Université de Montréal

SSJ : un cadre d'application pour la simulation stochastique en Java

par

Lakhdar Meliani

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

Avril, 2002

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé :
SSJ : Un cadre d'application pour la simulation stochastique en java

présenté par :
Lakhdar Meliani

a été évalué par un jury composé des personnes suivantes:

Michel Gendreau
.....
Président-rapporteur

Pierre L'Écuyer
.....
Directeur de recherche

Sang Nguyen
.....
Codirecteur de recherche

Jean Vaucher
.....
Membre de jury

Mémoire accepté le 12 août 2002

RESUMÉ

Dans ce mémoire, nous présentons SSJ, un cadre d'application pour la simulation stochastique basé sur le langage Java. Il supporte l'approche par événements, par processus, la simulation continue et la simulation mixte. SSJ est conçu pour être performant, simple, extensible et flexible. Il est basé conceptuellement sur la librairie SIMOD développée par P. L'Ecuyer sur la base du langage Modula 2. Nous discutons la conception et l'implémentation de SSJ et nous détaillons les alternatives d'implémentations auxquelles nous avons été confrontés. L'accent est mis sur les mesures et choix d'implémentation qui ont une influence significative sur les performances. L'effet de ces choix et mesures est illustré par des tests de performance. Des exemples de complexité variée sont développés pour montrer les possibilités de SSJ. D'autres librairies de simulation en Java sont aussi présentées à des fins de comparaison avec SSJ. Le choix du langage Java et l'apport de la programmation orientée objet sont abordés en détails. Les ressemblances et différences avec SIMOD sont mentionnées au fur et à mesure. Nous concluons que SSJ surpasse beaucoup les produits similaires en termes de performance, puissance, simplicité et extensibilité. Enfin nous donnerons quelques perspectives des développements futurs et des extensions possibles.

MOTS CLÉS

Approche par événement, approche d'interaction de processus, simulation continue, simulation mixte, performance de Java, simulation orientée objet, logiciel de simulation, thread, nombre aléatoire, variable aléatoire.

ABSTRACT

The aim of this thesis is to present SSJ, a well-designed set of classes and interfaces for stochastic simulation implemented in the Java programming language. It supports the event view, process view, continuous simulation, and arbitrary mixtures of these. SSJ is designed to be simple, efficient, extensible and flexible. Conceptually, SSJ is based on SIMOD, a library developed by P. L'Ecuyer and based on the Modula-2 programming language. We will discuss the implementation and design of SSJ and we will explain some choices and alternatives we have made during the implementation process. More importance is given to the implementation choices which have significant influence on the overall performance of SSJ. The effects of these choices are illustrated through performance tests. Examples of various level of complexity are developed to show the possibilities of SSJ. Some other Java libraries for stochastic simulation proposed in the last few years are presented and compared to SSJ. The choice of Java and the benefits of object oriented programming are discussed in detail. Common and different aspects between SSJ and SIMOD are mentioned along the way. Finally, we conclude that SSJ is better than a wide variety of similar products according to criteria such as performance, ease of use, flexibility and extensibility. Some perspectives of future work and possible extension are given at the end of this thesis.

KEY WORDS

Event view, process view, continuous simulation, combined simulation, Java performance, object oriented simulation, simulation software, thread, random numbers, random variate.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

DÉDICACE

A :

La mémoire de ma mère.

Mon père.

Mes frères et soeurs.

A Lamia aussi, pour son amour, sa patience et son dévouement.

REMERCIEMENTS

Je tiens à exprimer ma vive gratitude pour mon directeur de recherche P. L'Ecuyer, tant pour l'attention permanente et le suivi rigoureux que pour les idées et conseils qui ont grandement enrichi ce travail.

Je tiens aussi à exprimer ma reconnaissance et mes remerciements pour mon co-directeur S. Nguyen pour sa courtoisie et son appui tout au long de ce travail.

Mes remerciements s'adressent aussi à J. Vaucher pour ses critiques vives et constructives, ses conseils et discussions enrichissantes qui ont largement contribuée à l'aboutissement de ce travail.

Je remercie aussi tous mes amis pour leurs encouragements et innombrables services, spécialement A. Rouane Hacene.

Je tiens aussi à ce que monsieur R. Simard trouve ici mes remerciements.

En fin, que tous ceux et toutes celles que je n'ai pas cités, mais qui n'ont pas moins contribué d'une manière ou d'une autre à la réalisation de ce travail trouvent ici l'expression de ma vive reconnaissance.

Chapitre 1

Introduction

1.1 Objectif

L'objectif de ce travail est de concevoir et de développer un cadre d'application (en Anglais «Framework») orienté objet pour la simulation stochastique, basé sur Java et baptisé SSJ (Simulation Stochastique en Java). Ce cadre d'application dispose des outils nécessaires pour la simulation de différents types de systèmes complexes (systèmes discrets, continus et mixtes) et il supporte les deux approches populaires de simulation à savoir l'approche événementielle et celle d'interaction de processus.

Conceptuellement, SSJ se base sur la librairie SIMOD, développée par L'Ecuyer (L'Ecuyer [1988]) sur la base du langage Modula 2, et qui est à notre avis un outil puissant et facile à utiliser. SSJ se veut une suite et une amélioration de SIMOD, en tirant profit au maximum des possibilités et des avantages du langage Java et de l'approche orientée objet. Il n'est pas un langage de simulation à part entière mais une extension de Java, ce qui lui permet d'hériter de toute la souplesse, la puissance et la flexibilité d'un langage universel, très répandu et très disponible.

Les modèles de simulation peuvent être programmés dans différents langages, incluant les langages tout usage, les langages spécialisés ou les environnements graphiques dédiés à la simulation (voir chapitre 2). Les langages spécialisés et les environnements graphiques fournissent des outils très puissants pour la simulation des systèmes. Néanmoins, ils présentent certaines limitations par rapport aux langages tout usage. Ces derniers s'avèrent beaucoup plus flexibles et permettent de refléter certains aspects complexes des systèmes. Les langages et environnements spécialisés sont moins disponibles, coûtent beaucoup plus cher que les langages universels et leur utilisation

nécessite du temps et des efforts d'apprentissage. Ceci est un élément non négligeable surtout pour un usage occasionnel. Une bonne alternative consiste à développer des outils équivalents à ceux fournis par les langages spécialisés sans perdre les avantages offerts par un langage tout usage. C'est ce que fait SSJ en développant un cadre d'application de simulation basée sur Java.

1.2 Qu'est-ce que la Simulation?

La simulation est l'un des outils les plus puissants pour l'étude et l'analyse des processus et systèmes complexes. Elle devient une méthodologie de résolution de problème indispensable pour les ingénieurs, les concepteurs et les gestionnaires (voir Shannon [1998]). Il y a plusieurs définitions de la simulation, on cite celle de Shannon [1998] :

> «We will define simulation as the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system ».

Une solution analytique aux problèmes est parfois envisageable dans le cas où les relations qui composent le modèle du système à simuler sont très simples. Mais, comme la plupart des systèmes du monde réel sont trop complexes pour être évalués analytiquement, on a recours aux techniques de simulation (Law et Kelton [2000]).

La simulation consiste à reproduire pas à pas l'évolution d'un système en fonction du temps selon la logique des changements d'état. Pour ce faire, on a conçu plusieurs approches de descriptions de cette logique selon que l'on décrive le système à simuler par les événements qui se produisent, les activités qui seront exécutées ou les processus en interaction. Ces approches sont connues, respectivement, par : approche par événements, approche d'interaction de processus et approche par balayage d'activités (voir Banks [1998], Law et Kelton [2000]). Les deux premières approches sont très utilisées et très populaires. La vision par processus est, dans la plupart des cas, le moyen le plus naturel pour décrire les systèmes complexes. Toutefois, il y a une certaine

catégorie de systèmes qui peuvent être modélisés facilement en utilisant la vision par événements. Cette dernière est parfois préférée parce qu'elle donne lieu à une exécution plus rapide que l'approche par processus. SSJ offre des mécanismes pour les deux approches et les deux approches peuvent être mélangées dans un même modèle.

