastique dynamique. La mise en place de ce contexte dans une expérience implique la mise en scène et le suivi de la totalité des opérations du système réel. Il va sans dire que le travail représenté par une telle tâche est non trivial. L'objectif de ce mémoire est de présenter un environnement expérimental qui facilite cette tâche.

1.3 Modalités de l'environnement expérimental

Une caractéristique fondamentale de l'environnement expérimental que nous recherchons est sa nature virtuelle. En effet, la taille et la complexité des systèmes étudiés excluent automatiquement toute reproduction physique de leurs opérations. Néanmoins, les répliques virtuelles utilisées doivent correspondre fidèlement au système modélisé et posséder le même niveau d'aggrégation afin de permettre des expérimentations réalistes. De plus, des mécanismes permettant d'accomoder les composantes aléatoires des systèmes reproduits sont indispensables. Entre autre, les paramètres stochastiques observés dans une expérience doivent être reproductibles afin de permettre la répétition des résultats obtenus.

Par ailleurs, la symbolique de l'environnement expérimental requis doit être similaire à celle des problèmes étudiés. En effet, il est ainsi plus facile de définir le contexte stochastique dynamique de l'expérience puisqu'une correspondance exacte entre ses règles et

celle du système étudié peut alors être établie. De plus, il est souhaitable que l'environnement expérimental prédéfinisse la symbolique commune à tous les problèmes d'allocation de ressources de façon à accélérer la définition des contextes stochastiques dynamiques relatifs à ces problèmes. Toutefois, cette symbolique doit pouvoir être étendue ou remplacée lorsque les besoins particuliers d'une expérience ou d'un algorithme l'exigent.

Au point de vue fonctionnel, l'environnement expérimental recherché permet la conduite des trois types d'expériences qui permettent de déterminer de façon rigoureuse la performance d'un algorithme : les études de dépendance, les études de robustesse et les études formelles (McGeoch 1996). Les études de dépendance tentent d'établir une relation entre les paramètres de l'algorithme d'optimisation et la qualité de la solution obtenue. Les études de robustesse quant à elles analysent la distribution des mesures de performance atteintes par un algorithme lorsque confronté à des situations variées. Finalement, les études formelles explorent en détails chaque décision prise par un algorithme d'optimisation afin d'en distinguer les contributions unitaires à la performance totale de l'algorithme.

1.4 Conclusion

Les trois exemples de problèmes d'allocation de ressources que nous avons décrits illustrent bien la taille et la complexité de ces systèmes. Une façon d'améliorer leurs opérations consiste à utiliser un algorithme d'optimisation, méthode qui allie la rigueur des mathématiques à la vitesse des ordinateurs. Toutefois, les responsabilités qui sont conférées à ces algorithmes rendent

nécessaire une vérification empirique de leur performance. Or, ces expériences sont problématiques puisqu'il faut recréer l'évolution stochastique dynamique du système à optimiser. Afin de palier à ce problème, ce mémoire se propose de créer un environnement expérimental qui facilite cette tâche. Un tel environnement permet de reproduire virtuellement les systèmes étudiés tout en permettant le parfait contrôle de leur dimension stochastique. De plus, la symbolique de modélisation offerte par cet environnement est éloquente et comprend l'alphabet courant des problèmes d'allocation de ressources.

CHAPITRE 2 Pourquoi simuler ?

Ce chapitre est consacré à l'introduction de la simulation et à la discussion de son potentiel comme méthode de validation des solutions d'un réseau stochastique dynamique (RSD). Plus particulièrement, deux avantages distincts de la simulation appliquée au domaine des transports seront mis en évidence. Par la suite, les inconvénients de la simulation seront énoncés. Pour terminer, le devis d'un outil de simulation pour les RSD sera formulé.

2.1 Le concept de simulation

La simulation consiste à utiliser un ordinateur pour reproduire les activités d'un système qui nous intéresse à partir d'un ensemble de règles qui semblent modéliser ses opérations. Il existe plusieurs raisons qui peuvent motiver l'usage d'une simulation. Les plus courantes sont l'amélioration de notre compréhension du système réel, la prédiction des comportements futurs des structures étudiées et l'évaluation de changements hypothétiques aux opérations du système original (Rothenberg 1986). Une autre raison qui peut justifier l'usage des simulations est leur capacité de permettre l'étude de situations extrêmes ou la manipulation d'objets fictifs (Sharma et Rose 1988).

La simulation a des applications diverses tels les jeux vidéo ou les intégrations de Monte Carlo. Toutefois, elle est principalement utilisée comme environnement expérimental dans le cadre d'une démarche scientifique rigoureuse. Dans ce rôle, elle procure à ses utilisateurs tous les bénéfices d'un environnement expérimental parfaitement contrôlable (McGeoch 1996). Rien ne se retrouve dans une simulation sans que, consciemment ou non, le scientifique ne l'y ait introduit avec son modèle. De plus, la simulation permet de contrôler le flot du temps dans une expérience. Il est par conséquent possible d'étudier des phénomènes très longs en temps accéléré ou d'analyser en des événements très brefs.

Par ailleurs, l'ordinateur employé pour diriger une simulation peut aussi être utilisé pour recueillir une quantité importante de statistiques sur les opérations du système virtuel. Ces informations sont souvent impossibles ou tout simplement trop coûteuses à obtenir réellement. Cet avantage a cependant pour conséquence de provoquer une confiance injustifiée dans les résultats d'une simulation en raison du faux réalisme que fait naître une surabondance de données. En effet, il est important de souligner que la simulation d'un modèle stochastique procure des renseignements sur une seule instance des différents mondes possibles qu'il décrit. Les résultats obtenus ont donc une validité limitée uniquement à ce monde. Une étude statistique des résultats de plusieurs simulations sur un échantillon des mondes possibles décrit par un modèle stochastique est donc nécessaire. Pour cette raison, les simulations sont généralement moins performantes quand elles sont utilisées comme méthode de résolution que lorsqu'elles sont utilisées comme environnement expérimental (Law et Kelton 1991).

2.2 Application de la simulation au domaine des transports

La simulation est une technique intéressante pour les chercheurs opérationnels travaillant sur des problèmes de transport car ces problèmes possèdent tous des composantes sociales, matérielles et économiques dont la détérioration est injustifiable. De plus, les problèmes de transport sont généralement trop complexes pour permettre une résolution à l'aide de méthodes analytiques conventionnelles. Dans le cadre de la recherche opérationnelle appliquée au domaine des transports, on peut donc redéfinir la simulation comme étant une façon de manipuler des choses ou des situations qui sont trop coûteuses, au sens propre ou figuré, pour être utilisées directement.

La simulation a des applications dans toutes les étapes de la méthodologie employée en recherche opérationnelle. Par exemple, on peut l'utiliser pour valider des modèles ou encore pour approximer numériquement des équations. Toutefois, ces applications ne sont pas particulières à la recherche opérationnelle. Pour cette raison, ce mémoire se limite à une application bien particulière de la simulation, uniquement utilisée par les chercheurs opérationnels, qui consiste à valider les solutions proposées à un RSD. Cette application est spécifique à la recherche opérationnelle car elle est une science dont les résultats sont principalement des politiques d'opération plutôt que des nombres. La performance d'une politique d'opération étant impossible à démontrer analytiquement, sauf dans les cas triviaux, une méthode de validation alternative est nécessaire et l'environnement expérimental que procure la simulation est un choix logique.

Un des avantages principaux de la simulation pour valider des solutions à un problème de transport, outre ceux mentionnés précédemment, est la possibilité de suivre le trajet d'une entité particulière dans le système virtuel (Williams et Khoubyari 1996). Contrairement aux autres méthodes, qui aggrègent les données du système en quantités plus aisément manipulables, la simulation, elle, les explose. Par exemple, on n'a qu'à penser comment, à partir du RSD d'un système, les simulations peuvent créer des instances ponctuelles des mondes qui y sont décrits.

Un bénéfice de cette façon de faire est que le trajet réel des entités dans le système virtuel devient disponible. La plupart des méthodes de résolution fournissent des informations moyennes sur les temps de transit, les délais et les coûts associés au déplacement des entités mais très peu peuvent fournir des mécanismes pour recueillir des renseignements précis sur les faits et gestes d'une entité particulière. Les problèmes de transport étant souvent étroitement liés aux parcours des entités dans un système, la simulation est donc une méthode particulièrement appropriée. De plus, la simulation donne la possibilité d'affecter à certaines entités des caractéristiques spécifiques, lesquelles permettent d'expérimenter l'effet d'une situation inhabituelle sur les opérations du système. De façon générale, la simulation est une technique adaptée aux problèmes où chaque entité est importante.

Une deuxième propriété intéressante de la simulation appliquée au domaine des transports est la préservation du phénomène appelé « l'effet de vague » (Fishburn et al. 1995). Les différents constituants d'un problème de transport doivent habituellement effectuer une séquence de tâches dans un ordre bien précis. Une pénurie de

ressources dans un secteur du système virtuel peut donc provoquer des perturbations qui, à leur tour, affecteront d'autres secteurs du système. De fil en aiguille, une onde de choc se propagera dans l'intégralité du système.

L'effet de vague est très difficile à modéliser car il implique des relations transitives en cascade entre les différents constituants d'un système. De plus, un effet de vague affecte autant les entités qui se trouvent en amont de son origine que ceux qui se trouvent en aval. Néanmoins, la simulation permet de reproduire sans disposition spéciale les conséquences d'un effet de vague en raison de sa structure unique. Dans une simulation, les lois régissant les interactions des entités entre elles ainsi qu'avec leur environnement sont souvent exprimées sous la forme de paires situation / réaction. Si la réaction d'une entité modifie la situation d'une autre, l'effet de vague se propage automatiquement et de façon réaliste, sans planification explicite ni travail supplémentaire de la part de celui qui code la simulation. Les effets de vague étant des phénomènes au coeur de plusieurs problèmes de transport, la simulation peut s'avérer une technique pertinente pour ce domaine d'activité.

2.3 Inconvénients de la simulation

Malgré tous ses avantages, le choix de la simulation comme technique de validation des solutions apportées à un RSD pose aussi quelques inconvénients. Le plus important de ceux-ci, car le plus insidieux, est la facilité avec laquelle les résultats d'une simulation peuvent être interprétés hors de leur contexte. Cet inconvénient est en réalité la conséquence de deux autres inconvénients de la

simulation, à savoir sa nature statistique et la programmation qu'elle requiert.

La nature statistique des simulations implique que leurs résultats doivent être rigoureusement analysés. De nombreuses répétitions des expérimentations effectuées, ainsi que la connaissance et l'usage de techniques de réduction de variance, sont absolument nécessaires pour assurer des résultats significatifs. De plus, les formules statistiques populaires présument souvent l'indépendance, hypothèse qui n'est habituellement pas vérifiée dans le cadre d'une simulation. Finalement, des outils logiciels défectueux, en particulier les générateurs aléatoires, peuvent altérer les résultats d'une façon inperceptible pour un utilisateur qui ne se méfie pas des biais qui peuvent alors être introduits.

Par ailleurs, la programmation nécessaire pour créer des simulations sur mesure est un travail qui est généralement très long et coûteux. Des erreurs de programmation anodines peuvent corrompre l'ensemble des résultats sans avertissement et, par conséquent, l'outil de validation doit donc être lui-même assujetti à une validation rigoureuse. De plus, la conception d'une simulation par des personnes dont les connaissances en programmation sont douteuses résulte souvent en des programmes inefficaces et très difficiles à vérifier. La simulation est une technique très puissante mais nécessite une grande discipline pour être utilisée correctement.

2.4 Devis d'un outil de simulation pour les RSD

Malgré la multitude d'outils de simulation disponibles, il n'en existe pas à notre connaissance qui répondent de façon satisfaisante

aux besoins spécifiques des chercheurs opérationnels qui désirent valider des solutions à des RSD. Les caractéristiques souhaitées d'un tel outil de simulation sont nombreuses. Elles seront maintenant énoncées pour ensuite être discutées plus en détails dans les chapitres subséquents. Entre autres :

1- L'outil de simulation fonctionne en temps discret et utilise le paradigme de simulation par événement. Ce choix découle entre autre de la nature active et ponctuelle des procédés d'optimisation. La vision par processus aurait pu être utilisée mais les complications d'ordre logiciel engendrées par ce choix n'ont pas été jugées indispensables.
2- L'outil de simulation est le prolongement d'un langage de programmation standard. De cette façon, on s'assure que l'outil de simulation soit flexible et extensible, attributs essentiels si on considère le particularisme des problèmes de transport. De plus, il sera ainsi toujours possible d'imbriquer un algorithme d'optimisation aux simulations développées. Finalement, les utilisateurs potentiels n'auront pas à apprendre un nouveau langage limité à la construction de modèles.
3- L'outil de simulation utilise le paradigme de programmation orienté-objet. De cette manière, les bénéfices inhérents à l'approche orientée-objet, tels une meilleure gestion de la complexité, la production de programmes de plus grande qualité et la possibilité de réutiliser des composantes de modèles, sont imposés implicitement aux utilisateurs. En outre, la notion d'objet permet une meilleure correspondance conceptuelle entre le modèle symbolique et digital.
4- L'outil de simulation met à la disposition des utilisateurs un ensemble complet de composantes génériques de simulation de

RSD. Cette caractéristique est essentielle afin de ne pas désavantager les utilisateurs occasionnels ou néophytes de l'outil. Néanmoins, ces composantes doivent pouvoir être utilisées comme base par les utilisateurs experts pour créer des composantes sur mesure.

5- L'outil de simulation permet la création de simulations efficaces. La simulation de problèmes de transport modernes peut taxer les ressources des ordinateurs les plus récents en raison de leur complexité. Des structures de données performantes doivent donc être utilisées et mises à la disposition des utilisateurs de façon à ne pas augmenter inutilement le temps nécessaire pour exécuter une simulation.

6- L'outil de simulation est statistiquement fiable et possède des générateurs aléatoires valides et performants.

La création d'un tel outil de simulation adresse cependant un problème fondamental en simulation. En effet, l'ajout de composantes qui favorisent la facilité d'utilisation dans un outil de simulation se fait habituellement au détriment de sa puissance de modélisation et vice versa. Or, le devis de l'outil de simulation mentionné précédemment stipule un environnement complet de simulations sous forme de structures prédéfinies et ce, malgré le nombre illimité de systèmes qui se qualifient comme étant des RSD. Les récents développements du paradigme orienté-objet, et plus particulièrement du langage C++, offrent cependant une solution possible à ce paradoxe.

Les méthodes orientées-objet sont le siège de recherches intensives en simulation. Par contre, l'essentiel de cette recherche est axé sur les avantages conceptuels de cette approche pour construire

des modèles fidèles à la réalité et intuitifs. C'est pourquoi ce mémoire étudiera les avantages logiciels mis à notre disposition par les mécanismes d'héritages et de patrons (*templates*) définis par le langage C++ (Stroustrup 1994) pour définir des modèles. La littérature foisonne d'études sur les bénéfices engendrés par l'utilisation des hiérarchies de classes pour définir des modèles mais très peu d'emphase est mise sur la capacité de telles hiérarchies à croître pour satisfaire les besoins présents et futurs des utilisateurs.

Dans ce mémoire, nous investiguerons la capacité du paradigme orienté-objet à satisfaire à la fois les critères de facilité d'utilisation et d'extensibilité liés à la réalisation d'un outil de simulation permettant d'évaluer les performances des solutions apportées à des RSD.

2.5 Conclusion

La simulation est une technique intéressante pour évaluer la performance des solutions produites par un algorithme d'optimisation puisqu'elle constitue un environnement expérimental parfaitement contrôlable. Elle permet de sonder individuellement chaque composante d'un système et sa structure préserve l'effet de vague. Ces deux aptitudes sont respectivement très utiles pour conduire des expériences formelles sur un algorithme d'optimisation et pour modéliser le contexte dynamique d'un système. Pour sa part, le contexte stochastique d'un système peut être déterminé avec les générateurs aléatoires de l'outil de simulation utilisé. S'ils sont bien conçus, ces générateurs permettent de reproduire à volonté un contexte stochastique particulier. Pour profiter de tous ces bénéfices de la simulation, le chercheur doit cependant programmer dans son outil de simulation les spécificités du système qu'il étudie. Finalement,

il ne faut pas oublier que les résultats produits par simulation sont des approximations des quantités réelles recherchées et doivent être traitées comme telles.

CHAPITRE 3 Revue de littérature

Ce chapitre a pour objectif de présenter une revue de littérature sur la simulation en général et les outils de simulation en particulier. Lorsque pertinents, les sujets abordés seront enrichis d'une discussion sur leur contribution à la simulation des réseaux stochastiques dynamiques (RSD). De plus, des outils de simulation typiques seront examinés.

3.1 Taxonomie des simulations

Il est pratique de classifier les simulations selon les caractéristiques des modèles qu'elles utilisent. Un standard *de facto* utilisé à cette fin dans la littérature concerne le rôle du temps dans le modèle. Dans les modèles en temps continu, le flot du temps est un agent actif de la simulation. Il influence directement les valeurs des paramètres d'intérêt du système. Habituellement, ces modèles font usage d'équations différentielles qui spécifient le taux de changement des variables importantes du modèle par rapport au temps. Par la suite, des méthodes numériques, telles celles d'Euler ou de Runge-Kutta, sont utilisées pour obtenir des solutions à ces systèmes d'équation. Des exemples de systèmes qui se modélisent en temps continu sont la pollution atmosphérique et le niveau d'un liquide dans un réservoir qui se vide.

Par opposition, le flot du temps dans un modèle en temps discret est un agent passif. L'état de ces modèles se modifie instantanément à des moments distincts qui sont fonction des activités des entités qui le composent. Par exemple, les banques sont souvent modélisées en temps discret car l'état de ces systèmes change uniquement lors de l'arrivée d'un client. Il est cependant important de souligner que cette classification n'est pas exclusive. Il existe plusieurs systèmes qui peuvent se modéliser selon l'une ou l'autre de ces approches. Le nombre et le niveau souhaités d'agrégation des variables étudiées sont les facteurs qui déterminent le choix de la technique approriée.

Dans le contexte de l'optimisation des problèmes de transport, il est naturel d'utiliser des modèles en temps discret. En effet, les variables d'intérêt de tels systèmes sont habituellement des temps de transit, des niveaux de stock ou des délais de livraison et ces quantités sont seulement définies lorsqu'une entité atteint un endroit spécifique du système. Pour cette raison, ce mémoire se consacrera dorénavant uniquement à ce type de modèle. Les personnes intéressées à approfondir le sujet de la simulation en temps continu ou des outils de simulation s'y rapportant sont invités à consulter (Cellier 1991) et (Piera et de Prada 1996).

Les outils de simulation en temps discret font l'objet d'une classification supplémentaire dans la littérature selon la méthode qu'ils utilisent pour permettre la description par l'usager des moments clés lors desquels se modifie l'état du système. Dans l'approche par événements, chacun de ces moments est caractérisé par une routine distincte indépendante. Du fait de cette indépendance, ces routines

doivent pouvoir déterminer les entités affectées par l'événement ainsi que planifier les événements accessoires et consécutifs qui lui sont associés. Conceptuellement, la simulation par événements se résume à l'exécution chronologique d'une chaîne d'événements ramifiés. Cette façon de faire a pour avantage de bien compartimenter les programmes écrits par l'utilisateur et de promouvoir la réutilisation, mais a cependant pour effet de créer du code très difficile à vérifier en raison de l'enchevêtrement des relations transitives entre les différents événements (Sharma et Rose 1988).

Une façon de palier à ce déficit nous est procurée par l'approche par processus. Cette approche consiste à regrouper dans un même ensemble les événements pertinents à la même entité. Plus précisément, on peut définir un processus comme étant une séquence chronologique d'événements qui décrivent les expériences possibles d'une entité dans le système (Law et Kelton 1991). Une simulation par processus consiste donc à exécuter concurremment mais en ordre chronologique les différentes séquences d'événements pertinentes aux processus présents dans le système. Cette approche est plus naturelle pour l'utilisateur car elle centralise les opérations d'une entité et par le fait même permet de mettre en application le principe « diviser pour régner » (Sanderson et al. 1991). Par contre, cette méthode nécessite l'utilisation de mécanismes spéciaux pour gérer la concurrence. Les deux approches ont toutefois recours aux mêmes structures internes et pour cette raison, l'expression simulation discrète est souvent employée pour désigner les simulations qui utilisent l'une ou l'autre de ces approches.

Les effets de l'approche par processus sur un outil de simulation étant surtout cosmétiques, l'approche plus directe par événements a

été retenue pour notre librairie. Il va sans dire que l'approche par processus possède de nombreux attraits pour les utilisateurs d'un outil de simulation mais puisque son implémentation ne contribuait pas directement aux objectifs de ce mémoire, elle a été ignorée. Idéalement, tout outil de simulation devrait offrir les deux approches (Schwetman 1996).

3.2 Perspectives de recherche en simulation discrète

La disponibilité sans cesse grandissante du matériel informatique en conjonction avec l'augmentation continue de la complexité des systèmes étudiés par les scientifiques a eu pour conséquence de populariser la simulation comme technique d'analyse. Dès lors, la simulation discrète a été appliquée à une variété de domaines. Les expériences ainsi accumulées sont la source de nombreuses nouvelles perspectives en simulation et cette section a pour but de les présenter.

Une observation universelle qui est faite par les utilisateurs de simulations est la lenteur du procédé. Les recherches en simulation se sont donc principalement concentrées sur ce problème. Un secteur où des gains potentiels de vitesse ont rapidement été identifiés est la liste d'événements. La liste d'événements est une structure essentielle en simulation discrète. Elle s'occupe de gérer tous les événements futurs planifiés dans une simulation. De nos jours, elle porte toujours le nom de liste d'événements mais utilise habituellement une autre structure de données. La raison en est simple, la gestion de la liste d'événements peut être tenue responsable pour jusqu'à 40% du travail nécessaire dans une simulation (Law et Kelton 1991). Les algorithmes et structures de données maintenant utilisés sont des applications des structures standards, tel les monceaux et les arbres, et leur description

peut être retrouvée dans tout bon livre d'algorithmique. Les articles de (Brown 1988), (Jones et al. 1986) et (McCormack et Sargent 1981) décrivent en détails la pertinence de ces structures pour la simulation. Il n'existe cependant pas de choix infaillible, la performance de la liste d'événements étant fonction de la taille de la simulation ainsi que de la dispersion temporelle des événements.

Dans le même ordre d'idées, les simulations parallèles et les simulations distribuées sont une autre façon d'augmenter la vitesse des simulations et constituent un domaine actif de recherche en simulation discrète. Elles se définissent respectivement comme étant l'exécution de programmes de simulation discrète sur des systèmes multi-processeurs ou en réseau. Les principaux défis de ces domaines consistent à diviser et à répartir équitablement le travail total représenté par une simulation pour, par la suite, synchroniser les différentes partitions. La synchronisation, problème commun pour tout ce qui est parallèle ou distribué, prend ici une dimension supplémentaire en raison de l'horloge interne maintenue par la simulation. De plus, la simulation parallèle ou distribuée nécessite la création de nouveaux outils de simulation qui permettent la manipulation de paramètres supplémentaires tels des numéros de processeur ou des adresses réseau. D'ailleurs, la croissance constante de Java, premier langage facilement compatible avec l'Internet, offre de brillantes perspectives pour la simulation distribuée. Les articles de (Fujimoto 1995) et (Nicol et Fujimoto 1994) décrivent l'état des recherches dans ces domaines.

La poursuite de la vitesse n'est cependant pas le seul objectif de recherche en simulation discrète. En effet, les simulations intelligentes, combinaison d'une simulation et d'un procédé

d'optimisation, représentent le futur de la simulation discrète. Elles permettent d'optimiser des systèmes complexes dont l'optimisation par des moyens analytiques prendrait des années. Le procédé d'optimisation est utilisé pour guider la simulation parmi les scénarios possibles, distingués par des valeurs différentes des paramètres à optimiser, et pour décider si les conditions d'arrêt sont satisfaites. Les recherches portent principalement sur des études quantitatives des performances relatives des différentes techniques d'optimisation disponibles, recherche qui pourrait aboutir éventuellement à la création d'un système expert en la matière. En effet, les techniques utilisées sont nombreuses, tels les algorithmes génétiques, les recherches tabou et les méthodes d'approximation stochastique, et aucune ne fonctionne efficacement pour tous les problèmes possibles. L'article de (Merkaryev et Visipkov 1996) peut être utilisé comme tremplin pour prendre connaissance des résultats de ces recherches.

Le domaine des transports est un domaine qui justifie à lui seul les sommes investies dans les recherches sur les simulations intelligentes. La complexité des problèmes de transport modernes nécessite souvent le recours à la simulation. L'article de (Fishburn et al. 1995) est une bonne introduction au domaine. Un numéro complet de la revue (*Simulation* août 1998) est consacré aux simulations des systèmes portuaires tandis que l'article de (Williams et Khoubyari 1996) présente un sommaire des considérations importantes lorsque vient le temps de modéliser des systèmes de transport routier. Un exemple d'application de la simulation au transport aérien se trouve dans (Cornett et Miller 1996).