L'approche par événements consiste d'abord à répertorier les différents types d'événements pouvant avoir lieu au cours de la vie du système. Et ensuite, décrire la logique des changements d'états correspondant à ces divers types d'événements, par des algorithmes ou routines. L'étude du comportement dynamique du système consistera à exécuter dans l'ordre chronologique la logique de changements d'états associés à chaque événement. Les dates d'événements sont ordonnées dans une liste d'événements (dit échéancier) et le temps de simulation progresse d'une date d'événement à la date de l'événement suivant. L'approche d'interaction de processus met l'accent sur le cheminement d'une entité à travers le système à partir de l'événement qui a provoqué son arrivée jusqu'à l'événement provoquant son départ. On note qu'un processus génère une suite d'événements et d'activités qui décrivent l'historique de la progression d'une entité dans le système. La simulation des systèmes en temps continu consiste généralement à décrire le système à simuler par des équations différentielles ou équations aux dérivées partielles, pour lesquelles la solution analytique n'est pas disponible (voir Law & Kelton [2000]).

1.3 Choix du langage Java

Java est un langage de programmation à usage universel entièrement orienté objet et «**multithread**». Il emprunte sa syntaxe au langage C++. Son approche orientée objet est très proche de celle utilisée par le langage Smalltalk, qui est lui-même basé sur Simula, l'un des premiers langages utilisant l'approche par objets. Java n'offre aucune facilité spécifique pour la simulation stochastique mais il dispose de beaucoup d'avantages qui en font, à notre avis, un langage très approprié pour écrire des programmes de simulation.

- Le noyau de Java est fourni avec une librairie de classes extensibles; ce qui permet de développer facilement des librairies ou des cadres d'applications comme une extension du langage lui-même.
- Java est très répandu et partout disponible; ce qui permet aux programmeurs de la simulation d'utiliser un langage familier et d'éviter d'investir du temps et des efforts pour apprendre un nouveau langage.
- Java fournit les mécanismes nécessaires pour une simulation orientée objet par ses concepts d'objet et de Thread. Cela est très avantageux pour les programmes de simulation qui doivent refléter une réalité composée d'objets progressant en parallèle et en collaboration pour effectuer une tâche donnée. Néanmoins, notre expérience montre que les Threads de Java ne sont pas très appropriés pour faire de la simulation (vision par processus). Java a introduit son concept de thread pour supporter des applications vraiment parallèles comme un serveur avec plusieurs clients. Pour tirer profit des architectures multi-processeurs, les nouvelles implantations des threads en Java utilisent les threads du système d'exploitation (threads natifs) pour exécuter efficacement les tâches indépendantes sur des processeurs différents. En simulation, le parallélisme qui peut exister entres les entités actives (du système réel) n'est qu'un pseudo-parallélisme simulé par un programme séquentiel qui s'exécute sur un seul processeur. En plus, un programme de simulation peut facilement demander plus de threads (natifs) que le maximum que l'on peut créer. (Voir chapitre 5, sect. 5.3 pour plus de détail).
- Java est un langage très portable, grâce à son mécanisme de Machine Virtuelle Java (MVJ). La machine virtuelle Java exécute sur n'importe quelle plate-forme un code intermédiaire (appelé «bytecode») généré par le compilateur Java. Ce code n'est pas performant au même titre qu'un code purement compilé comme celui généré par le compilateur C++, mais il est plus rapide qu'un code complètement interprété comme Perl. Cependant, et selon notre expérience, l'utilisation des nouveaux compilateurs disponibles sur le marché et l'adoption des bonnes stratégies de programmation donnent des performances comparables à celles d'autres langages performants tel que Modula 2 ou C++ (voir chapitre 5,

sect. 5.3). Le lecteur intéressé est référé, aussi, aux tests faits par Mangione [1998]; selon lesquels, confirmant ainsi notre expérience, les performances de Java sont comparables à celles de C++ dans beaucoup de domaines.

- Java est très avancé dans les technologies Web. Il permet de développer des programmes de simulation s'exécutant à l'échelle planétaire dans un environnement dynamique, graphique et multimédia (voir Sawhney et al. [1999], Nair et al. [1996]).

1.4 L'implémentation du cadre d'application SSJ

Dans le cadre de ce travail, nous avons d'abord étudié en détail tous les modules de SIMOD (L'Ecuyer [1988]), et puis consulté et étudié d'autres librairies de simulation basées sur Java et C++. Ces dernières ne sont, généralement, pas bien documentées, et dans la plupart des cas nous n'avons pas trouvé de guides d'utilisation détaillés facilitant leur compréhension et utilisation. Nous nous sommes basés, généralement, sur des articles qui décrivent sommairement ces librairies.

Ensuite, et dans l'objectif d'élaborer un cadre d'application robuste, puissant, simple et efficace, nous avons envisagé plusieurs possibilités et alternatives surtout pour une implémentation optimale et efficace de la notion d'événement, de processus et de synchronisation entre processus. Une grande importance était accordée à l'étude des performances. Ainsi, et suite à des suggestions de la part de J. Vaucher, de nombreuses mesures ont été entreprises et implantées pour augmenter l'efficacité des classes SSJ (voir chapitre 5, sect. 5.3). Pour implémenter efficacement ces mesures, nous avons été amenés à des ajustements multiples et répétitifs, qui consiste à déterminer les fragments de code critiques, chercher et implémenter d'autres façons efficaces (susceptible de donner des meilleurs résultats) et ainsi de suite jusqu'à l'aboutissement à des résultats jugés satisfaisants.

Nous avons, aussi, envisagé et étudié quelques possibilités pour une implémentation flexible et simple des générateurs de nombres aléatoires et de variables aléatoires suivants différentes lois de probabilité. En outre, nous avons veillé à ce que

l'architecture de SSJ soit très ouverte et très facile à étendre et à enrichir. Ainsi, une attention particulière a été accordée à la façon d'implémenter la liste d'événements, en donnant la possibilité à l'utilisateur de choisir une structure pour la liste d'événements parmi celles disponibles ou de définir ses propres structures.

Afin d'implémenter efficacement les mécanismes de synchronisation des processus, nous avons étudié et expérimenté plusieurs alternatives pour suspendre, activer ou tuer un processus. Le mécanisme de Thread en Java offre les éléments nécessaires pour la gestion et la synchronisation de processus concurrents. Toutefois, et à partir de la version 1.2 de Java, beaucoup de méthodes sont dépréciées (en anglais « deprecated ») à cause de leur caractère insécurisé. Ceci nous a amené à implémenter les mécanismes pour remplacer ces méthodes qui sont nécessaires pour le fonctionnement d'un programme de simulation basé sur la vision par processus. Aussi, nous avons exécuté plusieurs tests de performance pour comparer les coûts liés à différentes alternatives. Ainsi plusieurs solutions ont été écartées parce qu'elles n'étaient pas efficaces.

Nous avons opté pour une implémentation flexible et simple de la notion de liste d'événements qui est au cœur de toute simulation stochastique. Ainsi, nous avons jugé important d'implanter les outils nécessaires qui permettent à l'utilisateur de choisir ou de changer facilement la structure de la liste d'événements utilisée dans SSJ. Ce dernier utilise, pour le moment, une liste doublement chaînée, qui n'est pas très efficace. D'autres structures plus performantes (arbre binaire, arbre rouge et noir, splay tree, etc.) seront éventuellement disponibles. (Notons que beaucoup de structures pour la liste d'événements ont été développées et sont actuellement disponibles. Le développement de ces structures est réalisé dans un cadre hors de ce mémoire). Au moment de l'initialisation de la simulation (méthode qui initialise l'horloge et la liste d'événements), l'utilisateur peut choisir une structure à utiliser pour la liste d'événements.

Pour une implémentation flexible des générateurs de nombres (pseudo)aléatoires, nous avons opté pour une solution permettant à l'utilisateur de choisir ou changer facilement

le générateur à utiliser pour la simulation. Le générateur utilisé par défaut est un générateur récursif multiple (en anglais «multiple recursive generator ») d'ordre k = 3, utilisant 32 bits et appelé **Mrg32k3** (voir l'Ecuyer [1999]). Ce générateur est de bonne qualité statistique avec une période très longue approchant 2^{191} et bonnes propriétés structurelles. Cette solution permet aussi à l'utilisateur de définir et d'implanter avec facilité d'autres générateurs de nombres aléatoires en implémentant l'interface **RandomStream** qui décrit les fonctionnalités d'un générateur de nombres aléatoires uniformes.

Pour la génération de variables aléatoires suivant différentes lois de probabilité, nous avons étudié et implémenté plusieurs façons pour permettre à l'utilisateur une utilisation simple et souple des variables aléatoires. Nous avons opté pour une solution extensible. Cette solution permet à l'utilisateur de définir et d'ajouter facilement d'autres lois de probabilité. La plupart des langages et librairies orientés objet que nous avons consultés implémentent chaque type de variable aléatoire comme une classe séparée. Ces variables sont générées à partir d'un générateur fixe que l'utilisateur ne peut pas changer. Indépendamment de la qualité du générateur (qui parfois laisse à désirer), cette façon de procéder n'est pas toujours adéquate à notre avis. Nous croyons qu'il est, souvent, plus approprié et plus naturel de définir les lois de probabilité comme des méthodes agissant sur un générateur au lieu des classes séparées. De ce fait, SSJ implémente les lois de probabilité comme des méthodes agissant sur un générateur qui peut être initialisé ou changé par l'utilisateur. Toutefois, et par souci de souplesse, l'utilisateur a aussi la possibilité de définir les lois comme des classes séparées (voir génération des variables aléatoires dans le chapitre 5).

Pour montrer la puissance et la souplesse de SSJ, nous avons fait une étude comparative avec d'autres librairies, notamment avec la librairie Silk (Healy et Kilgore [1997,1998]). Cependant, nous avons passé beaucoup de temps pour faire des exemples en Silk et pour faire fonctionner la version académique de Silk dont nous disposons (voir chapitre 4). Pour les autres librairies, même si leur obtention était facile et gratuite, nous ne nous sommes pas arrivés à faire des exemples significatifs pour pouvoir faire

des comparaisons très poussées. Nous avons aussi développé quatre exemples de systèmes réels. Trois d'entre eux sont repris de SIMOD, avec ajustements. Ces exemples touchent plusieurs domaines et chacun d'eux illustre un aspect particulier de SSJ. Ils sont aussi utilisés pour faire des comparaisons avec les autres librairies et à évaluer les performances de SSJ.

Pour expliquer et éclaircir le titre choisi pour ce mémoire, nous tenons à faire une distinction entre le concept de cadre d'application et celui de librairie de classes. Les deux concepts deviennent répandus avec la popularité de l'approche objet et sont utilisés, principalement, pour augmenter la qualité des applications et réduire le temps et le coût de développement. Néanmoins, il y a des différences entre les deux technologies. Le lecteur intéressé peut consulter le livre de Fayad et al. [1999] qui est une bonne référence dans le domaine des cadres d'applications. Les cadres d'application deviennent très populaires ces dernières années. Selon Fayad et al. [1999], ils seront au cœur des technologies des logiciels du vingt et unième siècle. Un cadre d'application est réutilisable, c'est une application semi-complète qui peut être complétée et spécialisée pour produire des applications spécifiques (Johnson et Foote [1988]). Principalement, il y a trois différences significatives entre un cadre d'application et une librairie de classes (voir Joines et Roberts [1998]) :

- Une librairie implémente le comportement des classes. Un cadre d'application spécifie, en plus du comportement des classes, les règles et la logique qui gouvernent l'utilisation de ces classes. Ainsi, les classes de SSJ implémentent une bonne partie de la logique des applications de simulation.
- Les classes d'une librairie sont utilisées par la création d'instances et l'activation des fonctions membres. Un cadre d'application fait la même chose, mais en plus il contrôle les interactions entre les objets. Ainsi, l'utilisateur de SSJ écrit des programmes en instanciant des classes et en invoquant des méthodes mais l'interaction entre différents objets est effectuée derrière la scène par SSJ.
- L'utilisation d'une librairie implique une réutilisation de code alors que l'utilisation d'un cadre d'application implique une réutilisation aussi bien du code que de la conception.

1.5 Une vue globale de SSJ

Dans cette section, nous parlerons d'abord, de quelques lignes directrices qui ont accompagné la conception et le développement de SSJ. Ensuite, nous donnons une vue globale des classes et méthodes de SSJ. Une description détaillée de ces classes et méthodes est donnée dans l'annexe A.

SSJ a été conçu et développé selon une méthodologie dont le souci était :

- Simplicité d'utilisation : SSJ est conçu pour être très simple et facile à utiliser. En effet, tout programmeur qui connaît le langage Java peut facilement utiliser SSJ, pourvu qu'il soit familier avec les concepts de base de la simulation stochastique, étant donné qu'un programme SSJ est un programme Java ordinaire sans aucune restriction ou contrainte.
- Puissance : les objets de SSJ intègrent un pouvoir d'expression très fort qui reflète les fonctionnalités et les facettes d'un objet réel. Ce qui ramène à des programmes très compacts et faciles à comprendre. Ainsi, on croit que même les systèmes les plus complexes peuvent être simulés en SSJ avec un minimum de code.
- Extensibilité et flexibilité : l'architecture de SSJ est ouverte et extensible. En effet, l'utilisateur peut ajouter de nouvelles classes, étendre les classes existantes ou changer le comportement des objets déjà définis. Ce facteur est important pour répondre à l'évolution des besoins.
- Efficacité : Nous avons implanté des mécanismes efficaces pour rendre les programmes développés en SSJ très performants.

Dans SSJ, la classe **Sim** entretient l'horloge et la liste des événements, elle contient aussi le processus principal de la simulation qui gère et contrôle le déroulement de la simulation. La liste d'événements est une instance de l'une des implémentations de l'interface **EventList.** Cette interface définit la structure et les fonctionnalités de base de toute implémentation de la liste d'événements. L'utilisateur peut définir sa propre structure ou utiliser une parmi celles disponibles.

La classe **Event** fournit des outils pour effectuer une simulation avec vision par événements. La classe **Process** fournit des outils pour effectuer une simulation avec vision par processus. Ces outils incluent la gestion, la synchronisation et la communication entre processus. Pour chaque type d'événement ou de processus, l'utilisateur doit créer une classe héritant de la classe **Event** ou **Process** selon le cas, en redéfinissant la méthode **actions()**. Dans le cas d'un événement, la méthode **actions()** est invoquée au moment de l'occurrence de cet événement. Dans le cas d'un processus, **actions()** décrit le comportement de ce processus, elle sera activée dès que le processus débute son cheminement.

L'interface **RandomStream** fournit la structure et les fonctionnalités de base de tout générateur de nombres (pseudo)aléatoires uniformes composé de multiples « streams ». La classe **Mrg32k3** est une implémentation de cette interface. La classe **Variate** sert comme une classe de base aux autres classes implantant des générateurs selon différentes lois de probabilités. Elle contient une référence de type **RandomStream** (un générateur de nombres aléatoire uniforme). La classe **Random1**, qui hérite de la classe **Variate**, fournit des méthodes pour la génération des variables aléatoires selon plusieurs lois de probabilités. L'utilisateur peut implémenter d'autres variables aléatoires de plusieurs façons (voir chapitre 5).