Pour terminer, il faut souligner que les améliorations au concept de simulation discrète énumérées précédemment sont inutiles si elles

ne sont pas rendues accessibles au public. En conséquence, de nombreuses recherches sont dédiées aux outils de simulation, autant du point de vue de l'ergonomie que du point de vue fonctionnel. Ce sujet étant le principal point d'intérêt de ce mémoire, la section suivante sera utilisée pour le couvrir plus en détails.

3.3 Taxonomie des outils de simulation

Les outils de simulation peuvent être divisés en trois grandes familles : les simulateurs, les langages de simulation et les progiciels ou librairies de simulation. De plus, certains outils de simulation fusionnent les caractéristiques des simulateurs et des langages de simulation pour former une quatrième famille que nous désignerons sous le nom d'outils de simulation hybrides. À notre connaissance, il n'existe pas d'outil de simulation qui unifie les caractéristiques des trois familles principales.

3.3.1 Les simulateurs

Les simulateurs sont une alternative sans programmation à la simulation. Ils se concentrent sur les applications d'un domaine particulier pour offrir une très grande facilité d'utilisation. La définition des modèles se fait graphiquement à l'aide de structures prédéfinies qu'il suffit de sélectionner, d'agencer et de personnaliser en modifiant les valeurs des paramètres proposés. De cette façon, des personnes n'ayant aucune ou très peu de connaissances en programmation peuvent développer des simulations en très peu de temps. Par contre, les capacités de modélisation de tels outils s'avèrent rapidement insuffisantes pour étudier des problèmes un tant soit peu singuliers (Geuder 1995).

Les limites de l'utilité des simulateurs sont causées par la difficulté de gérer la complexité des diagrammes de flot de données représentant un système hautement interconnecté et par le choix fini de structures, de paramètres et de politiques d'opération disponibles. Pour cette raison, les simulateurs doivent être considérés comme des outils de prototypage plutôt que comme des outils de simulation. Il est néanmoins important de souligner que si un simulateur adéquat existe pour le problème précis qui nous intéresse, il serait insouciant de ne pas l'utiliser. Par contre, il est utopique de croire qu'un simulateur peut accomoder une classe de problèmes variés, telle celle des réseaux stochastiques dynamiques, sans posséder de mécanismes pour exprimer chacune des exceptions spécifiques aux différents systèmes constituant son domaine d'application. L'article de (Pidd 1992) et les références s'y trouvant offrent un excellent point de départ pour explorer plus en détails les avantages et inconvénients des simulateurs.

3.3.2 Les langages de simulation

Les langages de simulation, quant à eux, sont des compromis entre la facilité d'utilisation des simulateurs et la flexibilité des librairies de simulation. Ils sont faciles à utiliser car leur syntaxe et leur sémantique sont adaptées à la simulation. Par conséquent, ils nécessitent moins de lignes de code qu'une librairie de simulation pour exprimer plus clairement les mêmes concepts. Ce nombre réduit de lignes de code facilite par le fait même la vérification des programmes écrits. De plus, les langages de simulation exhibent partiellement la principale qualité des langages tout-usage, laquelle consiste à

permettre la modélisation d'un système quelconque, peu importe sa taille ou ses particularités.

Les langages de simulation souffrent cependant de plusieurs défauts. Ils sont beaucoup moins connus et disponibles que les langages tout-usage. De plus, les langages de simulation sont généralement des univers clos souvent incompatibles avec l'emploi libéral de composantes logicielles écrites avec d'autres langages de programmation. Le besoin existe pour des environnements de simulation ouverts (Schwetman 1996) et la simulation des réseaux stochastiques dynamiques nous le démontre car elle est souvent motivée par la validation d'un nouvel algorithme d'optimisation. Les personnes désirant en connaître plus sur le design des langages de simulation sont invités à consulter l'article de (Sanderson et al. 1991) et les références qui y sont données. GPSS/H et SIMAN constituent les langages de simulation les plus populaires et les articles de (Crain et Smith 1995) et (Profozich et Sturrock 1995) donnent un bon aperçu des capacités de chaque langage.

3.3.3 Les outils de simulation hybrides

Les outils de simulation hybrides utilisent un langage de simulation pour augmenter les capacités d'un simulateur. Le principal inconvénient des simulateurs s'en trouve ainsi éliminé. D'ailleurs, GPSS/H et SIMAN possèdent des interfaces de type simulateur qui peuvent être utilisées lorsque la pleine puissance d'un langage de simulation n'est pas nécessaire. Cependant, les outils de simulation hybrides éprouvent les même difficultés que les langages de simulation à communiquer avec des composantes logicielles étrangères.

3.3.4 Les librairies de simulation

Finalement, les librairies de simulation offrent une très grande liberté à leurs utilisateurs en sacrifiant les enjolivures. L'utilisation d'une librairie nécessite la connaissance du langage tout-usage utilisé ainsi que celle des expressions particulières à la librairie. De plus, des tâches qui ne devraient pas incomber aux modélistes, telle la gestion de la mémoire, ne sont généralement pas prises en charge automatiquement par le progiciel. Contrairement aux simulateurs, les librairies de simulation favorisent les programmeurs plutôt que les modélistes (Sanderson et al. 1991).

Entre bonnes mains, une librairie de simulation offre cependant un environnement de simulation complètement ouvert et très flexible. L'emploi d'une librairie n'est néanmoins justifié que lorsque les outils de simulation de plus haut niveau sont inadéquats car la charge de travail qu'elles exigent est très importante. La simulation des réseaux stochastiques dynamiques nécessite le recours à une librairie car elle couvre un domaine très vaste dont les spécificités ne peuvent être énumérées de façon exhaustive. De plus, son implémentation est souvent assujettie à des composantes logicielles extérieures dont les spécifications nous sont à priori inconnues. Les librairies de simulation seront étudiées plus en détail dans une section suivante et les références pertinentes y seront données au moment opportun.

Pour terminer cette section sur les outils de simulation, il faut mentionner l'article de (Swain 1995) qui présente une revue de plus de 50 outils de simulation et l'article de (Hlupic 1997) qui décrit une base de données liée à un système expert qui permet de guider les

consommateurs parmi les différents outils de simulation commerciaux disponibles.

3.4 Étude de cas

Cette section a pour objectif de présenter un outil de chacune des grandes familles d'outils de simulation de façon à bien cerner leurs particularités.

3.4.1 Les simulateurs : SLAM II

SLAM est utilisé comme acronyme pour *Simulation Language for Alternative Modeling* (Pritsker 1995). Son interface graphique est représentative des mécanismes utilisés par les simulateurs. Toutefois, pour être entièrement exact, SLAM II est un outil de simulation hybride car il met à notre disposition un langage de simulation. SLAM II supporte l'approche par événements et par processus. Cependant, son interface graphique est conçue pour développer des modèles utilisant uniquement l'approche par processus. En effet, la nature ponctuelle des événements rend difficile de les greffer de façon intelligible à un diagramme de flot. Les processus représentant chaque entité du modèle sont modélisés sous la forme d'un réseau. Chaque noeud de ce réseau correspond à une action de l'entité tandis que les branches représentent le passage du temps entre les actions. Par la suite, le modèle réseau est traduit dans le langage de simulation défini par SLAM II et exécuté. L'interface graphique de SLAM II est alors utilisée pour afficher une animation du modèle ainsi que des rapports statistiques illustrés.

Il est indéniable que cette façon de créer des modèles est très intuitive. Un modèle réseau permet de rapidement saisir les objectifs d'une entité ainsi que les points tournants qui influencent son comportement. De plus, des statistiques sont recueillies automatiquement pour chaque noeud qui le nécessite. Ces statistiques sont disponibles pendant que le modèle s'exécute et procurent, avec l'animation, un moyen infaillible de discerner les erreurs de logique dans le modèle.

Cette situation n'est cependant vraie que pour des modèles assez simplistes. Des actions comme le *jockeying*, action d'une entité qui change de file d'attente pour essayer de minimiser son temps d'attente, se modélise mal avec un modèle réseau. Il en va de même pour simuler la réaction d'une entité à un événement aléatoire extérieur. Ces actions nécessitent l'introduction de noeuds spéciaux qui surveillent l'arrivée d'une situation particulière, d'où une certaine lourdeur du modèle réseau. De plus, les noeuds disponibles ne couvrent pas la totalité des actions envisageables. Finalement, le recueil automatique des statistiques offre très peu de latitude pour implémenter des méthodes de réduction de la variance et utilise régulièrement des formules qui nécessitent l'indépendance des observations, hypothèse souvent non vérifiée.

3.4.2 Les langages de simulation : SIMAN / Cinéma

Le langage de simulation SIMAN, dont le nom provient de *Simulation Analysis,* est un bon exemple de langage de simulation. Il offre une syntaxe et une sémantique spécifiquement conçues pour les besoins des modélistes, ce qui est typique de ce type d'outil de simulation. SIMAN supporte l'approche par événements et l'approche

par processus ainsi que l'animation, avec sa version Cinéma. Par contre, SIMAN est fortement biaisé envers l'utilisation de l'approche par processus. SIMAN possède aussi de nombreux utilitaires qui permettent de l'utiliser comme un simulateur. Par conséquent, pour être parfaitement rigoureux, SIMAN est un outil de simulation hybride.

Les programmes SIMAN sont divisés en deux parties distinctes, soient la description du modèle et la définition des paramètres expérimentaux, lesquels résident dans des fichiers différents, tout comme la déclaration et la définition d'une fonction dans un langage tout-usage. La description du modèle est constituée des interactions des entités avec les ressources. Ces interactions font référence à des valeurs, telles des distributions ou des types, spécifiés dans le fichier des paramètres expérimentaux. Il est ainsi possible de changer ces paramètres sans avoir à recompiler le modèle. Une fois les résultats obtenus, SIMAN met à la disposition du modéliste une grande quantité de procédures statistiques.

Une caractéristique qui saute aux yeux quand on regarde un programme SIMAN est le petit nombre de lignes de code nécessaire pour exprimer un modèle. Par contre, on remarque facilement l'absence des instructions complexes telles *IF THEN*, *FOR*, *WHILE* et *CASE*. Les programmes écrits avec un langage de simulation sont concis et clairs mais les constructions syntaxiques disponibles sont nettement insuffisantes pour implanter, par exemple, une simulation distribuée. Les langages de simulation permettent de créer et de personnaliser les composantes d'une simulation pour qu'elles correspondent exactement à nos besoins mais il est néanmoins impossible de définir autre chose que des modèles.

3.4.3 YANSL

YANSL (Joines et al. 1996), *Yet Another Network Simulation Library*, est une librairie de simulation orientée-objet en C++ qui offre les mêmes caractéristiques que SLAM II et SIMAN, sans toutefois posséder les capacités graphiques de ces deux outils. La définition des modèles se fait pourtant de la même façon. C'est donc dire qu'elle consiste à choisir parmi un ensemble prédéfini de classes les noeuds et branches qui composeront le modèle réseau. Pour un modéliste, YANSL ressemble donc en tout point à un langage de simulation. Cependant, les constructions du langage sont des instructions C++ parfaitement légitimes, avec tous les bénéfices s'y rattachant. Ces bénéfices seront explorés dans la section suivante.

Pour résumer cette section, l'utilisation d'une librairie de simulation est aussi aisée et fonctionnelle que celle d'un langage de simulation. La principale différence est que ces derniers jouissent habituellement d'une quantité d'accessoires graphiques et statistiques particuliers à la simulation. Il est donc permis de conclure que pour un même niveau de gadgets, il est préférable d'utiliser une librairie de simulation puisque les capacités expressives de cette dernière ne sont pas limitées.

3.5 Les librairies de simulation et l'approche orientée-objet

La partition des librairies de simulation discrète selon un standard de qualité arbitraire permet de faire deux observations. La première est que la seule distinction possible entre les librairies de même classe est le paradigme de programmation utilisé. La deuxième est que si on grade les librairies de simulation d'une même classe

selon un indice de facilité d'utilisation quelconque, deux librairies de simulation utilisant le même paradigme n'auront pas nécessairement la même note. De ces faits, on peut tirer la conclusion que le choix du paradigme de programmation utilisé est un aspect important de la conception d'une librairie de simulation car il sera l'intermédiaire entre la librairie et les usagers. Cependant, le choix du meilleur paradigme disponible est futile si la structure de la librairie n'encourage pas l'utilisation effective des avantages qui en découlent (Ball et Love 1995).

Le choix et l'utilisation efficace des paradigmes de programmation sont des domaines de recherche actifs en simulation discrète. La raison en est simple, aucun langage de simulation ne possède la flexibilité des langages de programmation tout-usage modernes. De nombreux efforts sont donc faits pour simuler des langages de simulation avec des librairies. Les langages de choix sont des langages très populaires dont l'usage implique de façon inhérente l'utilisation des principes du génie logiciel. On cherche de cette façon à atteindre un public très large tout en minimisant les difficultés de programmation.

Les librairies de simulation en Modula-2 sont un bon exemple de librairies que n'ont pas suivi cette tendance. Des librairies de simulation telles SIMOD (L'Écuyer 1993) et HPSIM (Sharma et Rose 1988) ont misé uniquement sur la qualité intrinsèque des programmes de simulation écrits en Modula-2. En effet, les notions de modules et de types opaques ainsi que le support des co-routines qui facilite l'implémentation de l'approche par processus ont pour conséquence de permettre la création rapide de programmes de simulation de grande qualité. On entend ici par programme de qualité des programmes

facilement compréhensibles, représentatifs et dont les erreurs sont aisément identifiables. Les expériences personnelles de l'auteur avec SIMOD, lors des cours de simulation prodigués à l'Université de Montréal, lui ont prouvé l'affinité de Modula-2 pour la simulation. Hélas, Modula-2 a été un échec populaire, et ce, malgré le support académique dont il a fait l'objet en raison de ses nombreux avantages théoriques. En effet, l'inertie de l'informatique rend vain tout ce qui n'est pas compatible avec les standards officieux, c'est à dire les langages C et C++. Les librairies de simulation en Modula-2 sont donc devenus, par manque d'adeptes, des environnements de simulation clos.

La recherche sur les paradigmes orientés-objet n'a pas été confrontée à ce problème. En effet, les deux langages les plus populaires à l'heure actuelle, C++ et Java, utilisent cette approche. Des exemples de librairie de simulation discrète utilisant ces langages sont YANSL (Joines et al. 1996), CSIM18 (Schwetman 1996) et Silk (Healy et Kilgore 1998). De plus, un numéro complet de la revue (*Simulation* juin 1998) est consacré à des articles traitant de ce paradigme. Cette effervescence qui entoure l'approche orientée-objet n'est pas étrangère au fait qu'elle ait d'abord été introduite par un langage développé pour la simulation, Simula (Dahl 1966). Il semble donc qu'il soit enfin possible de réconcilier les besoins des modélistes et des programmeurs.

3.5.1 L'approche orientée-objet pour le modéliste

Du point de vue des modélistes, l'avantage principal qui découle de l'utilisation de l'approche orientée-objet concerne les gains en représentativité ainsi obtenus (Joines et al. 1996). En effet, il est

naturel de voir notre environnement comme étant composé d'objets. L'approche orientée-objet permet la création illimitée de types abstraits de données qui peuvent être spécifiquement construits pour modéliser le système à l'étude. Les programmes conçus de cette manière offrent alors une meilleure correspondance entre le modèle conceptuel et digital et sont par conséquent plus faciles à vérifier. De plus, le concept d'encapsulation des données sous-jacent à la notion d'objet est imposé aux utilisateurs, ce qui implique l'écriture de programmes de meilleure qualité. Par contre, la manipulation des objets par le modéliste signifie que ce dernier est responsable de l'allocation dynamique de la mémoire. Or, la gestion de la mémoire est le siège de nombreuses erreurs de programmation, même de la part de programmeurs chevronnés, et c'est une tâche dont le modéliste doit s'acquitter pour pouvoir disposer de la pleine puissance expressive du paradigme orienté-objet.

Un atout supplémentaire de l'approche orientée-objet pour les modélistes est constitué par les mécanismes d'héritage. En effet, il est possible, à l'aide de ces mécanismes, de construire un prototype grossier de simulation qui peut être raffiné par l'ajout successif de propriétés jusqu'à l'obtention du niveau de détails souhaités (Rothenberg 1986). Une contribution de ce mémoire au domaine de la simulation discrète consistera d'ailleurs à démontrer qu'il est possible de changer le niveau de détails d'une simulation d'une façon dynamique. Cette capacité est intéressante du point de vue de l'efficacité, pour les périodes de réchauffement par exemple, et du point de vue conceptuel, pour étudier de façon minutieuse des situations inhabituelles.

Finalement, les avantages du concept de réutilisation inhérent à l'approche orientée-objet sont amplifiés lorsque ce paradigme est appliqué à la simulation discrète. Un modèle peut souvent être décomposé en deux parties distinctes, le modèle du monde et le modèle d'intérêt. Le modèle du monde sert à simuler l'environnement du modèle d'intérêt. Par exemple, dans le contexte de l'optimisation des problèmes de transport, le modèle du monde s'occupe de simuler les variations des flots de véhicules sur les axes routiers tandis que le modèle d'intérêt s'occupe de déplacer une ambulance sur ces axes. L'approche orientée-objet permet de réutiliser les classes du modèle d'intérêt mais plus important encore, permet la création de librairies de mondes (Rothenberg 1986).

3.5.2 L'approche orientée-objet pour le programmeur

Contrairement aux modélistes, tous les bénéfices de l'approche orientée-objet pour les programmeurs sont assujettis à la structure même de la librairie. Il est donc de la responsabilité du concepteur de la librairie de s'assurer que la structure mise en place maximise les possibilités d'utilisation de ces bénéfices (Ball et Love 1995).

L'intérêt de l'approche orientée-objet pour le programmeur réside dans le fait qu'elle est la seule approche qui permet une extensibilité infinie sans nécessiter la modification des codes sources qui lui sont associés (Geuder 1995). En effet, les mécanismes d'héritage s'appliquent à toutes les classes, même celles qui composent le coeur d'une librairie de simulation. Par conséquent, ces dernières sont elles aussi modifiables. Cette capacité, en conjonction avec les fonctions virtuelles, permet d'assurer la personnalisation à volonté des classes de la librairie. La principale contribution de ce

mémoire sera d'exhiber une librairie structurée de façon à ne pas limiter cette capacité d'extension.

3.5.3 Langages orientés-objet

Le langage qui semble le plus approprié pour réaliser une librairie orientée-objet de simulation pour les réseaux stochastiques dynamiques est le C++ puisqu'il est standardisé, bien documenté et très utilisé. Au contraire, Java n'est pas un bon candidat car c'est un langage en constante mutation, ce qui rend incertain tout effort de développement. De plus, les expériences de l'auteur lui font douter de la précision de Java en matière de calcul numérique, aspect important pour la simulation et l'optimisation. Cette situation est malheureuse car Java supporte les co-routines pour l'implémentation de l'approche par processus et possède ses propres librairies graphiques intégrées. Un langage alternatif serait Smalltalk, qui se compare à C++ au niveau fonctionnel et qui le surpasse grandement au niveau de la facilité d'utilisation, mais la complexité de son environnement le rend très lent, défaut appréciable quand on songe au nombre d'opérations impliquées lors d'une simulation.

3.6 Conclusion

Notre survol des différents types de simulation et des outils associés nous a permis d'identifier le format d'une librairie de simulation discrète par événements comme étant le plus apte à satisfaire nos besoins. En effet, ce format constitue un environnement de simulation ouvert. En conséquence, il sera toujours possible d'y intégrer un algorithme d'optimisation ou les dernières innovations concernant les simulations intelligentes, parallèles ou distribuées. De

plus, on peut atténuer certaines difficultés traditionnelles reliées à l'utilisation d'une librairie de simulation en choisissant un langage orienté-objet pour la programmer. On satisfait ainsi les exigences en terme de représentativité et d'extensibilité de l'environnement expérimental que nous recherchons pour évaluer la performance des solutions apportées à des réseaux stochastiques dynamiques.

CHAPITRE 4 Contribution au domaine

Ce chapitre a pour but d'expliquer en termes généraux les innovations introduites par ce mémoire. Pour débuter, nous énoncerons les grands principes qui ont gouverné la conception de notre librairie de simulation. Par la suite, nous introduirons les concepts orientés-objet et analyserons en détails la problématique d'une librairie de simulation pour les réseaux stochastiques dynamiques ainsi que les procédés orientés-objet que nous avons utilisés pour la résoudre. Enfin, les divisions fonctionnelles de la librairie seront présentées et un survol des principales tâches assignées à chaque partie sera effectué. Pour ne pas surcharger le texte, les détails d'implémentation ne seront révélés qu'au chapitre suivant.

4.1 Philosophie de la librairie

L'objectif principal de la librairie de simulation présentée dans ce mémoire est de permettre la validation des solutions apportées à un réseau stochastique dynamique par un algorithme d'optimisation. En conséquence, une considération importante dans le design de notre librairie a été de nous assurer qu'elle pourrait interagir avec des algorithmes d'optimisation indépendants. Dans notre système, la communication entre une simulation et un algorithme d'optimisation

est assurée par une interface bidirectionnelle qui traduit l'état de la simulation en information utilisable par l'algorithme pour ensuite transformer les instructions de l'algorithme en événements de la simulation. Cette façon de faire permet d'utiliser la même simulation avec plusieurs algorithmes en changeant uniquement l'interface qui les unit. De plus, la mise en place de l'interface se résume souvent en pratique en un simple échange de fichiers. La figure suivante illustre la communication entre une simulation et un algorithme d'optimisation.

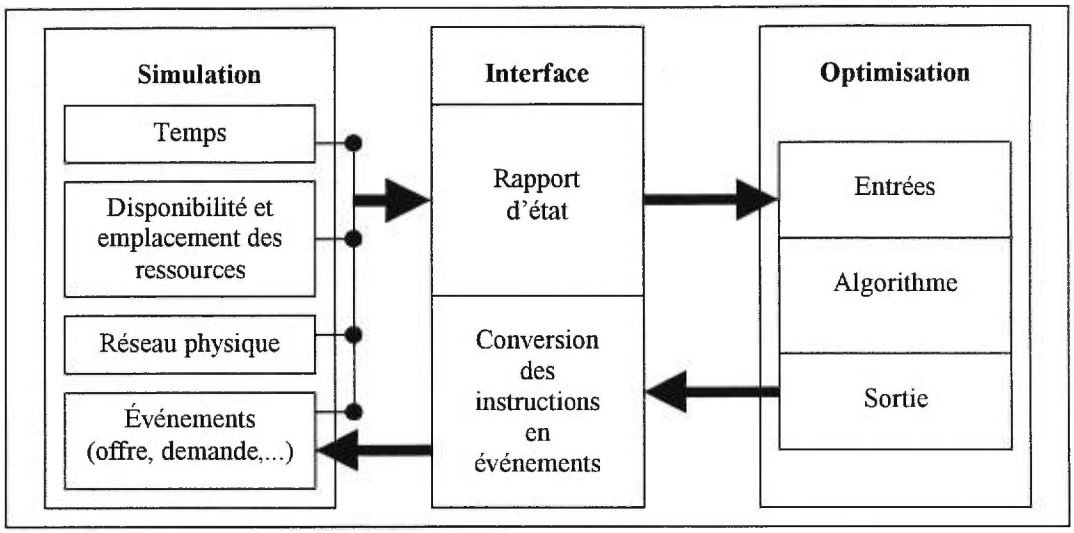

Figure 1 : Communication entre une simulation et un algorithme d'optimisation

Dans un autre ordre d'idées, les algorithmes d'optimisation sont habituellement développés par étape, simultanément avec les raffinements successifs apportés aux modèles des problèmes qu'ils sont appelés à résoudre. Pour cette raison, notre librairie a été spécialement conçue afin de permettre la création de simulations extensibles qui sont utilisables tout au long du cycle de développement d'un algorithme d'optimisation. La stratégie que nous employons est basée sur l'utilisation de composantes logicielles modulaires qui facilitent l'adaptation et la réutilisation des simulations. Chaque élément conceptuel d'une simulation d'un réseau stochastique

dynamique, les ressources ou les événements par exemple, est symbolisé dans la librairie par un type de données qui possède une interface standard. Il est alors possible de substituer deux éléments du même type en effectuant uniquement des modifications ponctuelles au code du programme. De plus, des mécanismes sont prévus pour permettre aux utilisateurs de la librairie d'étendre les capacités des types de données qu'elle propose au fur et à mesure que le modèle du problème simulé se complexifie. Le schéma suivant résume bien notre stratégie.