La classe **Resource,** qui est un mécanisme de synchronisation de processus, représente un centre de service, dont la capacité correspond au nombre de serveurs. Ces serveurs sont tous identiques. À chaque objet de type **Resource** sont associées deux listes, soit la liste des processus en attente pour la ressource et la liste des processus en service. La liste d'attente, dont la politique de service peut être fixée par l'utilisateur, est de capacité infinie. Un processus peut demander un certain nombre d'unités d'une ressource, attend si ce nombre d'unités n'est pas disponible, utilise la ressource pour un certain temps, et enfin libère la ressource. Par défaut, tous les processus demandant des unités d'une ressource ont une priorité égale à 0. Toutefois, l'utilisateur peut attribuer une priorité à un processus donné (voir chapitre 5). Les statistiques peuvent être

recueillies automatiquement sur une ressource si l'utilisateur invoque la méthode **collectStat(true).**

Les classes **Bin** et **Condition** fournissent des mécanismes additionnels pour la synchronisation de processus. La classe **Bin** permet une relation de type producteur-consommateur entre processus. La classe **Condition** représente un indicateur booléen associé à une liste de processus en attente. Les processus en attente restent bloqués jusqu'à ce que l'état de l'indicateur devienne vrai. La classe **StatProbe** et ses sous-classes **Tally** et **Accumulate** fournissent des outils pour le recueil des statistiques, l'obtention des rapports et le calcul d'intervalles de confiance. La classe **Tally** est instanciée lorsqu'on s'intéresse à une séquence d'observations x_1, x_2, ... d'une variable. La classe **Accumulate** est instanciée lorsqu'on s'intéresse à l'évolution de la valeur d'une variable dans le temps $X(t)$, $t \geq 0$.

1.6 Plan du mémoire

Ce mémoire présente SSJ en détail. Il est composé de cinq autres chapitres en plus du présent chapitre. Le chapitre 2 aborde les outils logiciels de la simulation, leur historique et les différentes catégories de langages utilisés pour programmer la simulation. Une part importante sera consacrée à la simulation orientée objet. Le chapitre suivant présente quelques exemples d'utilisation de SSJ. Ces exemples montrent les possibilités de SSJ et chacun d'eux illustre un aspect particulier. Ils serviront à évaluer les performances et à faire des comparaisons. D'autres librairies de simulation basées sur Java seront discutées dans le chapitre 4 dans le but de faire des comparaisons avec SSJ. La conception et l'implémentation de SSJ feront l'objet du chapitre 5, dans lequel nous discutons les différentes alternatives de conception et d'implémentation de SSJ. L'accent sera mis sur quelques particularités et difficultés rencontrées tout au long du développement de SSJ. Quelques choix et mesures qui ont affecté l'efficacité et l'implémentation seront discutés dans le même chapitre. Par la suite, une conclusion générale du présent travail sera donnée dans le dernier chapitre.

Chapitre 2

Les Outils Logiciels de la Simulation

Beaucoup d'outils logiciels (langages, librairies, environnements) furent créés pour faciliter le développement et l'exécution de la simulation des systèmes réels. Des langages spécialisés furent développés pour fournir aux usagers les outils nécessaires pour écrire des programmes de simulation, notamment, GPSS (Gordon [1978], Schriber [1974]), SIMSCRIPT (Karr et al. [1965]), SIMULA (Dahl et Nygaard [1967], Russel [1976]), SIMAN (Pegden [1985]), SLAM (Pritsker [1986]), etc.

De même, beaucoup de librairies furent développées autour des langages universels les plus utilisés tels que Pascal, Fortran, C, C++, Modula 2, et Java. De telles librairies sont importantes dans le sens où elles fournissent à l'usager toute la puissance, la souplesse, l'efficacité et la simplicité d'un langage déjà familier à l'usager. Des exemples de telles librairies sont : GASP basé sur FORTRAN (Pritsker [1974]), SIMOD basé sur Modula 2 (L'Ecuyer [1988]), Sim++ basé sur le langage C++ (Jade [1992]), et Silk basé sur Java (Healy et Kilgore [1998]).

Dans ce chapitre nous allons, brièvement, passer en revue l'historique et les différents types de langages utilisés pour programmer la simulation. Par la suite, nous abordons l'apport du style de programmation orienté objet à la simulation et nous donnons quelques exemples de langages et librairies de simulation orientés objet.

2.1 Historique

Depuis les années 50, où les premières simulations informatiques commencèrent à voir le jour, jusqu'à présent, où la simulation devient une méthodologie reconnue et très utilisée, plusieurs étapes ont été franchies et beaucoup de réalisations ont été

accomplies. Selon Banks et al. [2000], l'histoire de la simulation a passé par six étapes importantes. Sa description des cinq premières étapes a été reprise de Nance [1995].

1. «*The Period of Search (1955-60)*»: la simulation était programmée en FORTRAN et d'autres langages universels sans aucun support spécifique.
2. «*The Advent (1961-65)*» : c'est la période où sont apparus les précurseurs des langages de simulation, tels le langage GPSS et les packages basés sur FORTRAN comme SIMSCRIPT et GASP.
3. «*The Formative Period (1966-70)*» : les concepts se sont raffinés et ont été revus pour une bonne représentation des approches de simulation. C'est durant cette période que SIMULA (Dahl et Nygaard [1967]), l'un des précurseurs des langages à objet, a introduit les concepts de classe et d'héritage.
4. «*The Expansion Period (1971-78)*» : ce fut l'époque des améliorations des langages de simulation existants. Ainsi il a eu le développement de GPSS/H, qui est beaucoup plus rapide que les versions précédentes de GPSS. Il intègre de nouvelles fonctionnalités telles qu'un débuggeur interactif. Des changements majeurs ont été apportés au langage GASP avec l'apparition de GASP IV (Pritsker [1974]). D'autres efforts ont été faits pour simplifier le processus de modélisation. Ainsi il y a eu le développement des générateurs de programmes interactifs et des interfaces homme-machine en langage naturel.
5. «*Consolidation and Regeneration (1979-86)*» : les langages de simulation ont été adaptés ou réécrits pour les micro-ordinateurs. Cette période a vu l'apparition de SLAM II (Pritsker [1986]) et SIMAN (Pegden [1985]), tous deux descendants de GASP.
6. «*The present Period (1987-...)*» : cette période est marquée par l'expansion des langages de simulation sur les PCs et l'émergence des environnements de simulation avec des interfaces graphiques, des outils de visualisation et d'animation tel que Arena (Sadowski et Bapat [1999]) et AutoMod (Banks [2000]).

2.2 Programmation de la simulation

Les outils et langages utilisés pour programmer la simulation peuvent être classifiés en 4 catégories : les langages universels de haut niveau comme FORTRAN et C++, les langages généraux de simulation comme GPSS et SLAM, les environnements de simulation comme Arena et les librairies de simulation définies autour des langages universels les plus utilisés. Le lecteur peut se référer à Banks et al. [2000], Biles [1995], Pooch et Wall [1993], et Kreutzer [1986] pour plus de détails.

Qu'il soit dans cette catégorie ou dans l'autre, un outil ou un langage de simulation doit permettre au moins un ensemble de services de base tels que :

- génération de nombres aléatoires,
- génération de variables aléatoires suivant différentes lois de probabilités,
- gestion d'une horloge pour la gestion du temps de la simulation,
- gestion d'une liste d'événements et la localisation de l'événement imminent à exécuter
- recueil de données et les rapports des résultats,
- détection des conditions d'erreur,
- pseudo-parallélisme,
- ...

2.2.1 Simulation dans les langages tout usage

La simulation à événements discrets est un processus séquentiel, donc représentable sur une machine séquentielle. A partir de là, on conçoit qu'une telle simulation puisse être décrite par n'importe quel langage de programmation général de haut niveau. De ce fait, plusieurs programmeurs choisissent un langage comme FORTRAN, C ou Pascal pour implémenter un programme de simulation. Les principales raisons d'utilisation de ces langages sont la disponibilité, la célébrité, la portabilité et les performances. FORTRAN et C++ étaient parmi les plus utilisés pour la simulation. Pour plus de détail au sujet des avantages et inconvénients de l'utilisation de tels langages, voir Pooch et Wall [1993] en particulier le chapitre 13.

En général, l'emploi d'un langage tout usage demande un effort de conception et de programmation énorme propre à la simulation, surtout pour la simulation des systèmes complexes. Un tel effort n'est justifié que dans le cadre de projets importants où les performances et la portabilité peuvent être primordiales. Law et Kelton [2000], page 203, font une comparaison entre les packages de simulation et l'utilisation des langages de haut niveau.