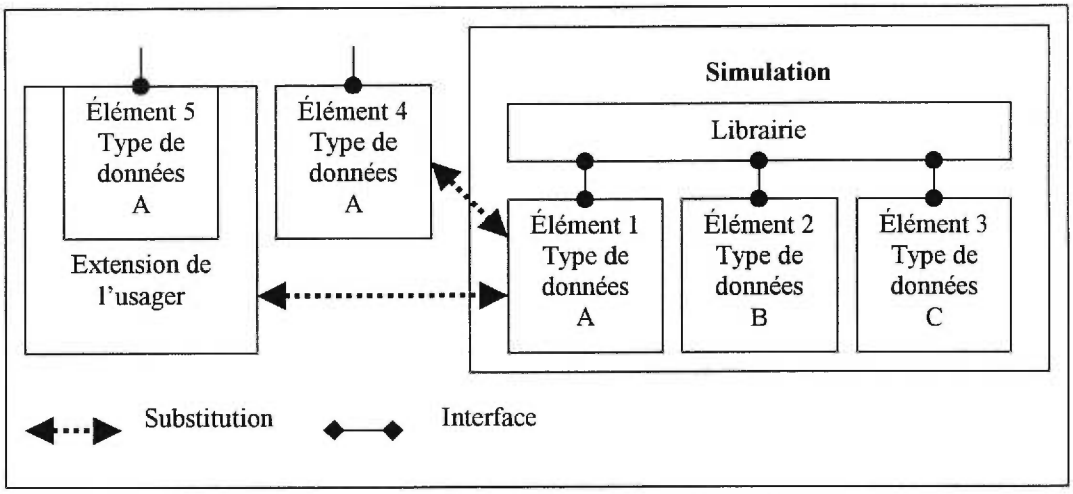

Figure 2 : Mécanismes d'extension de la librairie

Par ailleurs, il est important de mentionner que la faculté d'extension des simulations construites avec notre librairie constitue l'essentiel de notre contribution au domaine. Contrairement aux librairies standards, notre librairie offre à ses utilisateurs un ensemble de services dont les détails techniques lui sont facilement accessibles. Qui plus est, une procédure existe pour modifier ces services afin de créer une librairie qui répond exactement au besoin du moment. Par exemple, le générateur aléatoire de notre librairie peut être modifié par un utilisateur qui désire y ajouter une nouvelle distribution peu commune. En somme, l'idée maîtresse derrière notre librairie est que

si une simulation est une reproduction du réel, il faut nécessairement en accomoder la diversité.

4.2 L'approche orientée-objet

Le paradigme de programmation orienté-objet est articulé autour du principe que les données et les opérations qui les manipulent forment une partition conceptuelle élégante d'un programme. Chacune de ces partitions, dont le nom générique est classe, spécifie la représentation concrète des données relatives à une unité logique d'information dans un programme ainsi que les fonctions et procédures disponibles pour les administrer. Cette agrégation d'éléments primitifs d'un langage de programmation en unité fonctionnelle cohérente permet la définition ad hoc de nouveaux types de données complexes dont les instances dynamiques se nomment objets. Par exemple, il est pratique de définir une classe *employé* dans le contexte d'un programme de gestion des ressources humaines car il est alors possible de manipuler l'ensemble des attributs d'un employé, nom, fonction et salaire, avec une seule variable-objet.

Parallèlement à la notion de classe, le concept d'encapsulation des données fait partie intégrante de l'approche orientée-objet. Selon ce concept, un objet doit pouvoir être utilisé en faisant abstraction de l'implémentation de sa classe. Par analogie, on peut conduire une automobile sans avoir une connaissance approfondie de la mécanique. C'est pourquoi la définition d'une classe comprend une section privée, à l'intérieur de laquelle ses mécanismes internes sont définis, et une section publique, décrivant ses services. En conséquence, l'approche orientée-objet se prête bien à la création de composantes logicielles au fonctionnement complexe mais simple à utiliser, telle une classe qui

encode et déchiffre des chaînes de caractères. De plus, l'encapsulation des données permet de protéger une classe des interférences extérieures en limitant la visibilité du contenu de sa section privée à ses membres.

La fusion du concept de classes avec celui de l'encapsulation des données a pour résultat le mécanisme d'héritage, procédé orienté-objet qui permet la définition de classes qui sont des extensions à des classes déjà existantes (Sethi 1990). On peut illustrer ce concept avec l'exemple des classes *forme*, *triangle* et *carré*. La classe *forme* est un sous-ensemble des classes *triangle* et *carré* et peut donc être utilisée dans leurs définitions afin d'éviter les répétitions inutiles du même code. De plus, tout changement à la classe *forme* est alors automatiquement transmis aux classes *triangle* et *carré*. Dans la terminologie orientée-objet, la classe *forme* constitue la classe de base des classes dérivées *triangle* et *carré*. Une classe dérivée hérite de tous les attributs, variables et fonctions, de sa classe de base. Toutefois, une classe dérivée n'est évidemment pas considérée comme un membre de sa classe de base et ne peut donc pas accéder à la section privée de cette dernière. Cette situation limite l'utilité de l'héritage et c'est pourquoi la définition d'une classe contient une troisième section, la section protégée, dont l'accès est réservé à ses membres ou aux membres de ses classes dérivées.

Accessoirement aux mécanismes d'héritage, une classe peut définir des fonctions virtuelles, c'est-à-dire des fonctions dont la redéfinition est permise par une classe dérivée. On permet ainsi la création de classes de base flexibles mais fonctionnelles. Si on revient à notre exemple de la classe *forme*, la déclaration de la fonction « rotation » comme virtuelle laisse aux classes *triangle* et *carré* la

possibilité de la redéfinir afin de prendre avantage des particularités géométriques de ces formes. Dans le même ordre d'idées, les fonctions virtuelles pures permettent la déclaration de fonctions dont la définition est laissé aux classes dérivées. Par le fait même, on peut imposer des invariants à un ensemble de classes qui partagent une descendance commune. Toutefois, il ne peut exister d'objet d'une classe qui définit une fonction virtuelle, appelé classe virtuelle, car sa définition est incomplète. Un exemple permet de résumer ces concepts. La déclaration de la fonction « dessiner » comme pure virtuelle dans la classe *forme* oblige les classes *triangle* et *carré* à chacune spécifier leur propre définition de cette fonction. De plus, la classe *forme* n'est pas complète car aucune définition ne vient compléter la déclaration de la fonction « dessiner ». Il ne peut donc pas exister d'objet *forme* indépendant.

Pour terminer cette section, il faut souligner les patrons *(templates)*, procédé orienté-objet spécifique au langage de programmation C++. Un patron est un type de données paramétrique. En d'autres termes, un patron est une classe qui utilise des variables dont le type peut être spécifié uniquement lors de la création dynamique de ses objets. Un exemple éclatant de patron est la classe *liste*. Il peut être utile de mettre dans une liste des entiers, des caractères, des *employés* ou des *formes*. Plutôt que de définir quatre classes *liste* dont la seule différence est le type des objets contenus, il est pratique de définir le patron *liste*. Il suffit alors de spécifier la classe des objets à contenir lors de la création dynamique d'une liste pour obtenir l'objet désiré. Le principal usage des patrons est la création de classes récipient (*container*) dont le fonctionnement est indépendant du contenu.

4.3 L'approche orientée-objet au service de la simulation

Le paradigme de programmation orienté-objet offre au modéliste des types abstraits de données personnalisés de haut niveau qui permettent une bonne correspondance entre les modèles théoriques et les modèles digitaux. De plus, l'utilisation des classes facilite l'écriture de programmes de meilleure qualité par les modélistes en promulguant des concepts du génie logiciel tels l'encapsulation des données et la réutilisation. Cette réduction des contraintes d'ordre technique liées à l'utilisation d'un langage de programmation tout-usage permet la création de librairies de simulation dont la facilité d'utilisation est équivalente à celle des langages de simulation.

Comparativement aux langages de simulation, les librairies de simulation jouissent cependant d'un environnement de programmation beaucoup plus éloquent. Les constructions syntaxiques d'une librairie de simulation peuvent être utilisées dans un nombre potentiellement infini de programmes. Toutefois, cette flexibilité accrue n'est pas synonyme d'extensibilité. En effet, pour apporter des modifications ou des additions au contenu d'une librairie, il faut habituellement remanier son code source. Or, il n'est pas toujours souhaitable ou possible de permettre libre accès aux codes sources d'une librairie.

D'autre part, de nombreux champs d'activité, tel celui de la simulation des réseaux stochastiques dynamiques, englobent des problèmes variés dont les particularités ne peuvent être recensées de façon réaliste. Une librairie pour un tel domaine nécessite donc obligatoirement des capacités d'extension. Cette section a pour but de démontrer comment une utilisation judicieuse des mécanismes

orientés-objet définis par le langage C++ permet de réaliser une librairie de simulation discrète qui répond aux besoins des modélistes tout en étant extensible sans nécessiter de modifications aux codes sources de la librairie.

4.3.1 L'héritage et les fonctions virtuelles

On peut définir deux types d'extensibilité pour une librairie: l'extensibilité d'addition et l'extensibilité de modification. L'extensibilité d'addition est la capacité d'évolution d'une librairie. De façon similaire au monde naturel, cette propriété est assurée en C++ par les mécanismes d'héritage. La règle d'héritage de C++ stipule qu'une classe dérivée peut être utilisée partout où sa classe de base est attendue (Stroustrup 1994). La définition et l'utilisation d'un ensemble de classes de base complet par une librairie orientée-objet en C++ assure par conséquent l'extensibilité par addition d'une librairie. Dans le cas qui nous intéresse, les classes de base nécessaires correspondent aux éléments essentiels d'une simulation discrète d'un réseau stochastique dynamique, c'est-à-dire les événements, les entités, les ressources, les noeuds et les arcs.

L'héritage constitue également une composante essentielle aux mécanismes d'extensibilité par modification. L'extensibilité par modification consiste à redéfinir des éléments d'une librairie. La façon de permettre l'extensibilité par modification dans une librairie C++ repose sur l'utilisation des fonctions virtuelles. L'utilisation généralisée de fonctions virtuelles dans les classes qui constituent la fondation de la librairie permet aux utilisateurs de changer des fonctions prédéfinies sans avoir à toucher les codes sources de la librairie.

Un exemple permet de mieux cerner les avantages et inconvénients de l'héritage et des fonctions virtuelles. Le coeur d'une simulation discrète est la liste d'événements. La principale fonction de cette structure consiste à choisir parmi tous les événements planifiés celui qui doit maintenant être concrétisé. Par conséquent, tous les événements planifiés dans une simulation transitent à un moment ou à un autre par la liste d'événements.

Le premier obstacle qu'il convient d'adresser est la correspondance entre la définition d'événements exposée au chapitre précédent et le concept d'objet. Un événement est une routine qui décrit les changements qui se produisent dans le système lors de son avènement tandis qu'un objet est une instance d'un type abstrait de données. Or, les événements doivent être des objets si on désire utiliser l'héritage pour garantir l'extensibilité. Il faut donc définir pour chaque classe d'événements dérivée une fonction standard qui fait office de point d'entrée vers la routine associée à l'événement. La définition d'une fonction virtuelle dans la classe de base des événements nous permet d'imposer cette fonction standard à chaque classe dérivée.

De façon à faire son travail, la liste d'événements doit posséder des pointeurs vers chaque objet événement qui compose le futur de la simulation. Une façon simple de créer une liste d'événements consiste à prédéfinir la structure des événements. Cette manière de procéder facilite la déclaration de la liste d'événements mais fixe une fois pour toute l'information qu'ils peuvent véhiculer. Un autre dispositif envisageable nécessite l'utilisation des patrons (*templates*) de C++ pour permettre aux utilisateurs de choisir eux-mêmes la structure de données représentant un événement. Cette méthode impose

néanmoins un type unique à tous les événements de la simulation. L'approche des experts (*hackers*) quant à elle consiste à utiliser des pointeurs de type indéfini (*void*) pour créer une liste d'événements qui accepte plusieurs formats d'événements. En conséquence, tous les mécanismes de vérification des types de C++ sont rendus inutiles.

L'héritage permet d'enregistrer des événements dans la liste d'événements sans les inconvénients des autres techniques. En effet, la définition de la classe de base *événement* par la librairie permet aux utilisateurs de créer des classes d'événements dérivées qui sont toujours compatibles avec la liste d'événements. Cette compatibilité est assurée par le compilateur qui convertit automatiquement les pointeurs de classes dérivées en pointeurs de classes de base lorsque nécessaire. Cette conversion a cependant un effet secondaire ennuyeux. En effet, l'inverse de la règle d'héritage n'est pas vrai et le compilateur interdit donc toute référence aux membres d'une classe dérivée à partir d'un pointeur d'une classe de base. Par conséquent, même si l'information supplémentaire contenue dans un objet *événement* dérivé est physiquement disponible, elle devient logiquement inaccessible pour le compilateur. Dans le cas de notre liste d'événements, le point d'entrée de la classe de base sera utilisé plutôt que celui de la classe dérivée.

Une solution partielle à ce problème consiste à utiliser des fonctions virtuelles pures plutôt que des fonctions virtuelles simples. La définition par la librairie d'une classe de base virtuelle *événement* permet à la liste d'événements de manipuler des pointeurs virtuels. La conversion automatique du pointeur de classe dérivée en pointeur de classe de base a toujours lieu mais comme le compilateur sait qu'il ne peut exister d'objet d'une classe virtuelle, il maintient des informations

qui lui permettent de substituer la bonne définition à un appel de la fonction virtuelle pure.

Cette solution n'est toutefois pas complète. En effet, les usagers peuvent insérer un événement dans la liste et subséquemment vouloir l'en retirer pour effectuer des modifications. La liste d'événements fonctionne avec des pointeurs sur la classe de base *événement* et ne peut donc que retourner un tel pointeur lorsqu'interrogée. Il est donc nécessaire d'introduire dans toutes les classes de base de la librairie de simulation un champ *type* qui peut être utilisé pour déterminer la classe dérivée d'un objet de façon à pouvoir reconvertir son pointeur. Si ce champ n'est pas inclu, les mécanismes d'extensibilité décrits dans cette section se résument pour les usagers à retrancher aux classes dérivées les informations supplémentaires qu'elles définissent.

Il est certain que l'utilisation du champ *type* représente un travail supplémentaire pour l'utilisateur. Il offre toutefois des possibilités nouvelles comparativement au C++ traditionnel où ce concept est inconnu. Une utilisation vraisemblable consiste à changer dynamiquement le niveau de détails utilisé dans une simulation. En effet, il est facile de redéfinir la fonction virtuelle qui exécute les événements pour qu'elle retype dynamiquement ces derniers selon les valeurs du champ *type* et la valeur d'une variable de contrôle. En conséquence, il est possible de simuler avec précision un système qui atteint un seuil donné ou d'agréger une séquence d'événements inutile pour le moment.

4.3.2 Les patrons

Dans le cadre d'une librairie de simulation discrète extensible, la flexibilité des patrons peut être utilisée à bon escient. Il est de mise pour une librairie de simulation de qualité de prédéfinir des classes représentant les composantes courantes des simulations cibles de façon à offrir une facilité d'utilisation similaire à celle d'un langage de simulation. Dans l'optique de la simulation des réseaux stochastiques dynamiques, des exemples de telles classes peuvent être les classes *dépôts* et *camions*. Le fonctionnement fondamental de ces classes est assez général pour ne nécessiter que rarement une modification radicale par les mécanismes d'héritage et les fonctions virtuelles.

Pourtant, les types des variables internes manipulées par les classes prédéfinies peuvent faire obstacle à la modélisation d'un système particulier. Un exemple permet d'illustrer ce problème : supposons que la classe *dépôt* représente une ressource classique qui fait partie d'un réseau de location multi-points. Elle dispense son inventaire de ressources et recueille celles qui lui sont rapportées, peu importe leur provenance. Elle utilise à l'interne quatre types d'événements associés respectivement à l'atteinte des seuils d'inventaires minimum et maximum qui déclenche la procédure d'équilibrage des stocks, à la rupture de stock et à la notification des commandes satisfaites. Or, les informations convoyées par ces événements peuvent être insuffisantes pour les besoins précis d'un utilisateur. Il est alors essentiel de mettre à la disposition des usagers un mécanisme qui permet de modifier ces événements de façon à ne pas imposer une redéfinition de la classe *dépôt* chaque fois qu'une petite adaptation est nécessaire.

Peu importe le mécanisme choisi, il faut tout d'abord standardiser les communications entre les événements et la classe *dépôt*. En effet, chacun de ces événements convoit des informations essentielles différentes et doit par conséquent définir une procédure standard utilisée à l'interne par la classe *dépôt* pour enregistrer ces informations. Il est pratique d'employer à cette fin le constructeur de la classe, c'est-à-dire la procédure utilisée pour créer des instances de la classe. En conséquence, la seule exigence imposée sur la définition d'un événement destiné à remplacer un événement interne est la correspondance entre les constructeurs des deux classes.

Une méthode usuelle pour permettre la personnalisation des événements internes d'une structure par les usagers repose sur l'emploi de procédures qui associent à chaque type d'événement nécessaire un objet particulier. Il est important de noter que la procédure prend en paramètre un objet car il est illégal syntaxiquement d'utiliser un type. Une copie de cet objet est effectuée chaque fois qu'un objet de cette classe est nécessaire et les données essentielles y sont ensuite transférées. Cette technique est cependant inefficace pour un réseau qui englobe plusieurs ressources.

Les patrons permettent de définir la ressource *dépôt* sans utiliser ce dispositif. Chaque fois que l'utilisateur désire créer un objet de la classe *dépôt*, il doit paramétriser sa classe par le type des événements internes qu'il désire utiliser. Cette manière de faire a l'avantage de regrouper sur une seule ligne l'information pertinente à la ressource. Par contre, la notation nécessaire pour utiliser une ressource complexe qui utilise de nombreux événements peut devenir très lourde. Il est alors avantageux de dériver une nouvelle structure spécifique à chaque ressource.

La principale amélioration qui pourrait être apportée au concept des patrons est la définition de valeur par défaut. En effet, les structures prédéfinies avec des patrons obligent l'utilisateur à spécifier les paramètres qu'il désire utiliser, même si ce sont ceux qui ont été prévus originalement. Il est clair que les patrons, tels que définis en ce moment en C++, sont offerts pour créer des classes récipients plutôt qu'un mécanisme d'extension pour une librairie de simulation. Ils offrent toutefois un moyen élégant de personnaliser des structures prédéfinies.

4.4 Organisation de la librairie

Les mécanismes d'extensibilité décrits à la section précédente ont été mis en application dans une librairie orientée-objet pour la simulation des réseaux stochastiques dynamiques. La complexité de ces problèmes rend attrayant l'utilisation de la simulation comme technique d'analyse tandis que leurs grandes variétés requiert un environnement de simulation extensible. De plus, on doit avoir recours à la flexibilité du format d'une librairie puisqu'une intégration parfaite entre les programmes développés avec notre outil de simulation et les composantes logicielles qui motivent souvent les besoins de simulation des réseaux stochastiques dynamiques est requise. Un exemple typique d'une telle composante logicielle est un algorithme d'optimisation expérimental.

La librairie proposée est composée de huit collections de classes ayant chacune des fonctions spécifiques. Elles seront maintenant exposées à titre d'entrée en matière pour le chapitre suivant qui divulguera leurs détails d'implémentation.

4.4.1 Le groupe SIM

Le groupe fonctionnel SIM est le coeur de la librairie de simulation. Il contient la définition de la classe Simulation qui gère la liste d'événements. Bon usage est fait des paramètres par défaut pour définir des procédures qui permettent la planification d'événements avec ou sans priorité. De plus, les événements peuvent êtres planifiés en temps absolu ou relatif. Chaque insertion dans la liste d'événements retourne une notice d'événement qui peut ensuite être utilisée pour retracer, détruire ou modifier un événement particulier.

4.4.2 Les groupes EVENTS, ENTITIES et RES

Les collections de classes EVENTS, ENTITIES et RES définissent les classes de base associées aux trois composantes génériques essentielles de toute simulation, les événements, les entités et les ressources. Les entités sont des objets, au sens commun du terme, qui interagissent avec les ressources du système au gré des événements qui contrôle la simulation. L'ensemble des structures de la librairie utilise ces classes pour assurer la compatibilité future de la librairie avec tout ce qui peut en être dérivé. En conséquence, un champ *type* est disponible dans chacune de ces classes.

4.4.3 Le groupe NETWORK

Le groupe NETWORK définit les classes de base relatives aux noeuds et aux arcs d'un réseau stochastique dynamique. Les attributs habituels tels le coût ou la capacité d'un arc ainsi que le degré d'un noeud sont prédéclarés. De plus, le champ *type* est employé ici aussi

car ces deux classes sont ensuite utilisées comme fondation pour définir une troisième classe de base, qui représente un réseau. Des procédures pour construire et interroger un réseau sont mises à la disposition de l'usager.

4.4.4 Le groupe STATBLK et RANDGEN

Le groupe fonctionnel STATBLK offre des classes pour recueillir des statistiques en temps discret ou continu. Les statistiques en temps discret disponibles sont le nombre d'observations, les observations maximales et minimales ainsi que la moyenne et la variance. Les statistiques en temps continu recueillies sont l'intervalle de temps concerné, les observations maximales et minimales ainsi que la moyenne. La classe contenue dans RANDGEN, quant à elle, définit un générateur aléatoire qui possède plusieurs flots indépendants de variables aléatoires uniformes (L' Écuyer et Coté 1991). Ces variables peuvent ensuite être transformées en variables aléatoires de nombreuses distributions (Law et Kelton 1991) (Bratley et al. 1987) .

4.4.5 Le groupe DSTRUCT

Le groupe fonctionnel DSTRUCT contient les types abstraits de données les plus courants sous forme de patrons. Les listes, les monceaux et les queues sont disponibles.

4.5 Conclusion

Contrairement aux librairies de simulation orientées-objet actuelles qui exploitent uniquement les bénéfices de l'approche orientée-objet pour la modélisation, nous avons fait usage de cette

approche pour réaliser une librairie de simulation dont les structures prédéfinies sont extensibles. Cette faculté permet de créer un environnement expérimental spécifique aux problèmes d'allocation de ressources qui tient compte de la diversité de ces problèmes. La stratégie que nous avons utilisée repose sur l'utilisation combinée des mécanismes d'héritage et des fonctions virtuelles afin de permettre l'évolution des structures de la librairie ainsi que la redéfinition ponctuelle de ses composantes. De plus, notre emploi original des patrons autorise la création de structures modulaires dont les attributs sont interchangeables. Par contre, l'utilisation de ces mécanismes d'extension requiert l'introduction d'un champ type pour transtyper les objets généralisés par les règles d'héritage et une certaine dose de connaissances en programmation.

CHAPITRE 5 Détails d'implémentation

Ce chapitre explore le fonctionnement interne de la librairie de simulation décrite par ce mémoire. Les huit collections de classes qui la composent seront examinées succesivement et les particularités relatives à chacune seront mises en évidence.

5.1 Le groupe SIM

Le groupe SIM constitue la fondation de notre librairie de simulation. En effet, son membre principal, la classe *Simulation*, implémente une liste d'événements ainsi que l'interface permettant sa manipulation. En conséquence, on retrouve obligatoirement un objet de classe *Simulation* par programme qui utilise la librairie. D'ailleurs, cet objet, ou un pointeur y référant, est préférablement global à la simulation car il dispense des services, tel celui de l'horloge, qui peuvent être utiles à tous les objets accessoires y séjournant.

Une particularité évidente de la classe *Simulation* est sa déclaration sous forme de patron. Le paramètre exigé est la taille maximale de la liste d'événements qu'elle définit. La connaissance de ce paramètre permet d'allouer de façon statique la mémoire administrée par la liste d'événements, accélérant ainsi ses opérations. En contrepartie, il faut sélectionner une taille suffisament grande pour

ne pas que la simulation soit interrompue de façon impromptue par un débordement de la liste d'événements. La librairie possède d'ailleurs un message d'erreur spécifique pour identifier clairement cette situation. Le fait d'exiger ce paramètre n'est pas déraisonnable puisqu'une difficulté excessive à estimer de façon adéquate la taille de la liste d'événements trahit habituellement une faille importante dans la planification algorithmique d'un programme de simulation.

Le type abstrait de données choisi pour implémenter la liste d'événements est le monceau (*heap*). L'insertion et le retrait d'un événement dans une telle structure se fait dans O ($\in$ log n) (Standish 1994). De plus, l'algorithmique des monceaux se marie de façon très performante à l'utilisation de la mémoire statique sous forme de tableau. Chaque noeud du monceau contient un pointeur vers un événement ainsi que deux clés. La clé principale est le moment planifié auquel l'événement adjoint doit se concrétiser tandis que la clé secondaire est un indicateur de priorité. Cette dernière est exprimée par un nombre réel positif dont la valeur est inversement proportionnelle à la prééminence de l'événement, dix étant la valeur par défaut. La priorité zéro est réservée à l'usage interne de la librairie pour planifier des événements qui doivent être exécutés immédiatement. Les événements de même priorité sont traités de façon premier arrivé, premier servi (*FIFO*).