2.2.2 Simulation dans les langages dédiés

Assez tôt des langages de simulation furent créés pour fournir aux utilisateurs des outils nécessaires pour programmer la simulation. Ces langages apparaissent comme des extensions de langages généraux existants ou même comme des langages entièrement nouveaux. Ils minimisent le coût et les délais de réalisation. L'amélioration et la facilité d'utilisation de ces langages étaient parmi les facteurs de popularité de la simulation ces dernières années. Pour plus de détails sur les avantages et les inconvénients d'utiliser de tels langages, voir Pooch et Wall [1993], pages 302-303.

Malgré que les langages spécialisés fournissent des services équivalents, il existe des différences significatives entre eux (voir Pooch et Wall [1993]) :

- L'origine du langage qui peut être :
  - une extension d'un langage évolué tel que SIMSCRIPT qui étend FORTRAN,
  - un langage spécialisé et dédié à la simulation tel que SLAM.
- Les qualités de base du langage telles que la souplesse d'utilisation et la disponibilité sur différentes machines et systèmes.
- La manière de représenter le système à simuler qui consiste à considérer le système comme étant discret, continu ou combiné, la manière de voir le modèle en termes d'événements, activités ou processus et le niveau de détail du modèle.
- La manière de gérer le temps.
- La méthode de recueil et d'analyse de données.

GPSS fut le premier langage de simulation supportant l'approche d'interaction de processus. Il est hautement structuré et conçu spécialement pour la simulation de

systèmes de files d'attente. La description du modèle de simulation en GPSS se fait sous forme de blocs assemblés en un diagramme (Gordon [1979]). Les entités temporaires appelées transactions sont générées par des blocs spéciaux et circulent à travers le diagramme suivant les conditions et les délais spécifiés par ces blocs. La première version de GPSS fut développée au sein d'IBM en 1961 par Geoffrey Gordon (voir Gordon [1978]). Depuis, il a connu plusieurs versions. La plus populaire est GPSS/H (Schriber [1991]). Cette version est considérée comme très puissante et flexible. Elle est parmi les outils les plus utilisés en simulation.

Beaucoup d'autres langages spécialisés sont connus et utilisés pour programmer la simulation, notamment SLAM II (Pritsker [1986]), SIMSCRIPT (Kiviat et al. [1971]), SIMAN (Pegden et al. [1995]), etc.

2.2.3 Langages de simulation dédiés à une classe d'application

Cette catégorie de langages est conçue pour répondre à certains besoins spécifiques dans un domaine donné. Ces langages sont appelés traditionnellement des simulateurs. Par exemple, pour la productique (en Anglais « manufacturing »), on trouve AutoMod (Matthew 2000) et ProModel (Harrell et price [2000]); pour les systèmes de soins de santé, il y a entre autres MedModel (Harrell et Price [2000]); pour les systèmes de communication, on trouve SIMSATCOM (Murphy [2000]), etc.

2.2.4 Librairie de simulation autour des langages tout usages

Beaucoup de librairies furent développées autour des langages tout usage. De telles librairies constituent une extension des langages existants. Elles héritent de la souplesse et la puissance du langage étendu. Ces librairies sont un certain nombre de routines et fonctions qui s'ajoutent au langage sous-jacent. Des exemples de telles libraires sont PSIM (Vaucher 1984), SIMPAS (Bryant [1980]), SIMPascal (L'Ecuyer [1987]), DGTS (Frantz et Trott [1984]) basés sur Pasca, SAMOA (Lomow et Unger [1982]), QSIM (Mills [1984]), SIMADA (Cote [1990]) basés sur ADA, DISCO (Helsgaun [1980]), DEMOS (Birtwistle [1979]) basés sur SIMULA. C++SIM (Arjuna [1994]), CSIM18 (Schwetman [1996]), SIM++ (Jade [1992]) basés sur les langages C et C++.

2.2.5 Simulation continue

Beaucoup de produits logiciels furent conçus et développés spécifiquement pour la simulation des systèmes en temps continu. Des exemples de tels produits sont DYNAMO (Pugh [1963]), MIMIC (Petersen [1972]), CSSL, CSMP, etc. De plus, beaucoup de librairies et langages de simulation discrète contiennent des outils pour supporter la simulation des systèmes en temps continu. SIMSCRIPT II.5 (Delfosse [1976]), Arena (Pegden et al. [1995], AweSim (Pritsker et O'Reilly [1999]), Extend (Imagine [1997]) sont des exemples de langages qui supportent la simulation continue en plus de la simulation discrète.

2.3 Sélection des langages de simulation

L'une des décisions importantes dans un projet de simulation est le choix du langage à utiliser pour codifier la simulation. Si le langage sélectionné n'est pas flexible ou difficile à utiliser, le programme peut produire des résultats erronés et parfois le projet peut être carrément abandonné. Faire un choix adéquat parmi un large éventail de langages, est une question importante voire vitale pour le praticien de la simulation. Cet embarras de choix a incité la communauté de la simulation à déterminer des critères à considérer lors du choix d'un langage. Law et Kelton [2000], pages 208-215, discutent en détail quelques caractéristiques désirables pour un langage de simulation. Pour plus de détails au sujet de la sélection des langages de simulation, le lecteur peut se référer à Banks [1998a, 1998b] et à Nikoukaran et al. [1998]. Des guides d'outils de simulation sont publiés chaque année; par Exemple, IIE Solutions publie en mai de chaque année le guide « Simulation Software Buyer's Guide ».

2.4 Simulation orientée objet

Depuis l'introduction de Simula (Vaucher [1998], Pooley [1987]), un langage de simulation considéré comme le précurseur des langages à objet (Lapalme et Vaucher [1989]), l'intérêt et l'apport de la technique objet pour la simulation sont reconnus. En général l'intérêt de la programmation orientée objet au développement de logiciel est bien décrit dans la littérature (Booch [1995], Meyer [2000]). La réutilisabilité,

l'extensibilité, la modularité, la maintenabilité des programmes orientés objet plaident en faveur de cette technique spécialement pour la simulation. Les langages à objets fournissent ces avantages à travers des mécanismes d'héritage, de composition, d'encapsulation et de polymorphisme. Les modèles de simulation peuvent être construits à travers la définition et la réutilisation des classes d'objets correspondant aux objets du monde réel.

Un programme de simulation orienté objet est une collection d'objets actifs et autonomes qui interagissent l'un avec l'autre à travers le temps pour reproduire le comportement du système étudié (voir Joines and Roberts [1996], [1998], [1999]. Dans une programmation orientée objet, chaque objet contient deux parties : une partie descriptive et une partie action. La partie descriptive représente l'état de l'objet à un instant donné, alors que la partie action décrit les opérations que l'objet peut accomplir (le comportement de l'objet). L'interaction et la communication entre les objets se font par l'envoi de messages et l'activation des méthodes.

L'approche objet se base sur quelques concepts très faciles à assimiler et à mettre en œuvre. Parmi les concepts importants, on trouve l'encapsulation qui consiste à rendre invisibles les propriétés privées de l'objet. Les objets ayant la même structure et le même comportement sont regroupés en une classe. La création des objets à partir des classes s'appelle instanciation. Les liens entre objets sont de deux types : la composition, où un objet est composé d'autres objets (ce lien est de type « has-a »), et l'héritage qui peut représenter des inclusions entre ensembles ou une spécialisation/généralisation entre types (ce lien est de type « is-a »).

2.4.1 Apport de la programmation orientée objet à la simulation

Les logiciels de simulation bénéficient directement des progrès du génie logiciel. L'avènement et le succès qu'a connu le style de programmation orienté objet ont incité la communauté de la simulation à de nouvelles conceptions et implémentations de programmes de simulation basées sur ce style jugé très bénéfique. Voici quelques aspects de l'approche objet en simulation :

- **Conception orientée objet** : une conception orientée objet consiste à voir le monde en terme d'objets. Ceci, est généralement moins difficile et beaucoup plus naturel que la méthode traditionnelle qui consiste à décomposer le problème en un ensemble de fonctions (voir Joines et Roberts [1998]). Il y a une correspondance naturelle entre les objets du monde réel et le programme qui les implémente. Ceci est important pour la simulation où l'objectif principal est de représenter via un programme d'ordinateur un système réel. Rien ne peut être plus naturel que l'organisation de la structure d'un programme autour des objets simulés. Cet aspect du paradigme orienté-objet est probablement le plus significatif pour la simulation. Étant donné que le monde réel est constitué d'objets : employés, clients, machines, etc. alors quoi de mieux que de décrire les objets à simuler pour modéliser le monde réel en vue de le simuler. Dans la réalité, physique ou abstraite, les objets sont présents, les objets logiciels reflètent simplement ces objets externes.

- **Maintenabilité** : la maintenabilité est parmi les aspects les plus importants dans la production des programmes de qualité (Mayer [1990]). Ainsi les expériences ont montré que les projets de simulation sont de nature évolutive : les besoins changent avec le temps, les systèmes simulés changent, et les objectifs du projet changent aussi au cours de la vie du programme. Un programme orienté objet bien conçu est moins résistant à un tel changement et la maintenance de tels programmes est plus facile en général.

- **Réutilisabilité** : cet aspect, très important dans tous les domaines de l'ingénierie et très bénéfique pour la production des programmes de qualité, consiste à réutiliser partiellement ou entièrement une conception ou un programme déjà existant dans l'implémentation de nouveaux programmes. Dans la simulation orientée objet, les objets peuvent être utilisés et réutilisés dans des projets futurs pour un gain de temps et de coût.
- **Extensibilité** : l'une des limitations des langages non orientés objet est le manque de flexibilité et d'extensibilité. Les langages à objets sont facilement

extensibles en permettant à l'utilisateur d'étendre les classes existantes ou de changer le comportement des objets existants. Ceci est réalisé par les mécanismes d'héritage et de la redéfinition des méthodes (voir Joines and Roberts [1998]).

De ce qui précède, il résulte que le développement de programmes de simulation orientés objet est fort intéressant car il permet de remédier aux carences des langages classiques (non orientés objet) de la simulation aussi bien sur le plan de la conception (identification des objets du monde réel) que celui du développement. Ceci permet de produire des programmes de simulation répondants aux facteurs de qualité vus par le génie logiciel.

2.4.2 Quelques outils de simulation orientés objet

Plusieurs langages et librairies de simulation orientés objets ont été commercialisés ces dernières années. MODSIM III est un langage compilé et orienté objet pour la simulation de systèmes complexes. Il offre un environnement de développement graphique, visuel et interactif. Développé par CACI's Products Company (Wood et Tumay [1999]), il emprunte sa syntaxe aux langages Modula 2 et Ada, et les concepts de simulation au langage SIMSCRIPT.

Arena est un environnement graphique orienté objet bien élaboré, basé sur le langage SIMAN. Il est développé par Rockwell Software (Bapat et Swets [2000]) et conçu pour la simulation des systèmes discrets et continus. Les modèles de simulation sont construits à partir des objets graphiques appelés modules. Ces derniers sont représentés par des icones; les données associées peuvent être saisies à partir des boîtes de dialogues. SiMPLE++ est un environnement de simulation orienté objet pour la modélisation de systèmes de productique (en Anglais «manufacturing») (Kalasky et Levasseur [1997]).

Chapitre 3

Exemples de programmes utilisant SSJ

Nous présentons dans ce chapitre quelques exemples de programmes de simulation en Java utilisant SSJ. L'objectif est de montrer les possibilités de SSJ et d'illustrer son utilisation. Les exemples choisis mettront l'accent sur quelques aspects intéressants de SSJ (le lecteur peut au besoin se référer aux définitions fonctionnelles des classes et méthodes de SSJ données en annexe A). Ces exemples seront aussi utilisés dans les prochains chapitres pour évaluer les performances de SSJ d'une part et pour faire des comparaisons avec d'autres librairies d'autres parts.

Le premier exemple est classique et très simple, il s'agit d'une file d'attente à un seul serveur. Cet exemple est modélisé par événements et par processus; la version par événements montre l'utilisation des événements en SSJ. La version par processus illustre l'utilisation conjointe d'événements et de processus, elle montre aussi la synchronisation entre processus via le mécanisme **Resource**. Dans la section 3.2 nous présentons la simulation d'un serveur par lot. Cet exemple illustre l'utilisation de la classe **Bin** qui est un autre mécanisme de synchronisation de processus. La section 3.3 donne un programme de simulation d'un système à temps partagé qui montre la souplesse et la richesse du générateur de nombres aléatoires utilisé en SSJ. Il utilise des variables aléatoires communes pour comparer deux configurations différentes d'un système. L'exemple donné dans la section 3.4 illustre l'utilisation du SSJ pour la simulation continue, il illustre aussi l'utilisation d'événements dans la simulation continue. Notons que la plupart de ces exemples avaient été faits, d'abord, en SIMOD (L'Ecuyer [1988]). Dans certains cas, le texte des exemples est repris directement du guide de SIMOD avec adaptation. Quelques exemples ont été réajustés et commentés par P. L'Ecuyer pour l'enseignement de son cours de simulation (L'Ecuyer [2001a]). Les entêtes des déclarations des classes sont données en gras pour plus de lisibilité.

3.1 Une file d'attente à un seul serveur (M/M/1)

Cet exemple simule le fonctionnement d'une file d'attente à un seul serveur où les clients arrivent aléatoirement et ils sont servis un par un dans l'ordre de leur arrivée (FIFO, pour «First In, First Out»). On suppose que les temps d'inter-arrivées et les temps de service sont exponentiellement distribués et qu'ils sont mutuellement indépendants. La moyenne des temps d'inter-arrivées est de 1 minute, et celle des durées de service est de 0.8 minutes. Les clients qui arrivent pendant que le serveur est occupé doivent rejoindre la file d'attente. On voudrait simuler les 10^6 premières minutes du fonctionnement de cette file et de recueillir des statistiques telles que le temps d'attente moyen par client, la longueur moyenne de la file, etc. Ce modèle est dénoté dans la théorie des files d'attentes par M/M/1. Il existe des formules analytiques, pour ce modèle, qui permettent de calculer exactement des mesures telles que le temps d'attente moyen par client, la longueur moyenne de la file, etc., sur horizon infini.

Nous donnons deux programmes de simulation pour cet exemple : le premier utilise la vision par événements et le second utilise la vision par processus.

3.1.1 Vision par événements

La figure 3.1 est le programme de simulation du modèle M/M/1 utilisant la vision par événements. Chaque type d'événement est défini comme une classe héritant de la classe **Event**. La méthode **main()** crée une instance de la classe **QueueEv**. Cette dernière classe crée deux objets de type **Random1,** deux listes et deux blocs statistiques. Les deux listes **waitList** et **ServList** servent à référencer les clients en attente et ceux en service (dans notre cas, il y a toujours un seul client en service). Les deux objets **genArr** et **genServ** sont deux générateurs de valeurs aléatoires qui génèrent, respectivement, les temps d'inter-arrivées et les temps de service.

La classe **Arrival** définit le type d'événement d'arrivée d'un client. À chaque occurrence d'un événement de ce type, la méthode **actions()** est exécutée. Elle consiste à créer l'événement suivant du même type et à planifier son arrivée après un certain

temps déterminé par la variable aléatoire **genArr,** qui suit une loi exponentielle de moyenne 1.0. Ensuite, un objet client associé à cette arrivée est créé. Chaque objet client a deux attributs : **arrivTime** qui mémorise le temps d'arrivée du client, et **servTime** qui mémorise le temps de service. Si l'événement en cours trouve le serveur libre, le client passe directement en service et est inséré dans la liste de service **servList**, sinon le client doit attendre dans la file d'attente **waitList.**

La classe **Departure** définit l'événement départ d'un client. À chaque occurrence d'un événement de ce type, le client en service est retiré de la liste de service. S'il y a d'autres clients en attente, le premier client est retiré de **waitList** et inséré dans la liste de service **servList**. Les blocs statistiques sont mis à jour, le temps d'attente du client est donné comme une nouvelle observation à **custWaits** et la nouvelle taille de la file est donnée comme une observation à **totWait**.