La liste d'événements de la classe *Simulation* est accesible aux utilisateurs par une interface dont le contenu peut être qualifié de standard. Il permet de planifier des événements en temps relatif ou absolu avec ou sans priorité. De plus, des procédures permettant de retrouver et modifier des événements insérés dans le monceau ainsi que d'interroger l'horloge de la simulation sont disponibles. L'interface

de la classe *Simulation* possède cependant une caractéristique particulière : elle est extensible. En effet, la classe *Simulation* dérive la totalité de son interface de la classe virtuelle *Sim*. Cette classe déclare virtuelle pure chacune des procédures définies dans la classe *Simulation*. En conséquence, chaque procédure qui compose l'interface de la classe *Simulation* est virtuelle et est dès lors candidate à une éventuelle extension. La redéfinition de la boucle d'exécution des événements de la classe *Simulation* pour accomoder la synchronisation de plusieurs objets de cette classe, prémisse de la simulation distribuée, est un exemple typique d'une telle extension. De plus, l'utilisation stricte de pointeur de type *Sim* par les membres de la librairie assure la compatibilité avec la classe *Simulation* ou ses classes dérivées.

5.2 Le groupe EVENTS

Dans le but de compléter notre description de la classe *Simulation*, la classe *Event* sera maintenant introduite. La classe *Event* est la classe de base obligatoire de tous les événements présents dans un programme qui utilise notre librairie de simulation. Il est donc opportun de la mentionner avant de discuter la boucle d'exécution des événements de la classe *Simulation*.

La boucle d'exécution de la classe *Simulation* consiste initialement à déterminer le moment exact où prendra place le prochain événement. Cette information est par la suite utilisée pour avancer l'horloge de la simulation et procéder à l'actualisation de l'événement imminent. Ce cycle est ensuite répété jusqu'à ce que la condition d'arrêt de la simulation soit atteinte. L'utilisation d'un monceau pour représenter la liste d'événements facilite le choix du prochain

événement à considérer puisque l'événement qui se trouve à son sommet est toujours le plus urgent. Par contre, l'actualisation effective des événements est quant à elle problématique car elle requiert des actions qui sont différentes selon le type et le contexte de l'événement manipulé. C'est pourquoi l'interface de la classe *Event* est composée de trois procédures virtuelles pures sans paramètre nommées EvtMain, FollowUp et CleanUp dont les définitions sont laissées à l'usager. Ces procédures sont appelées dans l'ordre par la classe *Simulation* chaque fois qu'un événement doit être concrétisé. La procédure EvtMain contient le corps d'un événement tandis que la procédure FollowUp sert à planifier les événements qui en découlent. La procédure CleanUp quant à elle sert de conclusion à un événement, sa tâche principale étant la désallocation de la mémoire utilisée par l'événement avant sa destruction.

Cette façon de faire, contrairement à une approche qui utilise une seule procédure, a pour avantage de présenter à l'utilisateur un schéma standard qui lui rappelle les étapes nécessaires pour définir complètement un événement. Par contre, l'utilisation de trois procédures où une seule est suffisante introduit inévitablement une pénalité au point de vue de l'efficacité de la liste d'événements. Toutefois, cette pénalité est minime puisque les procédures EvtMain, FollowUp et CleanUp n'utilisent aucun paramètre et ne retournent aucune valeur. Un appel de ces procédures est donc essentiellement équivalent à un saut dans le programme en langage machine de la simulation car leur environnement est prédéfini par leur objet parent et non pas alloué dynamiquement sur la pile d'activation des procédures.

Le fait d'astreindre les utilisateurs à définir des événements avec des procédures sans paramètre peut sembler une contrainte

fonctionnelle importante et ce, même si mise en perspective avec la simplicité et la flexibilité de la boucle d'exécution des événements ainsi possible. Il n'en est cependant rien. En effet, les procédures EvtMain, FollowUp et CleanUp sont des composantes intégrales de l'interface d'un objet dérivé de la classe *Event* et permettent par conséquent d'accéder à sa collection interne de données. Il n'est donc pas nécessaire de spécifier des paramètres aux procédures d'actualisation de l'événement car son environnement interne peut jouer le même rôle en fournissant les informations nécessaires pour inférer ou localiser les données externes nécessaires. Des pointeurs vers des objets de classes *Sim* et *Entity* sont prédéfinis à cette fin dans la classe *Event*.

5.3 Le groupe ENTITIES

Les entités sont les objets qui interagissent avec les ressources du système au gré des événements qui contrôlent la simulation. Elles sont caractérisées par un ensemble d'attributs dont l'étude est souvent la raison d'être d'un programme de simulation. La valeur de ces attributs est déterminée en majeure partie par le dénouement des différents événements auxquels une entité est assujettie. En conséquence, il est essentiel de toujours pouvoir identifier les entités soumises à un événement afin de maintenir à jour l'état du système. C'est pourquoi la classe *Event* possède un pointeur vers un objet de classe *Entity*, classe de base des entités dans la librairie.

Contrairement aux classes précédentes, la classe *Entity* ne possède pas de procédure virtuelle. En effet, même si on restreint le domaine d'application de la librairie à la simulation des réseaux stochastiques dynamiques, une entité est un concept trop nébuleux pour posséder une déclaration fonctionnelle unique. Il est donc vain

de tenter de prédéclarer des procédures à ce niveau de détails et ce, malgré le fait que des attributs génériques, telle la cargaison, sont déjà identifiables. Ce paradoxe s'explique par la diversité des paramètres nécessaires aux procédures manipulant de tels attributs. Par exemple, le chargement d'un avion doit prendre en considération le poids et le volume des objets transportés afin que la cargaison ne déséquilibre pas l'appareil tandis que le contenu d'un train se divise en wagon ou conteneur dont seul le type est important.

En raison de cette situation, le rôle de la classe *Entity* dans la librairie se résume à offrir un type par lequel les classes entités dérivées par les usagers peuvent être incorporées dans les structures de la librairie. En conséquence, l'interface de la classe *Entity* offre très peu de procédures. Celui-ci est principalement composé des procédures permettant d'accéder aux variables *Type_id* et *LifeTime*. La variable *Type_id* est le champ type qui permet de convertir un objet de classe *Entity* en sa classe dérivée tandis que la variable *LifeTime* est utilisée pour calculer le temps de séjour d'une entité dans la simulation. Le chapitre suivant, qui contient des exemples d'utilisation de la librairie, présente des classes dérivées de la classe *Entity* qui représentent des entités spécifiques.

Finalement, on ne peut passer sous silence l'existence des règles syntaxiques en C++ pour déclarer et définir des procédures à paramètres variables. Des exemples typiques de telles procédures sont *printf* et *scanf*. L'utilisation de procédures virtuelles à paramètres variables permet de définir la classe *Entity* beaucoup plus précisément. Par contre, dans l'état actuel des choses, le travail requis de l'utilisateur pour définir une telle procédure éclipse les bénéfices qu'il peut en tirer. Les perspectives offertes par ce type de procédure sont

cependant intéressantes et d'éventuelles extensions en ce sens méritent incontestablement investigation.

5.4 Le groupe RES

Les ressources sont les composantes d'une simulation qui peuvent provoquer des délais indésirables dans les opérations d'une ou plusieurs entités. Elles procurent une grande variété de services aux entités selon diverses politiques d'opérations. Dans le but de permettre la définition par les usagers de classes ressources sur mesure compatibles avec la librairie, le groupe RES définit la classe de base virtuelle *Res*. De plus, le groupe RES contient les classes EvtRes, QueueRes et PriorRes qui sont respectivement des ressources sans file d'attente, avec file d'attente et avec file d'attente priorisée.

La classe Res définit les attributs standards d'une ressource, c'est-à-dire un identificateur, son niveau d'inventaire, sa capacité maximale ainsi qu'une valeur seuil qui indique un besoin de réapprovisionnement. De plus, des statistiques sont conservées sur le nombre de transaction qui y sont effectuées. Lorsqu'approprié, des statistiques sur la file d'attente de la ressource sont aussi conservées. Il est important de remarquer qu'un objet de classe *Res* gère un seul type de ressource. La classe *ResList* est mise à la disposition des usagers pour manipuler un groupe de ressources. L'identificateur d'une ressource est alors utilisé pour la retrouver dans la liste.

L'interface de la classe *Res* est constitué des procédures nécessaires pour manipuler les attributs énumérés précédement, les deux plus significatives étant Deliver et Return. Elles servent respectivement à retirer ou à remettre des unités de ressources dans

l'inventaire d'un objet de classe *Res*. Cet inventaire est représenté par un objet de la classe *MergeList*, liste dont les éléments, des objets de classe *MergeUnit*, sont composés d'un identificateur et d'un nombre. Si une *MergeUnit* possédant un identificateur déjà existant est inséré dans une *MergeList*, les deux éléments semblables sont fusionnés en un seul en additionnant leur valeur. Les plus observateurs se demandent sûrement déjà la raison de l'utilisation d'un objet *MergeList* dans la classe *Res* puisqu'elle ne gère qu'un seul type de ressource. La raison en est simple, il a été décidé d'harmoniser les transactions entre les entités et les ressources en désignant les *MergeUnit* comme monnaie d'échange commune. De cette façon, l'utilisateur n'a pas à gérer individuellement les unités de ressources accaparées par les entités et celles-ci peuvent aisément manipuler plusieurs types de ressources à la fois en les conservant dans un objet *MergeList* privé.

Une caractéristique particulière de la classe *Res* est sa dérivation de la classe *Entity*. En effet, la théorie de la simulation discrète sépare nettement les concepts d'entités et de ressources. Néanmoins, nous avons décidé de traiter les ressources comme une spécialisation du concept d'entité car les mécanismes d'héritage garantissent ainsi une plus grande cohésion dans la structure de la librairie. De plus, il est alors permis de définir des ressources qui instiguent des événements sans créer d'exception dans la logique de la librairie. Par exemple, le pointeur de type *Entity* de la classe *Event* peut être utilisé pour désigner une ressource.

La classe *EvtRes* est une application pratique de la relation entre les classe *Entity* et *Res*. Elle représente une ressource simple, sans file d'attente, qui instigue automatiquement des événements lorsque son

seuil de réapprovisionnement est atteint où lorsqu'une entité tente de déposer une quantité de ressources qui outrepasse sa capacité. Pour ce faire, la classe *EvtRes* est définie comme étant un patron dont les paramètres sont les types des deux classes d'événements associées à ses réactions automatiques. De cette façon, la classe EvtRes peut aisément être personnalisée sans nécessiter sa redéfinition. Toutefois, le constructeur des classes d'événements spécifiées doit répondre à un certain format afin de permettre la transmission des informations pertinentes. Les classes QueueRes et PriorRes fonctionnent selon le même principe de réponse automatique que la classe EvtRes mais avec la différence que les ressources ainsi définies possèdent une file d'attente. En conséquence, le patron de ces classes nécessite la déclaration de deux types d'événements supplémentaires, respectivement associé à un débordement de la file d'attente et à la réactivation d'une entité en attente. Ces classes démontrent qu'il est possible de prédéfinir des composantes spécifiques d'une simulation sans néanmoins sombrer dans la particularisation à outrance et ce, sans pénalité pour l'usager.

5.5 Le groupe NETWORK

En raison de l'emphase qui a été mise sur la flexibilité et l'extensibilité dans la conception de notre librairie, les classes décrites précédement sont assez génériques pour permettre la simulation d'un grand nombre de systèmes. Or, un des objectifs de ce mémoire est la création d'une librairie pour la simulation des réseaux stochastiques dynamiques. En conséquence, les classes *Link*, *Node* et *Network* ont été créés spécifiquement pour ce type d'application.

Il existe de nombreuses structures de données pour représenter un réseau de manière digitale. Celle qui a été retenue est la liste d'adjacence (Standish 1994). Selon cette méthode, un réseau est une liste de noeuds dont chaque élément contient une liste d'arcs. L'approche orientée-objet se prête bien à la création de cette architecture puisqu'une classe peut être définie pour agglomérer les données de chacun de ses constituants. Un encodage plus performant aurait pu être utilisé mais le contexte exploratoire de la simulation nécessite performance et flexibilité, deux considérations difficilement réconciliables. Une liste d'adjacence implémentée avec des pointeurs virtuels permet d'accomoder des noeuds et des arcs de format extensible sans toutefois sacrifier complètement l'aspect efficacité.

Un avantage supplémentaire des listes d'adjacence pour la simulation des réseaux stochastiques dynamiques est l'interconnexion des données qu'elle permet. Cette caractéristique est importante car la simulation des réseaux stochastiques dynamiques est souvent motivée par la validation d'un nouvel algorithme d'optimisation. La fusion d'un tel algorithme avec la librairie implique un transfert de données entre les deux systèmes, échange qui peut être initié par des événements à portée locale ou globale. En conséquence, l'ensemble des données nécessaires à un algorithme d'optimisation doit être disponible à partir de l'information parfois limitée contenue dans un événement. Or, l'architecture doublement chaînée d'une liste d'adjacence permet de la parcourir dans sa totalité et ce, peu importe le point d'entrée utilisé dans sa structure.

Le mécanisme choisi pour transmettre des informations entre la librairie et une composante qui lui est étrangère est l'échange de fichiers. Des alternatives plus rapides, telles la communication inter-

processus ou l'appel direct de procédure, ont été laissées de côté en raison du volume de données à transmettre ainsi que des conversions nécessaires. De plus, cette manière de procéder correspond aux entrées et sorties standards de la majorité des algorithmes d'optimisation. De façon à permettre la création en série de fichiers résumant les informations pertinentes sur l'état d'une simulation, les classes *Event*, *Entity*, *Res*, *Link* et *Node* définissent toutes l'opérateur de sortie standard de C++ («). Le résultat de l'application de cet opérateur est toutefois fonction de la classe effective de son opérande. En conséquence, il est parfois nécessaire d'utiliser l'attribut type défini dans chaque classe de la librairie pour obtenir l'information désirée.

La classe *Network* représente une liste d'adjacence dans notre librairie. Étant donné le rôle central des objets de cette classe, la classe *Simulation* contient un pointeur vers un objet de classe *Network* ainsi que les procédures nécessaires pour y permettre libre accès par tous les objets de la simulation. La classe *Network* contient une liste de noeuds ainsi que l'interface permettant la gestion du réseau, c'est-à-dire ajouter, retirer, et localiser des arcs, des noeuds et des entités (ressources). Dans la tradition de la librairie, toutes les procédures de l'interface sont virtuelles pour permettre des extensions et la totalité de ses attributs sont sujets aux mécanismes d'héritage. Une procédure est disponible pour récursivement parcourir et imprimer des informations sur tous les membres du réseau.

La classe *Node* symbolise les noeuds d'un réseau dans la librairie. Les attributs qu'elle définit sont les degrés d'entrée et de sortie du noeud (In-degrees,out-degrees), la liste des objets ressources locaux disponibles, la liste des entités présentes ainsi que la liste des

arcs sortant. La classe *Link* quant à elle représente les arcs d'un réseau. Fait à remarquer, les objets de classe *Link* sont des arcs unidirectionnels. La classe *Network* possède une procédure pour créer des arcs bidirectionnels en créant deux arcs unidirectionnels réciproques. Les attributs de la classe *Link* indiquent son point de départ et d'arrivée, son poids, ainsi qu'une liste d'entités en transit.

5.6 Le groupe RANDGEN

Comme son nom l'indique, un réseau stochastique dynamique évolue dans le temps selon le dénouement de phénomènes aléatoires. La simulation d'un tel système nécessite donc obligatoirement un mécanisme pour déterminer la valeur de ses paramètres stochastiques. À cette fin, la librairie définit la classe *RandGen*, classe qui contient un générateur aléatoire multi-flots, ainsi que les procédures nécessaires pour obtenir des valeurs assujetties aux lois de probabilité communes.

La définition de la classe *RandGen* est basée sur un générateur aléatoire déjà existant écrit en C (L' Écuyer et Coté 1991). Les seules modifications apportées ont été l'ajout d'une enveloppe C++ autour du code C original. Après discussion avec l'auteur du générateur, les seuls tests effectués ont consisté à établir la correspondance entre les nombres produits par les deux versions, la preuve de la distribution aléatoire de ces derniers ayant déjà été faite. Les procédures qui transforment une variable aléatoire de distribution uniforme sur le domaine [0,1] produite par le générateur en variable aléatoire d'une autre distribution ont été testées par inspection visuelle des histogrammes correspondant à cinq échantillons de taille 5000. Chaque échantillon provenait d'un flot différent parmis les 100 mis à la disposition des utilisateurs.

5.7 Le groupe STATBLK

Les fruits d'une simulation sont des statistiques nombreuses et variées qui permettent d'évaluer la performance du système simulé ou de mieux comprendre son fonctionnement. De façon à faciliter le recueil de ces statistiques, notre librairie définit les classes *DiscStatBlock* et *TimeStatBlock* pour recenser respectivement des valeurs discrètes et continues. Ces classes sont dérivées d'une classe de base virtuelle commune, la classe *StatBlock*.

La classe virtuelle *StatBlock* se veut un standard pour permettre la définition de classes statistiques par les usagers. Elle définit comme pures virtuelles les fonctions Log et Report, fonctions qui servent respectivement à rapporter une observation et à déclencher la création d'un rapport sur le bloc statistique. Il existe un très grand nombre de métriques et de techniques pour interpréter des données statistiques et il est crucial que les utilisateurs de la librairie puissent utiliser celles qui sont appropriées au problème étudié. Le mandat de la classe *StatBlock* est donc de fournir un type et une interface commune par lequel des blocs statistiques aux mécanismes différents peuvent être interchangés.

La classe *DiscStatBlock* permet de recueillir des statistiques sur des phénomèmes discrets. Les informations disponibles sur une séquence d'observations sont sa taille, les observations minimales et maximales ainsi que la moyenne et la variance. Il est important de mentionner que le calcul de la moyenne et de la variance présume l'indépendance de la séquence d'observations. La classe *TimeStatBlock* se spécialise quant à elle dans l'étude des variations

d'un attribut par rapport au temps. Les statistiques disponibles sont l'intervalle de temps sur lequel l'attribut a été examiné ainsi que sa valeur minimale, maximale et moyenne.

5.8 Le groupe DSTRUCT

Le groupe DSTRUCT est composé de toutes les structures de données utilisées par les membres de la librairie. Plus précisément, le groupe DSTRUCT contient des classes relatives a l'implémentation d'une liste, d'un monceau et d'une queue. Elles sont définies sous forme de patron de façon à pouvoir être utilisées par les utilisateurs de la librairie peu importe les conformations de leur contenu éventuel. La plupart des compilateurs C++ mettent à la disposition de leurs usagers une librairie de patrons standards pour les structures de données les plus courantes mais il a été jugé imprudent de fonder la librairie sur du code sur lequel aucun contrôle ne pouvait être exercé.

5.9 Conclusion

Ce chapitre a présenté en détails chacun des groupes fonctionnels qui composent notre environnement de simulation. Un premier fait saillant de cette présentation est l'utilisation d'une classe et de trois procédures virtuelles pures afin de créer un schéma standard d'événements qui soit intuitif pour les usagers tout en étant extensible. Une autre particularité de la librairie est l'utilisation de patrons pour créer des classes ressources qui instiguent automatiquement des événements lorsque certaines conditions sont satisfaites. Plus généralement, l'utilisation stricte de fonctions virtuelles ainsi que de pointeurs vers les classes de base définies dans la librairie permet de prédéfinir toutes les composantes nécessaires à la

simulation des problèmes d'allocation de ressources et ce en garantissant la compatibilité avec la librairie des classes que les usagers pourraient éventuellement en dériver.

CHAPITRE 6 Exemples d'utilisation

Ce chapitre est voué à la présentation de divers exemples qui démontrent l'utilisation des mécanismes particuliers de la librairie de simulation orientée-objet présentée dans ce mémoire. Pour débuter, nous introduirons un problème emprunté à la littérature ainsi que sa solution à l'aide d'un programme conçu avec la librairie. Ce programme sera ensuite employé comme point de départ pour présenter comment les stratégies d'extension de la librairie permettent d'implémenter efficacement la technique de réduction de la variance par nombres aléatoires communs. Par la suite, une méthode pour modifier dynamiquement le niveau de détails d'une simulation en utilisant une hiérarchie de classe et le champ type sera explorée. Pour terminer, nous exposerons un programme qui illustre la communication entre une procédure externe et notre librairie afin de simuler les opérations passées d'un réseau stochastique dynamique réel.

6.1 Prélude

La démonstration des particularités de notre librairie suppose que ses généralités sont connues. Or, depuis le début de ce mémoire, aucun exemple concret de programmes utilisant notre librairie de simulation n'a été produit. Pour palier à cette situation, nous allons employer la librairie pour résoudre un problème provenant de la

littérature. Plus précisément, nous allons déterminer à l'aide de la simulation si une banque doit acheter un guichet automatique de marque Zippytel ou deux de la marque Klunkytel (Law et Kelton 1991). Les coûts associés à l'achat, l'installation et la maintenance d'un guichet Zippy sont le double de ceux associés à l'achat d'un guichet Klunky. Par contre, le service d'un client prend deux fois moins de temps sur un Zippy que sur un Klunky. Les coûts pour la banque étant les mêmes peu importe l'alternative choisie, la question qui nous intéresse est de savoir quelle configuration minimise le temps d'attente des clients.

Les clients se présentent à la banque selon un processus de Poisson de taux 1 par minute. En conséquence, l'intervalle de temps entre deux arrivées est une variable aléatoire de distribution exponentielle dont la moyenne est 1. Le service d'un client à un guichet Zippy prend un temps variable de distribution exponentielle de moyenne 0.9 minutes tandis que les guichets Klunky ont des temps de services distribués selon une loi exponentielle de moyenne 1.8 minutes. Chacune des deux configurations possède une file d'attente unique. La mesure de performance de chaque système est le temps d'attente moyen des 100 premiers clients servis par un Zippy ou un Klunky, dénotée respectivement par $d_z(100)$ et $d_k(100)$. Initialement, le système est vide.

Le protocole de simulation que nous allons utiliser est le même que celui utilisé par Law et Kelton (1991, pp. 584) afin de pouvoir comparer les résultats obtenus. Il consiste à réaliser n simulations indépendantes pour chacune des configurations et à calculer la moyenne des délais moyens obtenus. Plus précisément, si x_{1j} est la moyenne des 100 temps d'attente pour la configuration Zippy lors de

la simulation j et x_{2j} est la moyenne des 100 temps d'attente pour la configuration Klunky lors de la simulation j, le domaine de j étant 1,2...n, alors $\bar{x}_1(n)$ et $\bar{x}_2(n)$ sont respectivement la moyenne des x_{1j} et des x_{2j}. La comparaison de $\bar{x}_1(n)$ et $\bar{x}_2(n)$ nous permet de déterminer la configuration la plus performante avec une marge d'erreur inversement proportionnelle à n.

Le point de départ de la simulation de ce système est un événement, appelons-le *ClientArrival,* qui symbolise l'arrivée d'un client dans la banque. Cet événement crée un objet de classe *Customer*, un client, qui tente de capturer une unité de ressource. Si une unité de ressource est disponible, le client est servi immédiatement et la fin de son service, un événement nommé *ClientDone*, doit être planifié. Un client qui trouve toutes les ressources du système occupées entre dans une file d'attente. Il est réactivé ultérieurement par un événement *StopWaiting* instigué par la ressource lorsqu'elle se libère. L'unique fonction de *StopWaiting* est de planifier un événement *ClientDone* pour le client qui quitte la file d'attente. Indépendamment du sort de son client associé, la dernière action de *ClientArrival* est de planifier l'arrivée du prochain client, amorçant ainsi un nouveau cycle du système. La simulation se termine quand l'événement *ClientDone* détermine que son client associé est le centième à avoir pénétré dans la banque, la discipline FIFO de la file d'attente garantissant que tous les temps d'attente ont été comptabilisés. Le programme suivant, réalisé avec notre librairie, implémente cette stratégie :

```
#include "sim.h"                        // inclure le nécessaire: liste d'événements
#include "events.h"                     // événements
#include "entities.h"                   // entités
#include "res.h"                        // ressources
#include "randgen.h"                    // générateur aléatoire

enum Config { ZIPPY, KLUNKY };          // Définir un type pour chaque
typedef enum Config Config;             // configuration possible

int NbRepetition = 20;                  // Nombre de répétitions des simulations
int NbClient = 1;                       // Condition d'arrêt NbClient == 100
Config Configuration = ZIPPY;           // type de la simulation en cours
Simulation<10> *SimZippy;               // Taille de la liste d'événements 10
Simulation<10> *SimKlunky;              // Taille de la liste d'événements 10

RandGen *Generator;                     // générateur aléatoire

// Les ressources sont des objets de classe QueueRes. Ces objets planifient
// automatiquement des événements quand le seuil de commande est
// atteint, quand la ressource déborde, quand la queue déborde et quand une entité
// est réactivée de la file d'attente. Seul ce dernier événement est nécessaire.

class StopWaiting;                      // Prédéclaration

QueueRes<NullEvent,NullEvent,NullEvent,StopWaiting> *Zippy;
QueueRes<NullEvent,NullEvent,NullEvent,StopWaiting> *Klunky;

// La classe ClientDone est l'événement executé quand un client quitte
// la banque

class ClientDone :  public Event        // Dérivé de la classe Event
{
  Customer *Client;                     // Pointeur vers le client source

  public :

  ClientDone(Customer *who) {           // Constructeur
     Client = who;
  }

  EvtMain () {                          // Corps de l'événement

    // Le client libère la ressource.

    if (Configuration == ZIPPY)
        Zippy->Return(Client->Put("teller",1), Client);
    else
        Klunky-> Return(Client->Put("teller",1), Client);
  }
```

Figure 3 : Zippy versus Klunky

```
  FollowUp () {                          // Planification des événements

        // Vérifier si la condition d'arrêt est atteinte.

    if (Client->GetID() == 100)
      if(Configuration == ZIPPY)
        SimZippy->Stop(SimZippy->Now());
      else
        SimKlunky->Stop(SimKlunky->Now());
  }

  CleanUp () {                           // Code de sortie
    delete Client;                       // Détruire le client
  }
};


// La classe StopWaiting représente la réactivation d'un client dans la file
// Cet événement est planifié automatiquement par les ressources. La classe
// Served est un modèle prédéfini pour les événements de ce type.

class StopWaiting : public Served        // Dérivé de la classe Served
{
  float ServiceTime;                     // Temps de service
  ClientDone *ServiceOver;               // Événement à planifier

  public :

  StopWaiting(MergeUnit *what, Customer *who, Sim *Engine) :
    Served(what,who,Engine) {            // Utilise le constructeur de Served
    ServiceTime = 0;                     // Initialisation
  }

  EvtMain() {                            // Corps de l'événement

    Who->Take(What);                     // Le client accapare une unité de ressource

        // Générer le temps de service

    if (Configuration == ZIPPY) ServiceTime = Generator->Expon(0.9,1);
    else ServiceTime = Generator->Expon(1.8,1);

        // Créer l'événement de fin de service

    ServiceOver = new ClientDone(Who);
  }
```

Figure 3 : Zippy versus Klunky (suite)

```
    FollowUp () {                          // Planification des événements

      if (Configuration == ZIPPY)
        SimZippy->IncSchedule(ServiceTime, ServiceOver);
      else
        SimKlunky->IncSchedule(ServiceTime, ServiceOver);
    }

    CleanUp () {
          // Rien à faire. Doit tout de même être définie car pure virtuelle.
    }
};

// La classe ClientArrival représente l'arrivée d'un client dans
// la banque.

class ClientArrival : public Event          // Dérivée de classe Event
{
    int client_number;                      // 1 à 100
    Customer *Client;                       // Le client
    float ServiceTime;                      // Temps de service du client
    ClientDone *ServiceOver;                // Planifier la fin du service

    public :

    ClientArrival (int id) {                // Constructeur
          client_number = id;               // Numéro du client
          Client = new Customer(id);        // Création du client
          ServiceTime = 0;                  // Initialisation
    }

    EvtMain () {                            // Corps de l'événement

       if (Configuration ==  ZIPPY)
          if (Client->Take( Zippy->Deliver(1, Client))) {      // Demande 1 unité
             ServiceTime = Generator->Expon(0.9,1);            // Service immédiat
             ServiceOver = new ClientDone(Client);             // Événement fin service
          }
       else
          if (Client->Take(Klunky->Deliver(1,Client))) {       // Demande 1 unité
             ServiceTime = Generator->Expon(1.8,1);            // Service immédiat
             ServiceOver = new ClientDone(Client);             // Événement fin service
          }
    }
```

Figure 3 : Zippy versus Klunky (suite)

```
    FollowUp () {                                    // Planification des événements

        // Planifier la fin du service du client, si nécessaire, et la prochaine
        // arrivée dans la banque.

      if (Configuration == ZIPPY) {
        if (ServiceTime) SimZippy->IncSchedule(ServiceTime, ServiceOver);
        SimZippy->IncSchedule(Generator->Expon(1,0), new ClientArrival(id+1));
      }
      else {
        if (ServiceTime) SimKlunky->IncSchedule(ServiceTime, ServiceOver);
        SimKlunky->IncSchedule(Generator->Expon(1,0), new ClientArrival(id+1));
      }
    }

    CleanUp () {
        // Rien à faire. Doit tout de même être définie car pure virtuelle.
    }
};

// Programme principal

void main () {

    DiscStatBlock *Zrepmean;      // La moyenne des résultats des répétitions
    DiscStatBlock *Krepmean;      // La moyenne des résultats des répétitions
    float Zdelay = 0, Kdelay = 0; // Délais moyen d'une répétition
    int n,i;

    Zrepmean = new DiscStatBlock("Delais moyen Zippy");
    Krepmean = new DiscStatBlock("Delais moyen Klunky");


    Generator = new RandGen ();   // Créer un générateur aléatoire


        // Créer les ressources

    Zippy = new QueueRes <NullEvent,NullEvent,NullEvent,StopWaiting>
            ("Zippy","teller", 1, 1,-1,FIFO,0,SimZippy);
    Klunky = new QueueRes <NullEvent,NullEvent,NullEvent,StopWaiting>
            ("Klunky","teller", 2, 2,-1,FIFO,0,SimKlunky);

    for (n = NbRepetition; n > 0; n—) {       // Faire NbRepetiton répétions

      NbClient = 1;
      Configuration = ZIPPY;                  // Nous simulons le Zippy
      SimZippy = new Simulation<10> ();       // Création de l'objet Simulation

      SimZippy->AbsSchedule(0,new ClientArrival(NbClient));
      SimZippy->Start();
```

Figure 3 : Zippy versus Klunky (suite)

```
        Zdelay = Zippy->GetDelay();          // Quel est le temps d'attente moyen ?
        Zrepmean->Log (Zdelay);              // L'ajouter aux statistiques
        Zippy->Reset();                      // Réinitialiser la ressource
        delete SimZippy;

            // Nouveau germe pour chaque générateur utilisé

        Generator->InitGenerator(0,NewSeed);
        Generator->InitGenerator(1,NewSeed);

        NbClient = 1;
        Configuration = KLUNKY;              // Configuration Klunky
        SimKlunky = new Simulation <10>(); // Création de l'objet simulation

        SimKlunky->AbsSchedule(0,new ClientArrival(NbClient));
        SimKlunky->Start();

        Kdelay = Klunky->GetDelay();         // Quelle est le temps d'attente moyen ?
        Krepmean->Log (Kdelay);              // L'ajouter au statistique
        Klunky->Reset();                     // Réinitialiser la ressource
        delete SimKlunky;

            // Nouveau germe pour chaque générateur utilisé

        Generator->InitGenerator(0,NewSeed);
        Generator->InitGenerator(1,NewSeed);

    }

    Zrepmean->Report();
    Krepmean->Report();
}
```

Figure 3 : Zippy versus Klunky (suite)

Les valeurs théoriques de $d_z(100)$ et de $d_k(100)$ rapportées dans Law et Kelton (1991, pp.583) sont de 4.13 minutes et 3.70 minutes. La banque doit donc choisir d'acheter deux guichets Klunky. Toutefois, en raison du faible écart entre les valeurs théoriques de $d_z(100)$ et de $d_k(100)$, les résultats obtenus par simulation se révèlent trompeurs. En effet, des expériences réalisées par Law et Kelton (1991, pp. 585) sur 100 répétitions du schème de simulation énoncé précédement ont révélées que 52% ($n=1$), 38% ($n=10$) et 34% ($n=20$) des simulations recommandent la mauvaise configuration. La reprise de ces expériences avec le programme conçu avec notre

librairie produit des résultats similaires, 49% (n = 1), 40% (n = 10) et 35% (n = 20) des simulations produisant une recommandation erronée.

Une caractéristique manifeste du programme présenté est la programmation structurée résultante de l'utilisation de l'approche orientée-objet et de l'obligation de définir les procédures EvtMain, FollowUp et CleanUp pour chaque événement. Par contre, on peut aussi remarquer que le programme produit est d'une taille disproportionnée comparativement à celle du problème. Cette situation est causée par le ratio très élevé qui existe entre le nombre de lignes de codes structurelles et le nombre de lignes de codes effectives du programme. Ce problème disparaît quand la librairie est utilisée pour simuler des systèmes plus complexes, tels les réseaux stochastiques dynamiques, qui bénéficient au maximum des avantages de la modularisation.

Une seconde caractéristique de notre programme de simulation est la correspondance stricte entre chaque classe définie et un élément conceptuel de la simulation. En conséquence, un objet doit être créé pour chacune des instances distinctes d'un élément de la simulation. Cette façon de faire est moins performante que l'approche traditionnelle à la vision par événement en ce qui a trait à la gestion de la mémoire car chacun des événements est un objet plutôt qu'un appel de procédure. En conséquence, la situation la plus dégénérée est une simulation qui comporte de nombreux éléments actifs très simples qui instiguent de multiples événements parallèles, ce qui maximise tous les défauts de la librairie. Néanmoins, en vertu de l'accroissement constant de la mémoire disponible dans les ordinateurs et de la complexité des problèmes à résoudre, les avantages de l'approche orientée-objet dépassent de beaucoup ses inconvénients.

6.2 Implémentation de la technique CRN

Dans le but d'améliorer les résultats ambigus produits par le programme de la section précédente, Law et Kelton (1991) conseillent d'employer la technique des nombres aléatoires communs (*Common Random Number* ou CRN). Cette technique de réduction de la variance repose sur le principe qu'une comparaison est toujours plus significative quand les éléments à comparer ont été obtenus dans les mêmes conditions expérimentales. Dans une simulation, les conditions expérimentales sont déterminées par les nombres aléatoires utilisés et c'est pourquoi la technique CRN propose de les utiliser conjointement dans chaque simulation homologue d'un système.

Il est important de souligner que pour implémenter la technique CRN correctement il ne faut pas seulement réutiliser les mêmes nombres aléatoires. En effet, il faut aussi que chacun d'entre eux soit utilisé de la même façon. Plus précisément, si X_{sj} est la j^e répétition de la configuration s d'un système et X_{tj} est la j^e répétition de la configuration t du même système, alors un nombre aléatoire r détermine le même paramètre stochastique du système dans X_{sj} et X_{tj}. Une façon rapide de synchroniser les problèmes à discipline FIFO stricte, comme le problème du Zippy et des Klunky, est d'utiliser des flots (*stream*) de nombres aléatoires distincts pour chaque source de hasard dans la simulation. Cependant, le simple fait d'utiliser deux files d'attente pour la configuration Klunky rend cette méthode inopérante. En général, la technique CRN est impossible à appliquer intégralement à un système complexe puisque l'ordre des événements dans la liste d'événements est lui-même une variable aléatoire.

Une méthode infaillible pour synchroniser deux simulations homologues consiste à lier les nombres aléatoires employés à des jalons qui permettent d'établir la correspondance entre les phénomènes stochastiques communs. Dans le problème du Zippy et des Klunky, ces jalons prennent la forme des numéros entre 1 et 100 assignés aux clients selon l'ordre dans lequel ils se présentent à la banque. En conséquence, étant donné que le problème contient deux sources de hasard, une manière simple d'assurer la synchronisation repose sur le stockage de chaque nombre aléatoire utilisé dans la première simulation dans un tableau de 2 rangées et 100 colonnes. Le tableau est ensuite employé comme générateur aléatoire de la deuxième simulation, chaque cellule du tableau étant caractérisée par un numéro de client et le paramètre stochastique qu'elle permet de déterminer. Cette façon de faire a cependant deux désavantages. Premièrement, puisque les références au générateur aléatoire sont disséminées dans le code du programme, il y aura répétition inutile des instructions pour sauvegarder et retrouver les nombres aléatoires utilisés. Deuxièmement, la quantité de variables aléatoires à enregistrer est proportionnelle à la longueur de la simulation.

Grâce aux mécanismes d'extension de la librairie, nous allons implémenter la technique CRN selon le schème précédent sans aucun de ses inconvénients. Pour ce faire, nous allons ajouter une procédure au générateur aléatoire qui sauvegarde et réutilise automatiquement les variables aléatoires exponentielles qu'elle produit. De plus, nous allons redéfinir la classe *Simulation* de façon à ce qu'elle administre deux simulations en parallèle. De cette façon, la taille du tableau nécessaire pour stocker les variables aléatoires est fonction de la différence maximale entre les numéros de deux clients présents simultanément dans le système puisque leur arrivée est coordonnée

dans les deux simulations homologues. Les expériences de la section précédente ont révélé que le nombre maximal de clients observés dans une file d'attente a été de 19. Pour plus de sécurité, nous allons utiliser un tableau de 3 rangées et 30 colonnes pour conserver les nombres aléatoires, une économie de 55% comparativement à la méthode avec deux simulations séries. La rangée supplémentaire est utilisée pour signaler une désynchronisation du tableau en raison d'une réutilisation trop hâtive d'une cellule.

La première étape du processus d'implémentation consiste a dériver la classe *GenCRN* de la classe *RandGen*. *GenCRN* définit l'attribut TabVA, un tableau de 3 rangées et 30 colonnes servant à sauvegarder les nombres aléatoires utilisés. Initialement, les cellules des deux premières rangées ont la valeur −1, ce qui signifie qu'elles sont inoccupées, tandis que celles de la troisième rangée ont la valeur zéro. La première rangée de TabVA est consacrée aux temps inter-arrivée tandis que la deuxième est dédiée au temps de service. Chaque fois qu'une référence est faite à une cellule de la deuxième rangée, pour sauvegarder ou retrouver un temps de service, la cellule de la troisième rangée qui partage la même colonne voit sa valeur augmentée de un. De cette façon, on peut vérifier si une variable aléatoire a été utilisée deux fois et détecter une désynchronisation accidentelle. Les colonnes correspondent au numéro du client, modulo 30. TabVA est géré par la procédure ExponCRN qui produit des variables aléatoires de distribution exponentielle selon le numéro du client demandeur, la raison de la demande, la moyenne de la distribution échantillonnée et le générateur choisi. Ce dernier paramètre est ignoré quand le TabVA contient déjà le nombre aléatoire demandé.

```
        // Type spécialisé pour identifier un nombre aléatoire qui va être utilisé comme
        // temps inter-arrivée ou temps de service.
        // IMPORTANT : les enum sont remplacés par le compilateur par des int.
        // ARRIVAL = 0 et SERVICE = 1

enum RequestType = { ARRIVAL, SERVICE };
typedef enum RequestType RequestType;

        // Générateur aléatoire spécifique pour implémenter la technique CRN

class GenCRN : public RandGen  {        // Classe dérivée de la classe RandGen

  protected:                            // Participe à l'héritage

        // Pour sauvegarder les nombres aléatoires. Les deux premières rangées sont
        // initialisées à –1, la dernière à 0. ( code d'initialisation omis )

  double TabVA[3][30];

  public:

        // Cette fonction retourne un nombre aléatoire de distribution exponentielle de
        // moyenne Beta. Les paramètres clientID et Request servent à placer ou
        // retrouver un nombre aléatoire dans la bonne cellule.

  double Expon (int clientID, RequestType Request, float Beta, gen g) {

    int Generate;                       // Est-il nécessaire de générer un nombre ?
    double result;                      // Une variable aléatoire expon. de moyenne béta

        // Si la cellule dans la troisième rangée de la colonne de ce client contient un
        // nombre pair, il faut générer un nombre. Generate = 1 si pair.

    Generate = ! (( TabVA[ 2 ] [ clientID % 30 ] ) % 2);

        // Si la cellule visée n'est pas libre et il faut générer un nombre, Erreur!!!

    if  ( Generate && ( TabVA[ Request ] [ clientID % 30 ]  != -1 )) {
       cerr << "Error: CRN out of sync.";
        exit(1);
    }

    if  (Generate) {  // S'il faut générer un nombre
       TabVA [ Request ] [ clientID % 30 ] = Uniform01(g);  // Uniform01 hérité de RandGen
    }
```

Figure 4 : Générateur aléatoire avec CRN

```
        // Créer la variable aléatoire exponentielle de moyenne Beta
        result = -Beta *  ( log (1 - TabVA [ Request ] [ clientID % 30 ]  ) );
        // Libérer la cellule si nécessaire
        if  (!Generate)  TabVA [ Request ] [ clientID % 30 ] = -1;
        // Incrémenter la variable de synchro
        if (Request == SERVICE) TabVA [ 2 ] [ clientID % 30 ]++;
        return result;
    }
};
```

Figure 4 : Générateur aléatoire avec CRN (suite)

La classe *GenCRN* a pour avantage de centraliser la gestion des nombres aléatoires communs au même endroit. De plus, la synchronisation atteinte est robuste. Cependant, il ne faut pas oublier que la taille réduite du tableau circulaire de nombres aléatoires utilisés par *GenCRN* est assujettie à l'hypothèse de la gestion simultanée d'une paire de simulations homologues. Pour satisfaire cette hypothèse, la classe *SimCRN* est dérivée de la classe *Simulation*. Elle contient deux listes d'événements ainsi que deux horloges. Sa boucle d'exécution actualise alternativement un événement de chaque liste. En conséquence, les procédures de planification des événements de *Simulation* doivent être redéfinies pour accepter un paramètre qui spécifie la liste d'événements à utiliser.

```
class SimCRN : public Simulation <SzEvtLst> {
    protected :
    EvtList<SzEvtList> ZippyEvtList;        // Liste d'événements sim. Zippy
    EvtList<SzEvtList> KlunkyEvtList;       // Liste d'événements sim. Klunky
    double ZippyTime;                       // Horloge sim. Zippy
    double KlunkyTime;                      // Horloge sim. Klunky
```

Figure 5 : Zippy versus Klunky avec CRN

```
int FlagEndZippy;                                   // End the Zippy sim
int FlagEndKlunky;                                  // End the Klunky sim

public:

SimCRN () {                                         // Constructeur
    ZippyTime = 0;
    KlunkyTime = 0;
    ZippyEvtList = EvtList<SzEvtList> ();
    KlunkyEvtList = EvtList<SzEvtList> ();
    FlagEndZippy = 0;
    FlagEndKlunky = 0;
}

void Start () {                                     // Boucle d'exécution
    while(!FlagEndSim) {

            // Exécuter un événement de la simulation Zippy

       if (!FlagEndZippy) {
          ZippyTime = ZippyEvtList.ReturnPKey();    // Mettre à jour l'horloge
          CurrentEvt = ZippyEvtList.Remove();       // Extraire l'événement
          CurrentEvt->EvtMain();                    // Exécuter le corps de l'événement
          CurrentEvt->FollowUp();                   // Planification des événements
          CurrentEvt->CleanUp();                    // Ménage
          delete CurrentEvt;                        // Destruction de l'événement
        }
            // Exécuter un événement de la simulation Klunky

       if (!FlagEndKlunky) {
          KlunkyTime = KlunkyEvtList.ReturnPKey();  // Mettre à jour l'horloge
          CurrentEvt = KlunkyEvtList.Remove();      // Extraire l'événement
          CurrentEvt->EvtMain();                    // Exécuter le corps de l'événement
          CurrentEvt->FollowUp();                   // Planification des événements
          CurrentEvt->CleanUp();                    // Ménage
          delete CurrentEvt;                        // Destruction de l'événement
       }
    }
    Report ();
}

    // Planification des événements

EventNotice IncSchedule (Config which, double inc, Event *event, double prio) {
    if (which == ZIPPY) return ( ZippyEvtList.Insert ( Now()+inc, prio, event ) );
    else return ( KlunkyEvtList.Insert ( Now()+inc, prio, event ) );
}
```

Figure 5 : Zippy versus Klunky avec CRN (suite)

```
        // Fin de la sim

    void Stop (Config which) {
        if (which == ZIPPY) FlagEndZippy = 1;
        else FlagEndKlunky = 1;

        FlagEndSim = (FlagEndZippy && FlagEndKlunky);
    }
};
```

Figure 5 : Zippy versus Klunky avec CRN (suite)

La reprise de l'expérience de la section précédente en remplaçant les classes *Simulation* et *RandGen* par les classes *SimCRN* et *GenCRN* a produit un taux d'erreur de seulement 2% pour n = 1, ce qui concorde avec le taux d'erreur de 3% rapporté par Law et Kelton (1991, pp 621) pour la même valeur de n. Par ailleurs, il a été démontré que des classes fondamentales de la librairie telles *Simulation* et *RandGen* peuvent être personnalisées pour des applications spécifiques sans nécessiter la modification du code source original de la librairie. À notre connaissance, l'approche orientée-objet est le seul paradigme de programmation qui permet, lorsqu'utilisé adéquatement, de prédéfinir des modules fonctionnels extensibles. Cette capacité est primordiale en simulation car il existe une très grande variété de systèmes, de nombreuses façons de recueillir ou de manipuler des statistiques et une multitude de stratégies de simulation. Le nombre effarant de combinaisons possibles de ces trois facteurs confirme la nature *ad hoc* de la simulation et la place de l'extensibilité dans les qualités distinctives d'un bon outil de simulation.

L'extensibilité a toutefois un prix. L'utilisation des mécanismes d'héritage nécessite des connaissances de deuxième degré en C++. De plus, la redéfinition d'une procédure suppose une étude

approfondie de son fonctionnement original de façon à ne pas introduire des erreurs à distance. La lecture d'un programme contenant des classes dérivées est ardu car des variables et des procédures semblent se matérialiser par enchantement. Ainsi s'amorce un long périple qui nous fait remonter toujours plus haut dans la hiérarchie des classes. Il va sans dire que les classes à héritage multiple amplifient ce problème considérablement. Les mécanismes d'extension de la librairie sont donc confinés à être utilisés par les programmeurs intermédiaires ou avancés.

6.3 Changement dynamique du niveau de détails

La section précédente nous a vu redéfinir la boucle d'exécution de la classe *Simulation* pour administrer deux simulations en parallèle. Une autre modification intéressante de la boucle d'exécution consiste à interroger le champ type d'un événement avant son actualisation pour changer dynamiquement le niveau de détails d'une simulation.

Une pratique courante en simulation consiste à programmer par étape, chaque programme simulant le système d'intérêt d'une façon plus détaillée que le précédent. L'héritage est particulièrement apte à assister ce type de développement puisqu'il permet de fonder de nouvelles classes sur les anciennes. En conséquence, tous les événements d'une simulation font partie d'une hiérarchie de classes dont *Event* est le roi. Si cette hiérarchie possède plusieurs niveaux, le fait de transtyper un événement permet de changer dynamiquement son effet sur la simulation car ses procédures virtuelles changeront. Il est important de souligner que ce transtypage est valide uniquement pour les classes appartenant à la même branche de la hiérarchie et qu'on ne peut jamais transtyper un événement vers une classe dérivée

de sa classe originale. Le champ type, constant dans tous les objets dérivés de la classe *Event*, permet de déterminer quelle est la classe réelle d'un objet événement.

Par exemple, si la classe *Niveau1* est dérivée de la classe *Event* et que les classes *Niveau2a* et *Niveau2b* sont dérivées de *Niveau1*, le programme suivant permet de changer dynamiquement le niveau de détails de la simulation. Initialement, les événements sont de classes *Niveau2a* ou *Niveau2b*. Ils sont néanmoins insérés dans la liste d'événements transtypés en objets de classe *Niveau1*. Lorsqu'un certain seuil est atteint, la boucle d'exécution transtype automatiquement les événements de classe *Niveau1* dans la classe du niveau 2 correspondante en interrogeant le champ type.

```
enum EventType = { NIVEAU2A, NIVEAU2B };
typedef enum EventType EventType;

class SimDYN : class Simulation<SzEvtLst> {

protected :

   int ChangeFlag;                    // Indique un changement de niveau de détails

public :

   void Start () {                                    // Boucle d'exécution
      while(!FlagEndSim) {

         Time = EventList.ReturnPKey();               // Mettre à jour l'horloge
         CurrentEvt = EventList.Remove();             // Extraire l'événement

               // Si on doit changer le niveau de détail et l'événement est de type NIVEAU2A

         if (ChangeFlag  && ( CurrentEvt->ReturnType_id () == NIVEAU2A )) {

            ((Niveau2a*) CurrentEvt)->EvtMain();      // Exécuter le corps de l'événement
            ((Niveau2a*) CurrentEvt)->FollowUp();     // Planification des événements
            ((Niveau2a*) CurrentEvt)->CleanUp();      // Ménage
         }
```

Figure 6 : Changement dynamique de niveau de détails

```
            // Si on doit changer le niveau de détail et l'événement est de type NIVEAU2B
        else if (ChangeFlag && ( CurrentEvt->ReturnType_id () == NIVEAU2B )) {

            ((Niveau2b*) CurrentEvt)->EvtMain();        // Exécuter le corps de l'événement
            ((Niveau2b*) CurrentEvt)->FollowUp();       // Planification des événements
            ((Niveau2b*) CurrentEvt)->CleanUp();        // Ménage
        }

        else {

            CurrentEvt->EvtMain();                      // Exécuter le corps de l'événement
            CurrentEvt->FollowUp();                     // Planification des événements
            CurrentEvt->CleanUp();                      // Ménage
        }
        delete CurrentEvt;                              // Destruction de l'événement
    }
    Report();
  }
};
```

Figure 6 : Changement dynamique de niveau de détails (suite)

Le changement dynamique du niveau de détails d'une simulation est particulier au paradigme de programmation orientée-objet puisqu'il est le seul à offrir les mécanismes d'héritage. Ses applications incluent les périodes de réchauffement et l'étude des phénomènes rares. Malgré son design orienté-objet, le langage C++ n'est cependant pas particulièrement adapté à cette opération. En effet, une analyse du programme précédent permet de constater qu'en C++ une expression transtypée n'est pas une lvalue (*left value*). Une lvalue est une expression à laquelle on peut affecter une valeur. En conséquence, la boucle d'exécution d'un objet dérivé de la classe *Simulation* doit contenir un cas spécifique pour chaque opération de transtypage possible, ce qui explique le code inélégant du programme présenté dans cette section. De plus, comme mentionné à maintes reprises dans ce mémoire, le C++ ne possède pas de mécanisme pour vérifier dynamiquement le type effectif ou réel d'une variable.