L'objet **endOfsim** est l'événement marquant la fin de la simulation. La méthode **actions()** de cet événement imprime les rapports statistiques pour les deux blocs (voir Figure3.2); ensuite, la simulation est stoppée.

```
import kernel.*;
import statistics.*;
import random.*;

public class QueueEv {

    static final double meanArr     = 1.0;
    static final double meanServ    = 0.8;
    static final double timeHorizon = 1000000.0;
    Arrival   arrival   = new Arrival();
    Departure departure = new Departure();

    Random1 genArr     = new Random1 ();
    Random1 genServ    = new Random1 ();
    List    waitList   = new List ("Customers waiting in queue");
    List    servList   = new List ("Customers in service");
    Tally   custWaits  = new Tally ("Waiting times");
    Accumulate totWait = new Accumulate ("Size of queue");

    class Customer { double arrivTime, servTime; }
```

```
public static void main (String[] args) { new QueueEv(); }

public QueueEv() {
   Sim.init();
   endOfSim.schedule (timeHorizon);
   arrival.schedule (genArr.expon (meanArr));
   Sim.start();
   }

Event endOfSim = new Event() {
    public void actions () {
       custWaits.report();  totWait.report();
       Sim.stop();
       }
    };


class Arrival extends Event {
    public void actions() {
       arrival.schedule (genArr.expon (meanArr));
                                        // The next arrival.
       Customer cust = new Customer();  // Cust just arrived.
       cust.arrivTime = Sim.time();
       cust.servTime = genServ.expon (meanServ);
       if (servList.size() > 0) {       // Must join the queue.
           waitList.insert (cust, List.LAST);
           totWait.update (waitList.size());
       } else {                         // Starts service.
           servList.insert (cust, List.LAST);
           departure.schedule (cust.servTime);
           custWaits.update (0.0);
           }
       }
    }


class Departure extends Event {
    public void actions () {
       servList.remove (List.FIRST);
       if (waitList.size () > 0) {
           // Starts service for next one in queue.
           Customer cust = (Customer) waitList.remove (List.FIRST);
           servList.insert (cust, List.LAST);
           departure.schedule (cust.servTime);
           custWaits.update (Sim.time () - cust.arrivTime);
           totWait.update (waitList.size ());
           }
       }
    }
}
```

Figure 3.1 : Simulation par événements d'une file d'attente M/M/1

La figure 3.2 montre le résultat du programme **QueueEv.** On voit qu'à l'instant 10^6 de la simulation, 1001851 clients ont terminé leur attente. Le temps maximum d'attente fut de 50.573, le temps minimum fut de 0 et la moyenne fut de 3.289. La taille moyenne de la file d'attente fut de 3.295, la taille maximale est 53.0 et la taille minimale est 0.0.

```
REPORT on Tally stat. collector ==> Waiting times
         min          max       average     standard dev.   nb. obs
       0.000       50.573         3.289         4.091       1001851

REPORT on Accumulate stat. collector ==> Size of queue
      from time    to time         min           max          average
        0.00     1000000.00       0.000       53.000           3.295
```

Figure 3.2 : Résultat de simulation du programme **QueueEv**

3.1.2 Vision par processus

La figure 3.3 montre un programme de simulation du modèle M/M/1 avec vision par processus. Dans cet exemple, événements et processus sont mélangés. Les clients sont vus comme des processus dont le comportement est décrit par la méthode **actions()** de la classe **Customer**. Le serveur est une instance de la classe **Resource**. L'événement de type **EndOfSim** imprime un rapport sur l'utilisation de la ressource et arrête la simulation.

Quand la classe **QueueProc** est instanciée par la méthode **main()**, le programme crée une ressource portant le nom **server** et deux instances de type **Random1** : **genArr** pour générer les temps d'inter-arrivée et **genServ** pour générer les temps de service.

Le constructeur de la classe **QueueProc** initialise la simulation, invoque une collecte automatique de statistiques pour le serveur, planifie l'événement fin de simulation (une instance de **EndOfSim**) à l'instant 10^6, crée le premier client et planifie son activation; ensuite, il démarre la simulation.

```
import kernel.*;
import statistics.*;
import random.*;

public class QueueProc {
    static final double meanArr     = 1.0;
    static final double meanServ    = 0.8;
    static final double timeHorizon = 1000000.0;

    Resource server  = new Resource (1, "server");
    Random1  genArr  = new Random1 ();
    Random1  genServ = new Random1 ();

    public static void main (String[] args) {new QueueProc(); }

    public QueueProc () {
        Sim.init();
        server.collectStat (true);
        new EndOfSim().schedule (timeHorizon);
        new Customer().schedule (genArr.expon (meanArr));
        Sim.start();
        }

    class Customer extends Process {
        public void actions () {
            new Customer().schedule (genArr.expon (meanArr));
            server.request (1);
            delay (genServ.expon (meanServ));
            server.release (1);
            }
        }

    class EndOfSim extends Event {
        public void actions () {
            server.report();
            Sim.stop();
            }
        }
    }
```

Figure 3.3 : Simulation par processus d'une file d'attente M/M/1.

La méthode **actions()** de la classe **Customer** est invoquée à chaque fois qu'un client débute son parcours. Elle planifie l'arrivée du client suivant après un certain temps déterminé par la variable aléatoire produite par **genArr**. Ensuite, le client demande le serveur en invoquant la méthode **request()**, puis occupe le serveur pendant un certain temps en invoquant la méthode **delay()**. Lorsque le client appelle **request()**, il obtient le serveur si celui-ci est libre, sinon il suspend son exécution et s'insère dans la file d'attente. Le client poursuit son exécution dès que le serveur devient libre (ces

opérations sont effectuées par la méthode **request()** derrière la scène). Quand **delayProc()** est appelée, le processus se bloque un certain temps, équivalent à son temps de service. Ce temps de service est généré par la méthode **expon()** de l'objet **genServ.**

```
REPORT ON RESOURCE : server

   From time :     0.00   to time : 1000000.00
                    min         max      average  standard dev.  nb. obs.
   Capacity           1           1        1.000
   Utilization        0           1        0.802
   Queue Size         0          53        3.295
   Wait           0.000      50.573        3.289        4.091   1001851
   Service      2.9E-7      11.456        0.801        0.800   1001850
   Sojourn      9.8E-6      52.476        4.090        4.169   1001850
```

Figure 3.4 : Résultat du programme **QueueProc**

La figure 3.4 montre les résultats du programme **QueueProc**. On voit que la capacité de la ressource est toujours égale à 1, le nombre moyen de serveurs utilisés est de 0.802 et la longueur moyenne de la file d'attente est de 3.295. Le temps moyen d'attente dans la file, de service et de séjour dans le system sont, respectivement, 3.289, 0.801 et 4.090. On voit aussi que 1001851 clients ont débuté leur service et 1001850 entre eux ont quitté le système. Le rapport fournit également d'autres statistiques, telles que les écarts-types.