Cet état des choses explique la nécessité de déclarer un champ type dans chacun des membres de la librairie.

6.4 Simulation d'un réseau stochastique dynamique

La raison pour laquelle nous avons exploré la flexibilité et l'extensibilité d'une librairie de simulation orientée-objet s'explique par les exigences de la simulation des réseaux stochastiques dynamiques. En conséquence, nous allons maintenant démontrer comment notre librairie convient à ce type d'application en simulant un réseau stochastique dynamique réel.

Le problème auquel nous allons faire référence tout au long de cet exemple est celui de l'allocation des conteneurs vides (Crainic et al. 1993). Ce problème s'intéresse à la gestion des déplacements terrestres des conteneurs vides provenant du transport maritime international. Il comporte deux volets : l'allocation des conteneurs vides aux clients et la redistribution des conteneurs vides disponibles afin de satisfaire les besoins futurs. Dans le cadre de leurs recherches, Crainic et al. ont obtenu un ruban magnétique contenant le registre annuel des offres et des demandes de conteneurs vides par les clients d'une compagnie maritime. Il contient 10 650 demandes et 9963 offres réparties sur 365 périodes ainsi que les ordres de mouvements utilisés pour les satisfaires. Le ruban contient aussi le laps de temps alloué (*delivery window*) pour répondre à chaque demande. Une telle information n'étant cependant pas disponible pour les offres de conteneurs vides, nous avons décidé d'allouer un délai maximal de 5 jours pour les compléter.

En utilisant ce ruban comme point de départ, nous allons produire un programme qui simule les obligations passées de cette compagnie maritime relativement à la gestion de sa flotte de conteneurs vides. Cet environnement sera ensuite analysé périodiquement par un algorithme d'optimisation fictif afin de déterminer les ordres de mouvements nécessaires pour satisfaire le plus efficacement possible les besoins des clients de la compagnie. La raison pour laquelle nous avons utilisé un algorithme d'optimisation fictif s'explique par le fait que l'algorithme d'optimisation réel pour ce problème (Abrache 1998) n'était pas encore disponible au moment de la conduite de notre expérience. En conséquence, nous avons remplacé l'algorithme d'optimisation par une procédure externe qui répète les ordres de mouvements originaux se trouvant sur le ruban de la compagnie maritime. Néanmoins, l'utilisation d'un simulacre d'algorithme d'optimisation permet tout de même de démontrer l'interface de communication de notre librairie ainsi que la pertinence de son design pour simuler les réseaux stochastiques dynamiques.

Le ruban magnétique dont nous disposons permet d'identifier trois composantes distinctes du problème : les ports, les dépôts et les clients. Les clients offrent les conteneurs vides résultant de la réception de marchandises et demandent des conteneurs vides pour expédier leurs produits. Dans le premier cas, les conteneurs sont transportés de l'emplacement du client à un port ou un dépôt tandis que dans le deuxième cas c'est l'opération inverse qui est effectuée. De plus, il est parfois possible d'effectuer des mouvements directs de conteneurs vides entre un client importateur et un client exportateur. Une quatrième composante du problème, identifiée par Crainic et al., est le flot des navires qui accostent ou quittent les ports, générant ou absorbant une partie de la réserve de conteneurs vides s'y trouvant.

Le ruban ne mentionnant aucune information à cet égard, l'effet des navires sur le système est modélisé implicitement dans notre programme par la remise à niveaux périodiques des réserves de conteneurs vides des ports et des dépôts. Ces réserves sont composées des 21 types de conteneurs différents mentionnés sur le ruban. Chaque port ou dépôt peut accomoder tous les types de conteneurs et le niveau initial des inventaires est de 5 conteneurs de chaque type pour les dépôts et de 10 pour les ports. Les classes suivantes sont utilisées pour représenter les entités du système. Notez que sur le ruban les ports, les dépôts et les clients sont désignés par un identificateur de 4 chiffres utilisés conjointement avec le nom usuel de la ville où ils sont situés, système que nous utilisons nous aussi.

```
        // Type spécialisé pour les trois composantes du problème : les ports,
        // les dépôts et les clients

enum EntityType = { PORT, DEPOT, CUSTOMER };
typedef enum EntityType EntityType;

extern Simulation *SimCON;                    // Prédéclaration de l'objet Simulation

        // La classe Depot sert à représenter les ports et les dépôts. Ils sont différenciés
        // par la valeur du champ Type_id hérité de Entity qui prend les valeurs PORT
        // ou DEPOT

class Depot : public Entity {                 // Dérivé de Entity

  protected :                                 // Ces attributs participent à l'héritage

    MergeList *Inventory;                     // L'inventaire de conteneurs du dépôt
    char InventoryReport[210];                // Va contenir le rapport d'inventaire du dépôt
    int IDNumber;                             // Numéro d'identification
    char CityName [40];                       // Nom de la ville hôte
    int Base;                                 // Le nombre initial de conteneurs
```

Figure 7 : Dépôts et clients de la compagnie maritime

```
public :

        // Constructeur de la classe Depot.  Les paramètres sont le type, PORT ou
        // Depot, le numéro d'identification et le nom de la ville hôte.

    Depot (EntityType type, int idnumber, char *cityname) {

        Type_id = type;
        if (Type_id == PORT) Base = 10;  // Le stock initial d'un port est de 10 conteneurs de
        else Base = 5;                   // chaque type.  Celui d'un dépôt est de 5
        IDNumber = idnumber;
        strncpy(&CityName[ 0], cityname, 39);

          // Création de l'inventaire initial.  Les codes de conteneurs utilisés sont ceux de
          // la compagnie maritime.

        Inventory = new MergeList();
        Inventory->Add ("2000",Base);    Inventory->Add ("2200",Base);
        Inventory->Add("2040",Base);     Inventory->Add("2051",Base);
                                .....
        Inventory->Add("8888",Base);     Inventory->Add("4199",Base);

        InventoryReport = new char [210];   //  Création du buffer du rapport d'inventaire
    }

        // Retourne le numéro d'identification de ce dépôt

    int GetID () {
        return (IDNumber);
    }

        // Création du rapport d'inventaire

    char * GetInventoryList () {
        ostrstream report (&InventoryReport[0],210);        // String stream

        report << *Inventory[ i ];                          // Impression de la MergeList
        return &InventoryReport [ 0];                       // Retourne le rapport
    }

        // Retirer des conteneurs de la réserve du dépôt

    MergeUnit *Deliver (char *ctype,int quantity) {
        return (Inventory->Get (ctype,quantity));
    }

        // Ajouter des conteneurs aux réserves du dépôt

    int Store(char *ctype,int quantity) {
        return (Inventory->Add(ctype, quantity));
    }
};
```

Figure 7 : Dépôts et clients de la compagnie maritime (suite)

```
        // Cette classe symbolise les clients du problème

class Customer : public Entity {            // Dérivé de la classe Entity

  protected :                               // Participe à l'héritage

    int IDNumber;                           // Numéro d'identification
    char CityName [40];                     // Nom de la ville hôte
    EvtNoticeList WindowList;               // Notice des offres et demandes de ce client

  public :

        // Constructeur de la classe Customer. Les paramètres sont le numéro
        // d'identification et le nom de la ville hôte

   Customer (int idnumber, char *cityname) {
        Type_id = CUSTOMER;                 // le type est CUSTOMER
        IDNumber = idnumber;
        strncpy(&CityName[ 0], cityname, 39);
    }

        // Retourne le numéro d'identification de ce client

   int GetID () {
        return (IDNumber);
   }

        // Garder trace des offres et demandes du client

   int AddWindow (int id, EventNotice notice) {
        return (WindowList->AddNotice( id, notice));
   }

        // La requête du client est satisfaite.  Détruire l'événement de fin de window

   int Acknowledge (int id) {
        EventNotice *todelete;              // Pointeur sur un événement Deadline

        todelete = WindowList->FindNotice(id);

          // Détruire l'événement qui signifie la fin de la window pour la requête

        if (todelete) return (SimCON->Cancel(todelete))
        return 1;
   }

   EventNotice FindNotice (int id) {
        return (WindowList->FindNotice(id));
   }
};
```

Figure 7 : Dépôts et clients de la compagnie maritime (suite)

La deuxième étape dans la réalisation du programme consiste à symboliser le réseau routier et ferroviaire utilisé par la compagnie maritime dans ses opérations courantes à l'aide d'un objet *Network* de la librairie. Le ruban nous donne la localisation des 138 dépôts et 717 clients de la compagnie. Les 138 dépôts se divisent en 29 dépôts portuaires et 109 dépôts terrestes. Hélas, le fichier ne contient aucune information sur l'interconnexion entre les dépôts et les clients et un atlas (Debenham 1984) a dû être utilisé pour déterminer les liens qui les unissent. Un bout de ficelle placé sur les principaux axes routiers et ferroviaires de la France et du Benelux a servi d'instrument de mesure. En conséquence, les distances utilisées sont approximatives et des liens sont probablement manquants. De plus, nous avons découvert subséquemment qu'un effort similaire mais plus complet avait déjà était fait. Toutefois, le réseau ainsi construit est suffisamment représentatif pour nous permettre des expérimentations réalistes. La durée du processus de raccordement des noeuds a heureusement été abrégée par la présence de plusieurs clients dans la même ville. Le poids assigné à chaque lien est sa longueur en kilomètres. Pour notre exemple, aucune différence n'est faite entre un lien routier et ferroviaire. Une classe spéciale dérivée de la classe *Node*, la classe *NodeCON*, est utilisée pour chaque noeud du réseau. Elle contient un champ texte contenant le nom usuel de la ville hôte ainsi que des pointeurs pour accomoder un objet de classe *Depot* et une liste d'objets *Customer*. En vertu des règles d'héritage, la classe *NodeCON* peut être utilisée partout où un objet *Node* est attendu.

```
        // Noeud de l'objet network représentant le réseau de la compagnie maritime
        // Les classes Depot et Customer sont définies plus loin.  Elles représentent
        // respectivement un dépôt (portuaire ou terrestre) et un client.

class NodeCON : public Node {                       // Dérivée de la classe Node
  protected:
    char CityName[40];                              // Nom de la ville
    Depot *Dep;                                     // Objet depot si nécessaire
    List<Customer> *CustList;                       // Liste de clients

  public:
    NodeCON (char *cityname) {              // Constructeur
        strncpy(&CityName[ 0], cityname, 39);       // Copier le nom de la ville
        CustList = new List<Customer> ();           // Créer la liste de clients
    }

    void AddDepot(Depot *depot) {                   // Ajouter un dépôt à la ville
        Dep = depot;
    }

    void AddClient(Customer *customer) {            // Ajouter un client à la liste de clients
        CustList->Insert(customer->GetID (), customer);
    }

    Depot * FindDepot(int id) {                     // Trouver le dépôt numéro id
        if  (Dep->GetID () == id) return Dep;
0       return NULL;                                // Retourne NULL en cas d'erreur
    }

    Customer * FindClient(int id) {                 // Trouver le client numéro id
        return CustList->Find ( id );               // Retourne NULL en cas d'erreur
       }

        // Rapport complet sur le noeud

    friend ostream& operator << ( ostream& s, const NodeCON& x ) {
        char t = ' ';                               // espace
        char *depID;                                // Identité d'un dépôt, si présent
        char *OutList;                              // La liste de liens sortants du noeud
        char *InventoryList;                        // Inventaire du dépôt, si présent

        *depID = t;                                 // Initialisation
        *InventoryList = t;
        OutList = OutputEndpoint();                 // Création de la liste de liens sortants

        if (Dep) {                                  // Si un dépôt présent, recueillir info
          depID = Dep->GetID();
          InventoryList = Dep->GetInventoryList();
        }
```

Figure 8 : Noeud du réseau de la compagnie maritime

```
        return s << "Node label : "        << x.CityName              << " \n"
                 << "InDegree : "          << x.InDegree              << " \n"
                 << "OutDegree : "         << x.OutDegree             << " \n"
                 << "#Depot : "            << (x.Dep == NULL)         << " \n"
                 << "#Customer : "         << x.CustList->Length ()   << " \n"
                 << "Depot : "             << depID                   << " \n"
                 << "Customer : "          << x.CustList              << " \n"
                 << "Link:"                << OutList                 << " \n"
                 << "Inventory"            << InventoryList           << " \n";
    }
};
```

Figure 8 : Noeud du réseau de la compagnie maritime (suite)

La principale fonction de l'objet Network dans le programme est de rendre accessible à l'algorithme d'optimisation l'information relative à la localisation des dépôts et des clients, aux itinéraires disponibles et aux niveaux d'inventaire des dépôts. Pour ce faire, un fichier sommaire du réseau doit être produit. La création de fichier est un aspect important de la simulation des réseaux stochastiques dynamiques en raison du volume élevé de données à transmettre entre le système simulé et l'algorithme d'optimisation adjoint. Nous supposons ici que l'algorithme d'optimisation choisi ne peut utiliser directement un objet Network. Un mécanisme particulier de l'approche orientée-objet, la surcharge d'opérateur, facilite la communication entre la simulation et son algorithme d'optimisation en permettant la définition de l'opérateur de sortie standard (<<) de C++ pour des classes arbitraires. L'opérateur << de la classe *NodeCON* permet de créer un résumé des informations pertinentes d'une de ses instances pour ensuite les imprimer dans un fichier ou, facultativement, à l'écran. La classe Network définit elle aussi l'opérateur << qui se borne à appliquer récursivement l'opérateur << à chaque item de sa liste de noeuds. La figure suivante montre une partie du fichier réseau produit par le programme. Les commentaires ont été ajoutés afin d'expliquer chaque item du fichier.

```
Node label : bruxelles                                  // Nom du noeud
InDegree : 4                                            // Nombre de liens entrants
OutDegree : 4                                           // Nombre de liens sortants
#Depot : 1                                              // Nombre de dépôt ( 0 ou 1 )
#Customer : 7                                           // Nombre de clients
Depot : 1110                                            // Identificateur du dépôt, si présent
Customer : 1110, 1111, 1112, 1115, 1116, 1118, 1119     // Identificateur des clients
Link :                                                  // En-têre, section des liens
antwerpen 37.5                                          // Nom du noeud, Distance en KM
gent 50
chatelet 50
tessenderloo 75
Inventory :                                             // En-tête, section inventaire
2000 5                                                  // Type du conteneur, Quantité
2200 5
  .....                                                 // Pour des raisons d'espace, les 21
2072 5                                                  // lignes de cette section ne sont
4500 5                                                  // pas montrées

Node label : amsterdam                                  // Prochain noeud
  .....
```

Figure 9 : Exemple de rapport sur le réseau

Ce fichier est toutefois incomplet. En effet, les offres et les demandes de conteneurs vides ne se retrouvent pas parmi les informations transmises à l'optimiseur. La raison en est simple, les offres et les demandes sont des événements et c'est la classe *Simulation* et non la classe *Network* qui les gère. Les événements de notre programme sont les classes *Demand*, *UnforecastDemand*, *Supply*, *UnforecastSupply*. Elles représentent respectivement les demandes planifiées, les demandes surprises, les offres planifiées et les offres surprises de conteneurs vides. Une offre ou une demande de conteneurs vides concerne uniquement un seul type de conteneurs. Les offres ou demandes composées sont séparées en événements distincts. L'actualisation d'un événement Demand, UnforecastDemand, Supply ou UnforecastSupply provoque la planification automatique d'un événement *Deadline* pour la période où se termine le temps maximal alloué pour la satisfaction de la requête (*time window*).

L'actualisation d'un *Deadline* signifie le non-respect de la requête d'un client et cette information est conservée comme mesure de performance du système. Chaque objet *Customer* garde trace des événements *Deadline* qui lui sont reliés afin de pouvoir les retirer de la liste d'événements quand les requêtes correspondantes sont satisfaites. Les objets *Deadline* se voient assigner un numéro selon l'ordre de leur création et c'est ce numéro qui est utilisé dans un objet *Customer* pour les retracer. La satisfaction d'une demande est symbolisée par un événement *DeliverCON* tandis que la satisfaction d'une offre est symbolisé par un événement *PickUpCON*. Les occurences de ces événements sont planifiées par l'algorithme d'optimisation. La figure suivante présente les événements du programme. Les classes *Supply* et *UnforecastSupply* ont été omises puisque les classes *Demand* et *UnforecastDemand* ont la même structure.

```
extern Simulation *SimCON;                    // Prédéclaration de l'objet Simulation
extern Network *NetCON;                       // Prédéclaration de l'objet Network
extern List<char> FailedDemand;               // Liste des demandes non satisfaites
extern List<char> FailedSupply;               // Liste des offres non satisfaites

        // Class Demand. Symbolise une demande dont l'occurance future est
        // prévue par l'algo d'optimisation.

class Demand : public Event {                 // Dérivé de la classe Event

  protected:                                  // Participe à l'héritage

        int CustomerIDNumber;                 // Numéro du client demandeur
        int IDNumber;                         // Numéro de la demande
        float WindowBegin;                    // Début de la demande
        float WindowEnd;                      // Fin de la demande
        char ContainerType[ 5];               // Type de conteneurs demandés
        int Quantity;                         // Combien de conteneurs demandés
```

Figure 10 : Chaîne d'événements pour les opérations de la compagnie maritime

```
public:

        // Constructeur. Les informations nécessaires sont le numéro du client demandeur,
        // le numéro de la demande, la période où la demande se produit, la période où
        // la demande se termine, le type des conteneurs demandés et la quantité de
        // conteneurs demandés

Demand (int who, int idnumber, float start, float beforehorizon, char *what, int howmuch) {
        CustomerIDNumber = who;
        IDNumber = idnumber;
        WindowBegin = start;
        WindowEnd = beforehorizon;
        strncpy(&ContainerType[ 0], what, 5);
        Quantity = howmuch;
}

        // Corps de l'événement. Doit être obligatoirment défini car pure virtuelle dans Event
        // Les objets Demand sont des indicateurs pour l'algo d'optimisation.  En
        // conséquence, rien à faire.

EvtMain () {
}

        // Planification des événements subséquents.  L'actualisation d'un objet Demand
        // provoque la création d'un Deadline.  Ne pas oublier de donner la référence
        // du Deadline au client concerné. Les Deadline sont planifiés avec priorité 15
        // pour s'assurer que les SupplyCON sont actualisés en premier.

 FollowUp  () {
        EventNotice Notice;
        Notice = SimCON->AbsSchedule(new Deadline( CustomerIDNumber, IDNumber,
                WindowEnd,&ContainerType[ 0 ], Quantity, "demand" ), WindowEnd, 15);
        (NetCON->FindCustomer (CustomerIDNumber))->AddWindow(IDNumber,Notice);
 }

        // Ménage. Doit être obligatoirement défini car pure virtuelle dans Event

CleanUp () {
}

        // Opérateur de sortie.  Nécessaire pour la communication avec l'algo
        // d'optimisation

friend ostream& operator << ( ostream& s, const Demand& x ) {
        return s  << "#Demand : "          << x.IDNumber          << " \n"
                  << "Customer : "         << x.CustomerIDNumber  << " \n"
                  << "WindowBegin : "      << x.WindowBegin       << " \n"
                  << "WindowEnd : "        << x.WindowEnd         << " \n"
                  << "Type : "             << x.ContainerType     << " \n"
                  << "Quantity : "         << x.Quantity          << " \n" ;
}
};
```

Figure 10 : Chaîne d'événements pour les opérations de la compagnie maritime (suite)

```
        // Class UnforecastDemand. Symbolise une demande imprévue.
        // Sa définition est semblable à celle de Demand sauf pour l'opérateur de sortie.

class UnforecastDemand : public Event {          // Dérivé de la classe Event

  protected:                                     // Participe à l'héritage

        int CustomerIDNumber;                    // Numéro du client demandeur
        int IDNumber;                            // Numéro de la demande
        float WindowBegin;                       // Début de la demande
        float WindowEnd;                         // Fin de la demande
        char ContainerType[ 5];                  // Type de conteneurs demandés
        int Quantity;                            // Quantité demandée

  public:

        // Constructeur. Les informations nécessaires sont le numéro du client demandeur,
        // le numéro de la demande, la période où la demande se produit, la période où
        // la demande se termine, le type des conteneurs demandés et la quantité de
        // conteneurs demandés

  UnforecastDemand (int who, int idnumber, float start, float beforehorizon,
                    char *what, int howmuch) {
        CustomerIDNumber = who;
        IDNumber = idnumber;
        WindowBegin = start;
        WindowEnd = beforehorizon;
        strncpy(&ContainerType[ 0], what, 5);
        Quantity = howmuch;
  }

        // Corps de l'événement. Doit être obligatoirement défini car pure virtuelle dans Event

  EvtMain () {
  }

        // Planification des événements subséquents. L'actualisation d'un
        // objet UnforecastDemand provoque la création d'un Deadline. Ne pas oublier
        // de donner la référence du Deadline au client concerné. Les Deadline sont planifiés
        // avec priorité 15 pour s'assurer que les SupplyCON sont actualisés en premier.

   FollowUp  () {
        EventNotice Notice;
        Notice = SimCON->AbsSchedule(new Deadline(CustomerIDNumber, IDNumber,
                WindowEnd,&ContainerType[ 0 ], Quantity, "demand" ), WindowEnd, 15);
        (NetCON->FindCustomer (CustomerIDNumber))->AddWindow(IDNumber, Notice);
   }

        // Corps de l'événement. Doit être obligatoirement défini car pure virtuelle dans Event

  CleanUp () {
  }
```

Figure 10 : Chaîne d'événements pour les opérations de la compagnie maritime (suite)

```
        // Opérateur de sortie.  La présence des UnforecastDemand doit demeurer cachée
        // à l'algo d'optimisation

    friend ostream& operator << ( ostream& s, const UnforecastDemand& x ) {
        return s;
    }
};

        // Cette classe représente la fin de la Window allouée pour satisfaire l'offre ou
        // la demande

class Deadline : public Event {                    // Dérivé de la classe Event

    protected:                                     // Participe à l'héritage

        int CustomerIDNumber;                      // Numéro du client en attente
        int IDNumber;                              // Numéro identification pour le client
        float WindowEnd;                           // Fin de la Window
        char ContainerType[ 5];                    // Type de conteneur demandé
        int Quantity;                              // Quantité demandée
        char Type[10];                             // Offre ou demande

    public:

        // Constructeur de la classe Deadline.  Les paramètres sont le numéro
        // du client en attente, le numéro utilisé pour retrouver la référence à cet
        // événement dans le client, la période de fin de Window,
        // le type de conteneur demandé, la quantité de conteneurs demandés

    Deadline(int who, int idnumber, float beforehorizon, char *what, int howmuch,char *type) {
        CustomerIDNumber = who;
        IDNumber = idnumber;
        WindowEnd = beforehorizon;
        strncpy(&ContainerType[ 0], what, 4);
        Quantity = howmuch;
        strncpy(&Type[ 0], type, 9);
    }

        // Corps de l'événement.  La requête n'a pas été satisfaite.
        // Annuler la requête et conserver le numéro de l'offre ou la demande.

    EvtMain () {
        SimCON->Cancel(
          ( NetCON->FindCustomer(CustomerIDNumber) )->FindNotice(IDNumber)
        );
        if (Type_id == DEMAND) FailedDemand->Insert(IDNumber);
        else FailedSupply->Insert(IDNumber);
    }

        // Rien à planifier. Doit être obligatoirement défini car pure virtuelle dans Event

    FollowUp () {   }
```

Figure 10 : Chaîne d'événements pour les opérations de la compagnie maritime (suite)

```
        // Ménage. Doit être obligatoirement défini car pure virtuelle dans Event

    CleanUp () {
    }

        // Opérateur de sortie

    friend ostream& operator << ( ostream& s, const Deadline& x ) {
        return s  << x.Type                 << "  Deadline"          << " \n"
                  << "Request ID :"         << x.IDNumber            << " \n"
                  << "Customer : "          << x.CustomerIDNumber    << " \n"
                  << "WindowEnd : "         << x.WindowEnd           << " \n"
                  << "Type : "              << x.ContainerType       << " \n"
                  << "Quantity : "          << x.Quantity            << " \n" ;
    }
};

        // Les événements de la classe DeliverCON sont utilisés pour satisfaire une demande.

class DeliverCON : public Event {                   // Dérivé de la classe Event

    protected:                                      // Participe à l'héritage

        int CustomerIDNumber;                       // Numéro du client
        int IDNumber;                               // Numéro de la demande
        char ContainerType[ 5];                     // Type de conteneur
        int Quantity;                               // Quantité

    public:

        // Constructeur, les paramètres sont le client destinataire, le numéro de la demande
        // à satisfaire, le type des conteneurs fournis et leur quantité

    DeliverCON (int who, int idnumber, char *what, int quantity) {
        CustomerIDNumber = who;
        IDNumber = idnumber;
        strncpy(&ContainerType[ 0], what, 4);
        Quantity = quantity;
    }

        // Corps de l'événement. Le client accuse réception de ses conteneurs

    EvtMain () {
        SimCON->Cancel(
          ( NetCON->FindCustomer(CustomerIDNumber) )->Acknowledge(IDNumber)
        );
    }

        // Rien à planifier. Doit être obligatoirement défini car pure virtuelle dans Event

    FollowUp () {
    }
```

Figure 10 : Chaîne d'événements pour les opérations de la compagnie maritime (suite)

```
        // Ménage. Doit être obligatoirement défini car pure virtuelle dans Event

    CleanUp () {
    }

        // Définition de l'opérateur de sortie.  Signifier à l'algo d'optimisation que des
        // conteneurs sont en route

    friend ostream& operator << ( ostream& s, const DeliverCON& x ) {
        return s  << "DemandCON"                                    << " \n"
                  << "Demand ID : "      << x.IDNumber              << " \n"
                  << "Type : "           << x.ContainerType         << " \n"
                  << "Quantity : "       << x.Quantity              << " \n" ;
    }
};

        // Les événements de la classe DeliverCON sont utilisés pour satisfaire une offre.

class PickUpCON : public Event {                    // Dérivé de la classe Event

    protected:                                      // Participe à l'héritage

        int CustomerIDNumber;                       // Numéro du client qui offre
        int DepotIDNumber;                          // Numéro du client qui va recevoir
        int IDNumber;                               // Numéro de l'offre
        char ContainerType[ 5];                     // Type de conteneur offert
        int Quantity;                               // Quantité de conteneurs offerts

    public:

        // Constructeur, les paramètres sont le client source, le dépôt destinataire,
        // le numéro de l'offre à satisfaire, le type des conteneurs recueillis et leur quantité

    PickUpCON (int who, int where, int idnumber, char *what, int quantity) {
        CustomerIDNumber = who;
        DepotIDNumber = where;
        IDNumber = idnumber;
        strncpy(&ContainerType[ 0], what, 5);
        Quantity = quantity;
    }

        // Corps de l'événement.  Le client retire son offre et les conteneurs sont ajoutés
        // à la réserve d'un dépôt

    EvtMain () {
        SimCON->Cancel(
          ( NetCON->FindCustomer(CustomerIDNumber) )->Acknowledge(IDNumber)
        );
        (NetCON->FindDepot(DepotIDNumber))->Add(ContainerType,Quantity);
    }
```

Figure 10 : Chaîne d'événements pour les opérations de la compagnie maritime (suite)

```
        // Rien à planifier. Doit être obligatoirement défini car pure virtuelle dans Event

    FollowUp () {
    }

        // Ménage. Doit être obligatoirement défini car pure virtuelle dans Event

    CleanUp () {
    }

        // Définition de l'opérateur de sortie.  Signifier à l'algo d'optimisation que les
        // conteneurs vont être recueillis

    friend ostream& operator << ( ostream& s, const ResponseCON& x ) {
        return s  << "PickUpCON"                                  << " \n"
                  << "SupplyID : "      << x.IDNumber             << " \n"
                  << "Type : "          << x.ContainerType        << " \n"
                  << "Quantity : "      << x.Quantity             << " \n" ;

    }
};
```

Figure 10 : Chaîne d'événements pour les opérations de la compagnie maritime (suite)

Les plus observateurs s'interrogent sûrement sur notre traitement de l'événement *PickUpCON*. En effet, le code utilisé semble décrire une situation où les conteneurs vides recueillis chez les clients sont immédiatement ajoutés à la réserve d'un dépôt et ce sans aucun délai de transport. Nous procédons de cette manière pour simplifier la mécanique de la simulation. En effet, le ruban de la compagnie maritime nous indique à quelle période une offre a été satisfaite et par quel dépôt. Un événement *PickUpCON* représente donc en fait le retour d'un conteneur à un dépôt plutôt que sa cueillette chez le client. Cette façon de faire est uniquement valide si le temps total nécessaire pour satisfaire une offre, incluant le transport, ne s'étend pas sur plusieurs périodes. Dans le cas contraire, nous allons possiblement détecter des offres dont les délais sont expirés même si ce n'est pas le cas. L'objectif de ce programme n'étant pas l'interprétation de ses résultats, nous avons décidé de

procéder de cette façon afin de simplifier le code et par conséquent, sa présentation.

Un second point d'implémentation important est le recours au mécanisme de priorité de la librairie. Les événements Deadline sont planifiés avec une priorité inférieure à la priorité par défaut en effet dans la librairie de façon à garantir qu'une requête n'est pas retirée du système uniquement en raison de l'ordre d'actualisation des événements. Aussi, il est important de souligner que l'opérateur de sortie de la classe *UnforecastDemand* est sans effet puisque la présence des événements de ce type dans la liste d'événements ne doit pas être communiquée à l'algorithme d'optimisation. Finalement, une amélioration structurelle importante à nos événements consisterait à regrouper les attributs partagés par les classes *Demand*, *UnforecastDemand*, *Deadline*, *DeliverCON* et *PickUpCON* dans la même classe et à utiliser les mécanismes d'héritage pour éviter la répétition de leur déclaration. De façon à permettre une lecture plus facile du code présenté, cette amélioration n'a pas été implémentée.

Initialement, la simulation contient les événements relatifs aux offres et demandes pour les 7 premières périodes. Par la suite, chaque fois qu'une période t se termine, les événements de la période t+7 sont ajoutés à la simulation. De cette façon, l'algorithme planifie toujours les mouvements de conteneurs vides en fonction des événements prévus pour les 7 prochains jours. Chaque offre ou demande a une probabilité de 5% de devenir une surprise, c'est-à-dire un événement de classe *UnforecastSupply* ou *UnforecastDemand*. Le ruban spécifie pour chaque requête le type et le nombre de conteneurs demandés. La figure suivante présente un échantillon du fichier présenté à l'optimiseur.

```
// Ceci est un échantillon du fichier présenter èa l'optimiseur à la fin
// de la période 1.

#Demand : 2              // Cette demande n'est pas encore concrétisée
Customer : 1139          // Le client demandeur
WindowBegin : 2          // Période prévue pour la confirmation de cette demande
WindowEnd : 5            // Dernière période pour satisfaire la demande
Type : 2200              // Le type de conteneur demandé
Quantity : 2             // La quantité de conteneurs demandés

demand Deadline          // Cette demande est confirmée
RequestID : 1            // Numéro de la demande
Customer : 1110          // Client demandeur
WindowEnd : 4            // Dernière période pour satisfaire la demande
Type : 2232              // Le type de conteneur demandé
Quantity : 1             // La quantité de conteneurs demandés
```

Figure 11 : Exemple de rapport sur les événements

Un événement très simple que nous n'avons pas décrit, l'événement *Optimize*, est planifié pour la fin de chaque période. Il est planifié avec une priorité encore plus basse que celle des objets *Deadline* afin d'être toujours le dernier événement actualisé d'une période. Cet événement instigue la création du fichier sommaire sur la simulation et démarre l'algorithme d'optimisation. Le ruban magnétique contient pour chaque offre et demande le dépôt ainsi que la période à laquelle la requête correspondante a été satisfaite. Notre algorithme d'optimisation fictif consiste donc à retrouver sur le ruban quel dépôt a été utilisé originalement par la compagnie maritime pour satisfaire la requête, à diminuer la réserve de conteneurs du dépôt de la quantité spécifiée et à planifier un événement *DeliverCON* ou *PickUpCON* à la période indiquée sur le ruban. En conséquence, le fichier sommaire de la simulation doit être analysé pour déterminer les nouvelles requêtes de chaque période. Des conteneurs sont artificiellement créés afin de palier à toute rupture de stock d'un dépôt. Notre algorithme d'optimisation produit un fichier où chaque ligne indique les paramètres nécessaires pour créer un objet

DeliverCON ou *PickUpCON* ainsi que la période à laquelle il doit se produire.

// EventType	#Client		#Requête	#Conteneurs	Quantité	Période
DeliverCON	1291		34	2072	2	8
DeliverCON	1244		35	2002	1	5
DeliverCON	1124		36	4300	1	7
// EventType	#Client	#Depôt	#Requête	#Conteneurs	Quantité	Période
PickUpCON	1139	1259	37	2040	1	9
PickUPCON	1225	1264	38	2020	1	7

Figure 12 : Exemple de réponse de l'algorithme d'optimisation

L'événement *Optimize* termine son travail en planifiant chacun des événements spécifiés dans le fichier résultat de l'algorithme d'optimisation et la fin de son exécution signale un changement de période et le début d'un nouveau cycle de simulation.

Malgré les nombreuses simplifications de ce programme dans le traitement du problème de l'allocation des conteneurs vides, toutes les composantes nécessaires pour une simulation plus réaliste de ce problème sont en place. En effet, la faiblesse de ce programme est le soi-disant algorithme d'optimisation utilisé, ce qui est hors des considérations de ce mémoire. Par contre, le programme met en valeur plusieurs avantages de notre librairie pour la simulation des réseaux stochastiques dynamiques. Premièrement, nous avons pu manipuler un fichier de données externe sans problème, ce qui n'est pas toujours le cas dans les simulateurs ou langage de simulation. Deuxièmement, les mécanismes d'héritage de l'approche orientée-objet nous ont permis de créer une classe qui représente spécifiquement les noeuds de notre réseau stochastique dynamique

tout en demeurant compatible avec la classe prédéfinie *Network*. Troisièmement, une composante externe à la librairie de simulation, un algorithme d'optimisation, a pu être utilisé conjointement avec la librairie. Quatrièmement, l'ensemble de la librairie a permis de reproduire les opérations d'une compagnie maritime sur une période d'une année.

Certains désavantages de la librairie ont toutefois été mis en lumière par ce programme. La définition de l'opérateur $<<$, si utile pour générer les sommaires de la simulation, nécessite obligatoirement des modifications aux classes *Simulation* et *Network* car un opérateur ne peut être déclaré pur virtuel. En conséquence, de longues opérations de transtypage sont nécessaires pour garantir l'utilisation de la bonne instance de $<<$. De plus, la librairie ne possède pas d'interface pour des formats de fichiers standards afin d'échanger facilement des fichiers avec des algorithmes d'optimisation. Le code qui manipule le fichier sommaire de la simulation dans notre algorithme d'optimisation représente à lui seul environ 50% de la taille du programme présenté.

6.5 Conclusion

Nous avons prouvé dans ce chapitre la flexibilité des simulations construites avec notre librairie et l'extensibilité de la librairie elle-même. Plus important encore, nous avons démontré un mécanisme général qui permet la coopération entre un algorithme d'optimisation indépendant et la simulation d'un problème d'allocation de ressources réel. Par contre, le fait qu'une expression transtypée ne soit pas considérée une *lvalue* en C++, c'est-à-dire une expression à laquelle on peut affecter une valeur, rend nos stratégies d'extension

inélégantes lors de la création d'un programme complexe. Force est de constater qu'un mécanisme dynamique d'identification des types est indispensable pour bénéficier au maximum des possibilités d'extension de notre librairie, mécanisme inexistant dans le langage que nous avons choisi pour la programmer. Paradoxalement, le C++ est un langage populaire, efficace et bien standardisé qui constitue un outil de choix pour programmer des algorithmes d'optimisation.

CONCLUSION

La recherche opérationnelle est la science des décisions. Appliquée au domaine des transports, elle a pour but de procurer aux agents de décision les informations pertinentes pour établir des politiques d'opération optimales. La taille et la subtilité des problèmes de transport sont cependant telles que seule l'informatique permet d'en gérer la complexité d'une façon efficace et cohérente. En conséquence, la solution de ces problèmes prend souvent la forme d'un algorithme d'optimisation pour réseau stochastique dynamique. Un réseau stochastique dynamique est un modèle probabiliste d'un problème de transport.

En raison des pouvoirs décisionnels qui sont conférés aux algorithmes d'optimisation, il est indispensable d'évaluer empiriquement la performance des solutions qu'ils proposent. Or, ces expériences supposent une évolution stochastique dynamique implicite du système étudié, hypothèse uniquement vérifiée dans le système réel. L'acquisition d'un environnement expérimental virtuel qui permet de reproduire cette dimension des problèmes étudiés est donc indispensable à la conduite de ces expériences. Nous avons limité le domaine des problèmes étudiés aux problèmes d'allocation de ressources, problèmes qui consistent à orchestrer les actions qui permettent de disposer au moment et à l'endroit opportun des unités

de ressources aptes à satisfaire les modalités physiques et temporelles d'un ensemble de requêtes.

La simulation consiste à utiliser un ordinateur pour reproduire les activités d'un système qui nous intéresse à partir d'un ensemble de règles qui semblent modéliser ses opérations. Elle constitue un environnement expérimental parfaitement contrôlable. Le contexte stochastique d'un système peut être déterminé avec les générateurs aléatoires de l'outil de simulation utilisé qui, s'ils sont bien conçus, permettent de reproduire à volonté un contexte stochastique particulier. De plus, la simulation préserve l'individualité de chaque composante du système, faculté particulièrement utile pour étudier formellement les effets des décisions d'un algorithme d'optimisation, et l'effet de vague, racine du contexte dynamique d'un système. La nature statistique des simulations implique toutefois que leurs résultats doivent être rigoureusement analysés.

L'inconvénient principal de la simulation est la programmation qu'elle requiert. Cette difficulté va en s'accroissant avec l'emploi d'un simulateur, d'un langage de simulation ou d'une librairie de simulation. Toutefois, la flexibilité de l'outil de simulation employé augmente selon la même règle. La simulation des réseaux stochastiques dynamiques nécessite le recours à une librairie car elle couvre un domaine très vaste dont les spécificités ne peuvent être énumérées de façon exhaustive. De plus, son implémentation est souvent assujettie à des composantes logicielles extérieures, tels des algorithmes d'optimisation, dont les spécifications nous sont à priori inconnues.

Le paradigme de programmation orientée-objet, tel que défini par le langage C++, permet d'implémenter une librairie de simulation par

événements discrets qui atténue les difficultés de programmation liées à son format tout en permettant la création d'un environnement expérimental éloquent et extensible. Cette faculté n'est pas étrangère au fait que l'approche orientée-objet ait été introduite par Simula, un langage développé pour la simulation. Le paradigme de programmation orientée-objet offre au modéliste des types abstraits de données personnalisées de haut-niveau qui permettent une bonne correspondance entre les modèles théoriques et les modèles digitaux. De plus, l'utilisation des classes facilite l'écriture de programmes de meilleure qualité par les modélistes en promulguant des concepts du génie logiciel tels l'encapsulation des données et la réutilisation. Cette réduction des contraintes d'ordre technique liée à l'utilisation d'un langage de programmation tout-usage permet la création d'une librairie de simulation dont la facilité d'utilisation est équivalente à celle des langages de simulation.

Par ailleurs, une utilisation judicieuse des mécanismes orientés-objet définis par le langage C++ permet de créer un environnement expérimental spécifique aux problèmes d'allocation de ressources qui tient compte de la diversités de ces problèmes. D'une part, cet environnement contient des classes génériques symbolisant les éléments essentiels d'une simulation discrète d'un réseau stochastique dynamique, c'est-à-dire les événements, les entités, les ressources, les noeuds et les arcs. D'autre part, les mécanismes d'héritage et les fonctions virtuelles coopèrent afin de permettre l'évolution des classes de la librairie ainsi que la redéfinition ponctuelle de leurs composantes. Cette dualité a été utilisée à bon escient pour créer un schéma standard d'événements qui est intuitif pour les usagers tout en étant extensible. De plus, un autre mécanisme orienté-objet, les patrons, autorisent la création de structures modulaires dont les attributs sont

interchangeables. Ainsi, on peut créer des ressources qui instiguent automatiquement des événements arbitraires lorsque certains seuils sont atteints ou lorsqu'une situation particulière se présente. Ces mécanismes sont néanmoins assujettis à la création d'un champ type dans les classes de la librairie qui permet d'identifier la classe effective d'un objet.

Ce champ type est le principal reproche qu'on peut faire à notre librairie. Le C++ ne possède pas de mécanismes pour déterminer dynamiquement le type d'un objet. De plus, une expression transtypée n'est pas une *lvalue* en C++, c'est-à-dire une expression à laquelle on peut affecter une valeur. Ces deux caractéristiques, liées au fait que les règles d'héritage généralisent les objets, ont pour effet d'exiger la création d'une variable différente pour chaque type de pointeur à transtyper. Cette situation est ennuyante dans les procédures complexes. Les mécanismes d'héritage de C++ ne sont clairement pas conçus pour l'usage bidirectionnel que nous en faisons. Le fait d'avoir choisi de ne pas utiliser les procédures à paramètres variables est quant à lui à reconsidérer. Ces procédures ont été laissées de côté en raison de leur excentricité et des contraintes qu'elles posent pour les programmeurs débutants. Toutefois, elles auraient permis de définir plus précisément la structure des classes qui composent la librairie.

La pertinence de concevoir une librairie pour la simulation des réseaux stochastiques dynamiques de cette manière a été prouvée en implémentant le problème de l'allocation des conteneurs vides. La simulation réalisée reproduit les offres et demandes de conteneurs vides des clients d'une compagnie maritime pendant un an. À cette simulation, nous avons adjoint un algorithme d'optimisation fictif qui

déterminait dans les registres de la compagnie comment la requête originale avait été satisfaite. Malgré cette simplification, il a été démontré comment la librairie peut communiquer avec un algorithme d'optimisation disjoint en utilisant des fichiers. De plus, des exemples accessoires ont été présentés afin d'illustrer en détails les mécanismes d'extension de la librairie. Ils consistaient à implémenter la technique de réduction de la variance par nombre aléatoire commun et à changer dynamiquement le niveau de détails d'une simulation.

Bibliographie

Abrache, J., *Mise au Point et Implantation d'Algorithmes pour l'Allocation Déterministe de Conteneurs Vides*, Mémoire de Maîtrise, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1998.

Ball, P., Love D., The Key to Object-Oriented Simulation : Separating the User and the Developper, *Proceedings of the 1995 Winter Simulation Conference*, eds., C. Alexopoulos, K. Kang, W. R. Lilegdon, D. Goldsman, 1995, pp. 768-774.

Braklow, J. W., Graham, W. W., Hassler, S. M., Peck, K. E., Powell, W. B., Interactive Optimization Improves Service and Performance for Yellow Freight System, *Interfaces*, 22,1, 1992, pp. 147-172.

Bratley, P., Fox, B. L., Schrage, L. E., *A Guide to Simulation*, 2e édition, 1987, Springer Verlag.

Brown, R., Calendar Queues : A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem, *Communications of the ACM,* 31, 1988, pp. 1220-1227.

Cellier, F. E., *Continuous System Modeling*, 1991, Springer Verlag.

Cornett, K. K., Miller, S. A., An Aircraft Operations Simulation Model for the United Parcel Service Louisville Air Park, *Proceedings of the 1996 Winter Simulation Conference*, eds., J. M. Charnes, D. J. Morrice, D. T. Brunner, J. J. Swain, 1996, pp 1341-1346.

Crain, R. C., Smith, D. S., Industrial Strength Simulation Using GPSS/H, *Proceedings of the 1995 Winter Simulation Conference*, eds., C. Alexopoulos, K. Kang, W. R. Lilegdon, D. Goldsman, 1995, pp. 487-493.

Crainic, T. G., Gendreau, M., Dejax, P., Dynamic and Stochastic Models for the Allocation of Empty Containers, *Operations Research*, 41, 1, 1993 pp. 102-126.

Dahl, O. J., Simula – An Algol-Based Simulation Language, *Communications of the ACM,* 9, 1966, pp. 671-678.

Debenham, F., *The Great World Atlas*, 1984, The Reader's Digest Association Limited.

Fishburn, P. T., Golkar, J., Taaffe, K., Simulation of Transportation Systems, *Proceedings of the 1995 Winter Simulation Conference*, eds., C. Alexopoulos, K. Kang, W. R. Lilegdon, D. Goldsman, 1995, pp. 51-54.

Fujimoto, R. M., Parallel and Distributed Simulation, *Proceedings of the 1995 Winter Simulation Conference*, eds., C. Alexopoulos, K. Kang, W. R. Lilegdon, D. Goldsman, 1995, pp. 118-125.

Geuder, D. F., Object-Oriented Modeling with Simple++, *Proceedings of the 1995 Winter Simulation Conference*, eds., C. Alexopoulos, K. Kang, W. R. Lilegdon, D. Goldsman, 1995, pp. 534-540.

Healy, K. J., Kilgore, R. A., Introduction to Silk and Java based Simulation, *Proceedings of the 1998 Winter Simulation Conference*, 1998.

Hlupic, V., Simulation Software Selection using SimSelect, *Simulation*, 69, 4, 1997 pp. 231-239.

Joines, J. A., Powell, Jr, Roberts, S.D., Object-Oriented Modeling and Simulation with C++, *Proceedings of the 1996 Winter Simulation Conference*, eds., J. M. Charnes, D. J. Morrice, D. T. Brunner, J. J. Swain, 1996, pp 65-72.

Jones, D. W., Henriksen, J. O., Pedgen, R. G., Sargent, R. M., O'Keefe, R. M., Unger, B. W., Implementations of Time (Panel), *Proceedings of the 1986 Winter Simulation Conference*, eds., J. Wilson, J. Henriksen, S. Roberts, 1986, pp 409-416.

Law, A., Kelton, W., *Simulation Modeling and Analysis*, deuxième édition, 1991, McGraw-Hill.

L'Écuyer, P., *SIMOD : Définition fonctionnelle et guide d'utilisation*, 1993, rapport technique # 221, Département d'informatique et de recherche opérationnelle, Université de Montréal.

L'Écuyer, P., Coté, S. Implementing a Random Number Package with Splitting Facilities, *ACM Transactions on Mathematical Software*, 17, 1, 1991, pp. 98-111.

McCormack, W. M., Sargent, R. G., Analysis of Future Event Set Algorithms for Discrete Event Simulation, *Communications of the ACM,* 24, 1981, pp. 801-812.

McGeoch, C., Toward an Experimental Method for Algorithm Simulation, *INFORMS Journal on Computing*, 8, 1, 1996, pp. 1-15.

Merkaryev, Y., Visipkov, V., A Survey of Optimisation Methods in Discrete Systems Simulation, *First Joint Conference of International Proceedings,* 1996, 104-110.

Nicol, D., Fujimoto, R. M., Parallel Simulation Today, *Annals of Operation Research*, 53, 1994, pp. 249-286.

Paul, R, Simulation Optimisation Using a Genetic Algorithm, *Simulation Practice and Theory,* 6, 1998, pp. 601-611.

Pidd, M., Guidelines for the Design of Data Driven Generic Simulators for Specific Domains, *Simulation*, 59, 4, 1992, pp. 237-243.

Piera, M. A., de Prada, C., Modeling Tools in the Continuous Simulation Field, *Simulation*, 66, 3, 1996, pp 179-189.

Powell, W. B., On Languages for Dynamic Ressource Scheduling Problems, *Fleet Management and Logistics*, eds., T. G. Crainic et G. Laporte, Kluwer Academics Publishers, 1998, pp 127-157.

Powell, W. B., Sheffi, Y., Nickerson, K. S., Atherton, S., Maximizing Profits for North American Van Lines Truckload Division, A New Framework for Pricing and Operations, *Interfaces*, 18, 1, 1988, pp. 21-41.

Pritsker, A. A., *Introduction to Simulation and SLAM II*, quatrième édition, 1995, Halsted Press.

Profozich, D. M., Sturrock, D. T., Introduction to SIMAN/Cinema, *Proceedings of the 1995 Winter Simulation Conference*, eds., C. Alexopoulos, K. Kang, W. R. Lilegdon, D. Goldsman, 1995, pp. 515-518.

Rothenberg, J., Object-Oriented Simulation : Where do We Go From Here, *Proceedings of the 1986 Winter Simulation Conference*, eds., J. Wilson, J. Henriksen, S. Roberts, 1986, pp 464-469.

Sanderson, D. P., Sharma, R., Rozin, R., Treu, S., The Hierarchical Simulation Language HSL : A Versatile Tool for Process-Oriented Simulation, *ACM transactions on Modeling and Computer Simulation*, 1, 2, 1991, pp. 113-153.

Schwetman, H., CSIM18 – The Simulation Engine, *Proceedings of the 1996 Winter Simulation Conference*, eds., J. M. Charnes, D. J. Morrice, D. T. Brunner, J. J. Swain, 1996, pp 517-521.

Sethi, R., *Programming Languages, Concepts and Constructs*, 1990, Addison Wesley.

Sharma, R., Rose, L. L., Modular Design for Simulation, *Software-Practice and Experience,* 18, 10, 1988, pp. 945-966.

Simulation, Special Issue : Harbour and Maritime Simulation, *Simulation*, 71, 2, 1998.

Simulation, Special Issue : Object-Oriented Simulation, *Simulation,* 70, 6, 1998.

Standish, T. A., *Data Structures, Algorithms, and Software Principles*, 1994, Addison Wesley.

Stroustrup, B., *The C++ Programming Language*, deuxième édition, 1994, Adisson Wesley.

Swain, J. J., Simulation Survey : Tools for Process Understanding and Improvement, *OR/MS Today*, August 1995, pp. 64-79

Williams, E., Khoubyari, S., Modeling Issues in a Shipping System, *Proceedings of the 1996 Winter Simulation Conference*, eds., J. M. Charnes, D. J. Morrice, D. T. Brunner, J. J. Swain, 1996, pp. 1353-1357.

ANNEXE

Définition fonctionnelle des classes de la librairie

Classe Simulation

La classe Simulation gère la liste d'événements et l'horloge de la simulation. En conséquence, il existe obligatoirement un objet de cette classe par simulation. Bon usage est fait des paramètres par défaut pour définir des procédures qui permettent la planification d'événements avec ou sans priorité. De plus, les événements peuvent êtres planifiés en temps absolu ou relatif. Chaque insertion dans la liste d'événements retourne une notice d'événement qui peut ensuite être utilisée pour retracer, détruire ou modifier un événement particulier. Toutes les procédures de la classe Simulation sont héritées de la classe de base virtuelle Sim où elles sont déclarées comme pure virtuelle. Une référence à un objet de classe Simulation devrait toujours se faire par un pointeur de classe Sim pour garantir la compatibilité des programmes écrits.

---

template<unsigned int SzEvtLst> class Simulation : public Sim

La classe Simulation est un patron qui prend comme paramètre un entier positif non nul, SzEvtLst. Ce paramètre détermine la taille de la liste d'événements.

Simulation ( )

Constructeur de la classe Simulation. Initialise l'horloge de la simulation à zéro et alloue suffisament de mémoire pour une liste d'événements de taille SzEvtLst.

double Now ( )

Retourne l'heure courante sur l'horloge de la simulation.

EventNotice AbsSchedule (double when, Event *event, double priority)

Planifie l'événement event au temps when avec la priorité priority. La priorité s'exprime avec un entier positif non nul. Par défaut, priority a la valeur 10. Cette procédure retourne la notice d'événement de l'événement planifié.

EventNotice IncSchedule (double increment, Event *event, double priority)

Planifie l'événement event dans increment unité de temps avec la priorité priority. La priorité s'exprime avec un entier positif non nul. Par défaut, priority a la valeur 10. Cette procédure retourne la notice d'événement de l'événement planifié.

Event * FindEvent (EventNotice notice)

Retrouve l'événement désigné par notice dans la liste d'événements. Cette procédure retourne un pointeur vers l'événement en question. Notez que l'événement demeure dans la liste.

Event * Cancel (EventNotice notice)

Retrouve l'événement désigné par notice et le retire de la liste d'événements. Cette procédure retourne un pointeur vers l'événement en question. Notez qu'il peut être réinséré dans la liste d'événements avec AbsSchedule ou IncSchedule.

void Start ( )

Démarre la simulation. Les événements dans la liste d'événements sont exécutés en ordre chronologique. Les événements planifiés pour le même moment sont exécutés selon leur priorité et les événements de même priorité sont exécutés selon la discipline FIFO. Notez qu'après leur exécution, les événements sont automatiquement détruits.

void Stop (double time)

Termine la simulation au temps time. La priorité de l'événement qui signale la fin de la simulation est telle qu'il est toujours exécuté en premier pour un instant donné.

void RegisterNetwork (Network * net)

Cette procédure permet d'associer l'objet réseau net à la simulation. Il peut être interrogé en utilisant la procédure ReturnNetwork.

Network * ReturnNetwork ( )

Retourne l'objet réseau associé à la simulation.

Classe Event

La classe Event est la classe de base virtuelle obligatoire de tous les événements utilisés dans une simulation conçue avec la librairie. Elle définit comme pure virtuelle les trois procédures utilisées par la classe Simulation pour actualiser tous les événements.

---

int ReturnType_id ( )

Permet d'interroger un événement afin de connaître son type réel. Il doit être fixé par le constructeur de la classe dérivée.

virtual void EvtMain ( ) = 0

Cette procédure pure virtuelle est le corps de l'événement. Elle est la première étape dans l'actualisation d'un événement.

virtual void FollowUp ( ) = 0

Deuxième étape dans l'actualisation d'un événement. Cette procédure sert à planifier les événements qui découlent de l'actualisation de l'événement actuel.

virtual void CleanUp ( ) = 0

Dernière étape dans l'actualisation d'un événement. Cette procédure sert à libérer la mémoire utilisée par les structures internes de l'événement.

Classe Entity

La classe Entity est la classe de base obligatoire de toutes les entitités utilisées dans une simulation conçue avec la librairie. Le concept d'entité étant très large, cette classe ne possède pas une définition fonctionnelle propre. Elle est définie pour offrir un type commun qui permet de faire référence à des entités, peu importe leur type réel.

---

int ReturnType_id ( )

Permet d'interroger une entité afin de connaître son type réel. Il doit être fixé par le constructeur de la classe dérivée.

void BeginLife (double time)

Cette procédure marque le temps d'entrée dans le système de l'entité.

void EndLife (double time)

Cette procédure marque le temps de sortie du système de l'entité.

double ReturnLifeTime ( )

Cette procédure retourne le temps total passé dans le système par l'entité.

Classe Res et ses classes dérivées

La classe de base virtuelle Res a pour objectif de permettre la définition par les usagers de classes ressources sur mesure compatibles avec la librairie. Les classes EvtRes, QueueRes et PriorRes, qui sont respectivement des ressources sans file d'attente, avec file d'attente et avec file d'attente priorisée, sont dérivées de la classe Res. Elle définit les attributs standards d'une ressource, c'est à dire un identificateur, son niveau d'inventaire, sa capacité maximale ainsi qu'une valeur seuil qui indique un besoin de réapprovisionnement. De plus, des statistiques sont conservées sur le nombre de retraits et de dépôts acceptés ou rejetés. Lorsqu'approprié, des statistiques sur la file d'attente de la ressource sont aussi conservées. Il est important de remarquer qu'un objet de classe *Res* gère un seul type de ressource. La classe *ResList* est mise à la disposition des usagers pour manipuler un groupe de ressources. L'identificateur d'une ressource est alors utilisé pour la retrouver dans la liste.

---

int ReturnType_id ( )

Permet d'interroger une ressource afin de connaître son type réel. Il doit être fixé par le constructeur de la classe dérivée.

virtual MergeUnit *Deliver (float nbunits, Entity * who) = 0

Cette procédure pure virtuelle donne nbunits unités de ressources à who dans un objet de classe MergeUnit. Si la fonction retourne NULL, l'opération n'a pas fonctionné. C'est une erreur d'utiliser cette procédure avec un objet de classe PriorRes.

virtual MergeUnit *Deliver (double prior, float nbunits, Entity *who) = 0

Cette procédure pure virtuelle donne nbunits unités de ressources à who dans un objet de classe MergeUnit selon sa priorité prior. Si la fonction retourne NULL, l'opération n'a pas fonctionné. C'est une erreur d'utiliser cette procédure avec un objet de classe EvtRes ou QueueRes.

virtual int Return (MergeUnit *unit, Entity *who) = 0

Cette procédure pure virtuelle retourne à la ressource un objet de classe MergeUnit provenant de who.

virtual int NbInQueue ( ) = 0

Cette procédure pure virtuelle retourne le nombre d'objets de classe Entity dans la file d'attente. La valeur de retour est toujours 0 dans le cas de ressource sans file d'attente.

float GetStockLevel ( )

Cette procédure retourne le niveau de stock actuel de la ressource.

char *GetResID ( )

Cette procédure retourne l'identificateur de la ressource.

template<class Reorder,class StockFull> class EvtRes : public Res

Cette classe symbolise une ressource sans file d'attente. Les paramètres exigés sont les classes des événements à planifier automatiquement quand certaines situations se produisent. Un événement de classe Reorder est instigué quand le niveau d'inventaire de la ressource est inférieur ou égal au seuil de commande. Les événements de classe StockFull sont quant à eux planifiés quand un appel à la procédure Return excédant la capacité de la ressource se produit. Ces deux classes ont obligatoirement un constructeur dont les paramètres sont un pointeur qui identifie la ressource émettrice, un pointeur qui identifie l'entité impliquée dans l'événement et un pointeur vers l'objet Simulation qui gouverne la simulation.

```
EvtRes (    char *label, char *restype, float maxlevel, float stocklevel,
            float reorderlevel, Sim *engine )
```

Constructeur de la classe EvtRes. Crée une ressource nommée label qui procure un service ou des biens de type restype. Le niveau d'inventaire initial de la ressource est stocklevel, il ne peut excéder maxlevel et reorderlevel est le seuil de commande. Le paramètre engine est un pointeur vers l'objet Simulation qui gouverne la simulation.

```
template<class Reorder,class StockFull,class QueueFull,class Served>
class QueueRes : public Res
```

Cette classe symbolise une ressource avec une file d'attente. Les paramètres exigés sont les classes des événements à planifier automatiquement quand certaines situations se produisent. Les informations pertinentes aux classes Reorder et StockFull sont décrites dans la section sur la classe EvtRes. Les événements de classe QueueFull sont planifiés quand une entité trouve la file d'attente

pleine. Les événements de classe Served sont quant à eux planifiés quand une entité quitte la file d'attente. Les paramètres exigés pour le constructeur de la classe QueueFull sont les mêmes que pour Reorder et StockFull tandis que ceux de Served sont un pointeur vers une MergeUnit qui représente le service de l'entité, un pointeur vers l'entité en question et un pointeur vers l'objet Simulation qui gouverne la simulation.

```
QueueRes ( char *label,char *restype,float maxlevel,float stocklevel,
        float reorderlevel,QType qtype,int qsize,Sim *engine )
```

Constructeur de la classe QueueRes. Crée une ressource nommée label qui procure un service ou des biens de type restype. Le niveau d'inventaire initial de la ressource est stocklevel, il ne peut excéder maxlevel et reorderlevel est le seuil de commande. La ressource possède une file d'attente de taille qsize et de discipline qtype. Si qsize a la valeur 0, la file d'attente est illimitée. qtype peut prendre les valeurs FIFO ou LIFO. Le paramètre engine est un pointeur vers l'objet Simulation qui gouverne la simulation.

```
template<class Reorder,class StockFull,class QueueFull,class Served>
class PriorRes : public Res
```

Cette classe symbolise une ressource avec une file d'attente priorisée. Les paramètres de la classe sont les mêmes que pour la classe QueueRes.

```
PriorRes ( char *label,char *restype,float maxlevel,float stocklevel,
        float reorderlevel,QType qtype,int qsize,Sim *engine )
```

Constructeur de la classe PriorRes. Sa description est similaire à celle de QueueRes. QueueRes utilise la procédure Deliver sans le paramètre de priorité tandis que PriorRes utilise la procédure Deliver avec le paramètre de priorité.

Classe Link

La classe Link est la classe de base obligatoire de tous les arcs dans notre représentation d'une liste d'adjacence. Fait à remarquer, les objets de classe Link sont des arcs unidirectionnels. Les attributs de la classe Link indiquent son point de départ et d'arrivée, son poids, ainsi qu'une liste d'entités en transit.

---

int ReturnType_id ( )

Permet d'interroger un arc afin de connaître son type réel. Il doit être fixé par le constructeur de la classe dérivée.

Link (char *label,float weight,int capacity, Node *startpoint,Node *endpoint)

Constructeur de la classe Link. Crée un arc identifié label de poids weight entre startpoint et endpoint. Le paramètre capacity détermine la taille de la liste utilisée pour conserver les entités en transit. Une valeur de 0 signifie illimité.

char *GetLabel ( )

Cette procédure retourne l'identificateur de l'arc.

float GetWeight ( )

Cette procédure retourne le poids de l'arc.

void SetWeight (float weight)

Cette procédure permet de changer dynamiquement le poids de l'arc.

int GetCapacity ( )

Cette procédure retourne la taille maximale de la liste d'entités en transit.

int GetTransitLength ( )

Cette procédure retourne la taille de la liste d'entités en transit.

Node * GetStartpoint ( )

Cette procédure retourne un pointeur vers le noeud qui commence l'arc.

Node * GetEndpoint ( )

Cette procédure retourne un pointeur qui termine l'arc.

friend ostream& operator<<(ostream& s, const Link& x)

Opérateur de sortie pour cette classe. Imprime un sommaire des informations sur le lien.

Classe Node

La classe Node est la classe de base obligatoire de tous les noeuds dans notre représentation d'une liste d'adjacence. Les attributs qu'elle définit sont les degrés d'entrée et de sortie du noeud (In-degrees,out-degrees), la liste des objets ressources locaux disponibles, la liste des entités présentes ainsi que la liste des arcs sortant.

---

int ReturnType_id ( )

Permet d'interroger un noeud afin de connaître son type réel. Il doit être fixé par le constructeur de la classe dérivée.

Node (char *label)

Constructeur de la classe. Crée un noeud identifié label.

char *GetLabel();

Cette procédure retourne l'identificateur du noeud.

int GetOutDegree ( ), void IncOutDegree ( ), void DecOutDegree ( )

Respectivement, ces procédures retournent, incrémentent et décrémentent le dégré de sortie d'un noeud.

int GetInDegree ( ), void IncInDegree ( ), void DecInDegree ( )

Respectivement, ces procédures retournent, incrémentent et décrémentent le dégré d'entrée d'un noeud.

Link *AddLink (char *label,float weight,int capacity,Node *endpoint)

Cette procédure crée un arc identifié label de poids weight et de capacité capacity débutant à ce noeud et se terminant au noeud endpoint.

int RemoveLink (char *endlabel)

Cette procédure détruit l'arc entre ce noeud et le noeud identifié endlabel. Retourne 1 en cas d'erreur.

int AddRes (Res *ressource)

Cette procédure ajoute ressource à la liste de ressources du noeud. Retourne 1 en cas d'erreur.

int RemRes (char *label)

Cette procédure retire la ressource identifiée label de la liste de ressources du noeud. Retourne 1 en cas d'erreur.

Res *FindRes (char *label)

Cette procédure retourne un pointeur vers la ressource identifiée label.

friend ostream& operator<<(ostream& s, const Node& x)

Opérateur de sortie pour cette classe. Imprime un sommaire des informations sur ce noeud.

Classe Network

La classe *Network* représente une liste d'adjacence dans notre librairie. Elle contient une liste de noeuds ainsi que l'interface permettant la gestion du réseau, c'est-à-dire ajouter, retirer, et localiser des arcs, des noeuds et des entités (ressources). Une procédure est disponible pour récursivement parcourir et imprimer des informations sur tous les membres du réseau.

---

int ReturnType_id ( )

Permet d'interroger un réseau afin de connaître son type réel. Il doit être fixé par le constructeur de la classe dérivée.

Network (char *label,Sim *engine)

Constructeur de la classe Network. Le réseau créé est identifié label. Le paramètre engine pointe sur l'objet Simulation qui gère la simulation associée.

Sim *ReturnEngine ( )

Cette procédure retourne un pointeur vers l'objet Simulation qui gère la simulation.

Node *AddNode (char *label)

Cette procédure ajoute un noeud identifié label au réseau. Un pointeur vers le noeud ainsi créé est retourner afin de permettre la fin de sa description.

```
Node *FindNode (char *label)
```

Cette procédure retourne un pointeur vers le noeud identifié label dans le réseau.

```
int RemoveNode (char *label)
```

Cette procédure retire le noeud identifié label du réseau. Tous les arcs y faisant référence sont aussi détruits. Retourne 0 si le noeud label n'existe pas.

```
Link *AddLink (char *label,char *startlab,char *destlab,
                float weight,float capacity )
```

Cette procédure ajoute un arc identifié label de poids weight et de capacité capacity débutant au noeud startlab et se terminant au noeud destlab. Si l'arc existe déjà, il n'est pas recréé. La procédure retourne un pointeur vers l'arc correspondant au paramètre.

```
void Add2Link(char *labelS1,char *labelS2,char *lab1,
                char *lab2,float weight,float capacity);
```

Cette procédure ajoute deux arcs identifiés labelS1 et labelS2 de poids weight et de capacité capacity entre le noeud lab1 et le noeud lab2. labelS1 identifie l'arc débutant à lab1. Si un des arcs existe déjà, il n'est pas recréé.

int RemoveLink (char *startlabel,char *endlabel)

Cette procédure retire du réseau l'arc débutant à startlabel et se terminant à endlabel. Retourne 1 si un tel arc n'existe pas.

int AddRes (char *Nlabel,Res *ressource)

Cette procédure ajoute ressource à la liste de ressources du noeud identifié Nlabel. Retourne 1 si le noeud Nlabel n'existe pas.

int RemRes (char *Nlabel,char *Rlabel)

Cette procédure retire la ressource identifiée Rlabel de la liste de ressources du noeuds Nlabel. Retourne 1 si une telle ressource n'existe pas.

friend ostream& operator<<(ostream& s, const Node& x)

Opérateur de sortie pour cette classe. Imprime un sommaire des informations sur le réseau en utilisant l'opérateur << des noeuds et des liens.

Classe RandGen

La classe *RandGen* contient un générateur aléatoire multi-flots ainsi que les procédures nécessaires pour obtenir des valeurs assujetties aux lois de probabilité communes.

RandGen ( )

Constructeur de la classe. Le générateur aléatoire est initialisé de façon à contenir 100 générateurs aléatoires virtuels qui contiennent chacun 2^{31} sous-séquence de longueur 2^{41} nombres aléatoires.

void SetInitialSeed (long s[4])

Remplace le germe par défaut du générateur 0 par s. Ces valeurs doivent satisfaire les règles suivantes : 1 <= s[0] <= 2147483646, 1 <= s[1] <= 2147483542, 1 <= s[2] <= 2147483422, 1 <= s[3] <= 2147483322. Les germes des autres générateurs sont recalculés en conséquence. Tous les générateurs sont réinitialisés à leur première sous-séquence.

void InitGenerator(Gen g, SeedType Where)

Réinitialise le générateur g selon la valeur de Where. Where peut prendre les valeurs InitialSeed, LastSeed et NewSeed qui désignent respectivement le début de la sous-séquence initial du générateur g, le début de la sous-séquence actuelle du générateur g ou le début de la prochaine sous-séquence du générateur g.

void SetSeed (Gen g, long s[4])

Remplace le germe par défaut du générateur g par s. Les remarques de la procédure SetInitialSeed s'appliquent ici aussi. Notez qu'après l'usage de cette fonction les générateurs ne sont plus séparés par 2^{31} fois 2^{41} nombres aléatoires.

void GetState (Gen g, long s[4])

Cette procédure écrit dans le tableau s le germe du générateur g.

double Uniform01 (Gen g)

Cette procédure retourne une variable aléatoire uniforme sur [0,1] en utilisant le générateur g.

double Uniform (float a,float b,Gen g)

Cette procédure retourne une variable aléatoire uniforme sur [a, b] en utilisant le générateur g.

double Expon (float Beta, Gen g)

Cette procédure retourne une variable aléatoire exponentielle de moyenne Beta en utilisant le générateur g.

double Weibull(float Alpha, float Beta, Gen g)

Cette procédure retourne une variable aléatoire de distribution Weibull de forme Alpha et d'échelle Beta en utilisant le générateur g.

double Gamma (float Alpha, float Beta, Gen g)

Cette procédure retourne une variable aléatoire de distribution Weibull de forme Alpha et d'échelle Beta en utilisant le générateur g.

double Normal (double mean, double var, Gen g)

Cette procédure retourne une variable aléatoire normale de moyenne mean et de variance var en utilisant le générateur g.

double LogNormal(double mu, double sigma_squared, Gen g)

Cette procédure retourne une variable aléatoire de distribution lognormale de forme sigma_squared et d'échelle mu en utilisant le générateur g.

double Beta(double alpha1, double alpha2, Gen g)

Cette procédure retourne une variable aléatoire de distribution Beta avec les paramètres de forme alpha1 et alpha2 en utilisant le générateur g.

int Bernoulli (double p, Gen g)

Cette procédure retourne une variable aléatoire de Bernoulli avec probabilité de succès p en utilisant le générateur g.

long DiscUniform (int a,int b, Gen g)

Cette procédure retourne une variable aléatoire uniforme discrète sur [a, b] en utilisant le générateur g.

long Binomial (int t,double p, Gen g)

Cette procédure retourne une variable aléatoire binomiale de paramètre t, le nombre d'essais et p, la probabilité de succès en utilisant le générateur g.

long Geometric (double p, Gen g)

Cette procédure retourne une variable aléatoire géométrique avec probabilité de succès p en utilisant le générateur g.

double Poisson (double lamda, Gen g)

Cette procédure retourne une variable aléatoire de Poisson de paramètre lamda en utilisant le générateur g.

Classe StatBlock et ses classes dérivées

La classe virtuelle *StatBlock* se veut un standard pour permettre la définition de classes statistiques par les usagers. Elle définit comme pures virtuelles les fonctions Log et Report, fonctions qui servent respectivement à rapporter une observation et à déclencher la création d'un rapport sur le bloc statistique. La classe *DiscStatBlock* permet de recueillir des statistiques sur des phénomèmes discrets. Les informations disponibles sur une séquence d'observations sont sa taille, les observations minimales et maximales ainsi que la moyenne et la variance. Il est important de mentionner que le calcul de la moyenne et de la variance présume l'indépendance de la séquence d'observations. La classe *TimeStatBlock* se spécialise quant à elle dans l'étude des variations d'un attribut par rapport au temps. Les statistiques disponibles sont l'intervalle de temps sur lequel l'attribut a été examiné ainsi que sa valeur minimale, maximale et moyenne.

---

virtual void Log (double obsv) = 0

Cette procédure pure virtuelle doit être définie par tous les blocs statistiques de la librairie. Elle sert à rapporter une observation.

virtual void Report () = 0

Cette procédure pure virtuelle doit être définie par tous les blocs statistiques de la librairie. Elle sert à imprimer un rapport sur le bloc statistique.

DiscStatBlock(char *label)

Constructeur de la classe DiscStatBlock. Recueille des statistiques sur des observations discrètes. Le paramètre label est utilisé pour identifier le bloc.

DiscStatBlock : : double ReturnMean ()

Cette procédure retourne la moyenne des observations enregistrées dans un bloc statistique discret. L'appel de cette procédure quand le nombre d'observations est zéro cause une erreur.

DiscStatBlock : : double ReturnVar ()

Cette procédure retourne la variance des observations enregistrées dans un bloc statistique discret. L'appel de cette procédure quand le nombre d'observations est zéro cause une erreur.

DiscStatBlock : : unsigned long ReturnNbObsv()

Cette procédure retourne le nombre d'observations dans un bloc statistique discret.

DiscStatBlock : : void Reset ( )

Cette procédure reinitialise un bloc statistique discret.

TimeStatBlock(char *label,double initvalue,Sim *engine);

Constructeur de la classe TimeStatBlock. Cette classe recueille des statistiques sur des quantités continues. L'objet créé possède

l'identificateur label et la valeur initiale initvalue. Le paramètre engine est un pointeur vers l'objet horloge de la simulation.

TimeStatBlock : : void Stop ( )

Cette procédure marque le moment auquel un bloc continu cesse d'être actif.

TimeStatBlock : : void Reset ( )

Cette procédure reinitialise un bloc statistique continu.