3.2 Un serveur par lot

Cet exemple est tiré de Healy et Kilgore [1998], il consiste en un simple flux d'arrivée d'entités, et deux serveurs de nature différente. Les deux serveurs doivent servir les clients dans l'ordre de leur arrivée. Chaque client doit passer d'abord par le premier serveur et puis par le deuxième. Une file d'attente est formée devant chaque serveur par les entités qui doivent attendre la disponibilité du serveur. Le premier serveur ne peut servir qu'une seule entité à la fois. Le deuxième sert les entités en groupes dont la taille varie entre 10 et 20 entités. Le deuxième serveur attend toujours jusqu'à ce qu'il y ait au

```
import kernel.*;
import statistics.*;
import random.*;

class BatchServer {
    static final double meanArr   = 1.0;
    static final double meanServ1 = 0.8;
    static double meanServ2 = 12.0;
    Random1  genArriv  = new Random1(); //times between arrivals.
    Random1  genServ1  = new Random1(); //service time for first server
    Resource server1   = new Resource(1,"Server 1");
    Server2  server2;
    Bin      tokens    = new Bin("Server 2 Bin ");
    Tally    statSejour  = new Tally("Time in System");

    public static void main(String[] args) { new BatchServer(); }

    public BatchServer () {
      server1.collectStat(true);
      Sim.init ();
      new Customer().schedule(0.0);
      (server2= new Server2()).schedule(0.0);
      end.schedule(10000.0);
      Sim.start();
    }

   Event end = new Event() {
      public void actions() {
         statSejour.report();
         server1.waitList().report();
         tokens.waitList().report();
         Sim.stop();
      }
    };


   class Customer extends Process {
       public void actions() {
          double arrivalTime=Sim.time();
          new Customer().schedule(genArriv.expon(meanArr));
          server1.request(1);
          delay(genServ1.expon(meanServ1));
          server1.release(1);
          if (tokens.waitList().size()>=9 && !server2.busy)
              server2.resume();
          tokens.take(1);
          statSejour.update(Sim.time()-arrivalTime);
       }
    }


   class Server2 extends Process {
       boolean  busy = false;
       Random1  genServ2  = new Random1();//  for service time
```

```
        public void actions() {
           tokens.waitList().collectStat(true);
           while (true) {
              if (tokens.waitList().size()<10) {
                  busy=false;
                  suspend();
              }
              busy=true;
              delay(genServ2.expon(meanServ2));
              tokens.put (Math.min (20, tokens.waitList().size()));
           }
        }
    }

} // end of BatchServer class
```

Figure 3.5 : Simulation du modèle de serveur par lot

moins 10 entités dans la file qui lui est associée. Si jamais, après avoir terminé le service d'un groupe, il trouve entre 10 et 20 entités dans la file, il les prend toutes. S'il en trouve plus que 20, il prend seulement les 20 premières. S'il trouve moins que 10 entités, il se bloque.

Un programme de simulation de ce modèle est donné à la figure 3.5. Le second serveur est vu comme un processus dont le comportement est décrit par l'objet **server2** (une instance de la classe **Server2** qui hérite de la classe **Process**). Le premier serveur est vu comme une ressource avec une capacité égale à 1. Le bloc statistique de type **Tally,** appelé **statSejour,** mesure les durées de séjours des clients dans le système. L'événement **end** imprime les rapports statistiques et met fin à la simulation à l'instant 10000.0. L'objet **tokens** de type **Bin** est utilisé pour synchroniser les processus clients et le processus **server2**. Chaque client, lorsqu'il quitte le premier serveur pour rejoindre le second, demande un jeton en invoquant la méthode **server2.take(1).** Les jetons sont générés par le processus **server2** en invoquant la méthode **tokens.put(Math.min(20, server2.waitList().size()))**. Ces jetons ne seront disponibles que lorsque le **server2** est libre et qu'il y a plus de 10 clients en attente pour le deuxième serveur. Un client, lorsqu'il exécute sa méthode **actions(),** mémorise son temps d'arrivée, crée et planifie

l'arrivée du prochain client, demande le premier serveur, l'utilise pour un temps égal à son temps de service, puis le libère. Ensuite, il incrémente le nombre des clients en attente pour le deuxième serveur. Si ce nombre est plus grand ou égal à 10, il envoie un signal d'activation au **server2** pour le réveiller. Par la suite, le client se bloque en attente d'un jeton disponible. Lorsqu'il y a un jeton disponible, il le prend, décrémente le nombre de clients en attente pour les jetons et attend pour un temps égal à son temps de service (le temps de service est généré par le processus **server2**). Enfin, avant de quitter le système, le client met à jour le bloc statistique **statSejour** en lui donnant la durée de séjour.

```
REPORT on Tally stat. collector ==> Time in System
         min          max       average     standard dev.   nb. obs
       0.053      135.621        26.374         19.330        9939


REPORT ON LIST :  Waiting List for Server 1
   From time:     0.00 to time:    10000.00
                   min          max       average   standard dev.   nb. Obs
   Size              0           31         3.163
   Sojourn         0.000       22.560       3.176        3.702         9962

REPORT ON LIST : Waiting List for Server 2 Bin

   From time:     0.00 to time:    10000.00
                   min          max       average   standard dev.   nb. Obs
   Size              0          129        22.285
   Sojourn        4.1E-3      133.446      22.403       18.803         9939
```

Figure 3.6 : Résultat du programme **BatchServer**

Le résultat du programme **BatchServer** est donné à la figure 3.6. On voit qu'à l'instant 10000.0, 9939 clients ont quitté le système après avoir terminé leur service. Le temps moyen de séjour dans le système est de 26.374 unités de temps de simulation. On voit aussi que 9962 clients ont terminé leur attente pour le premier serveur. Le temps moyen d'attente est 3.176 et la taille moyenne de cette file est 3.163 clients. Pour le deuxième stage de service, 9939 clients ont terminé leur service, le temps moyen de séjour pour ce serveur est 22.403 et la taille moyenne de la file d'attente est de 22.285 clients. À noter que les statiques affichées pour le deuxième serveur concernent les temps de séjour des

clients à ce serveur même s'ils sont collectés sur la file d'attente associée à ce serveur. Cela s'explique par le fait que les clients restent bloqués dans la file d'attente jusqu'à ce qu'ils quittent complètement le système.

3.3 Un système à temps partagé

Cet exemple est tiré de Law et Kelton [2000] et de L'Ecuyer [1988]. Soit un système informatique simplifié à temps partagé composé de T terminaux identiques et indépendants, utilisant un même serveur (un serveur de base de données ou un CPU). Supposons que tous les terminaux sont utilisés et que chaque utilisateur d'un terminal soumet une tâche au serveur, attend la réponse, puis réfléchit pendant une durée aléatoire avant de soumettre une nouvelle tâche, etc.

Nous supposons que le temps de réflexion est une variable aléatoire suivant la loi exponentielle de moyenne μ, tandis que le temps de réponse requis pour une tâche est une variable aléatoire de loi Weibull de paramètres α et λ. Les tâches en attente de service forment une file d'attente avec une politique de service de « round robin » avec quantum de taille q. Le fonctionnement de cette file d'attente est comme suit : lorsqu'une tâche s'empare du serveur, elle s'exécute pendant q secondes et retourne à la fin de la file. S'il lui reste moins que q secondes pour terminer son exécution (la valeur du quantum), elle conserve le serveur jusqu'à la fin de son exécution. Dans les deux cas, il faut aussi h secondes supplémentaires («overhead») pour l'enlever du serveur et, s'il y a lieu, y remettre une autre tâche.

Le temps de réponse d'une tâche est défini comme étant le temps écoulé entre l'instant où la tâche quitte son terminal et l'instant où se termine son enlèvement du serveur lorsqu'elle termine son exécution. On s'intéresse au temps de réponse moyen à l'état stationnaire. On simule le système jusqu'à ce que N tâches se soient terminées, en supposant qu'initialement tous les terminaux sont dans l'état de réflexion. Afin d'alléger le biais initial causé par cette dernière hypothèse, on commence à recueillir les statistiques seulement lorsque N_0 tâches seront terminées (on calcule le temps moyen de réponse pour $N-N_0$ tâches). Cette simulation sera répétée, indépendamment, R fois.

Supposons qu'on veut comparer la moyenne des temps de réponse pour deux configurations différentes de ce système. Une configuration est caractérisée par un vecteur de paramètres (T, q, h, μ, α, λ). Nous ferons R répétitions indépendantes pour chaque configuration. Pour des fins de comparaison, on va utiliser des variables aléatoires communes. On voudrait, en fait, que les différences de performances observées entre les deux configurations soient dues le plus exclusivement possible aux différences entre les configurations, et le moins possible aux fluctuations aléatoires. On doit s'assurer que d'une configuration à l'autre, la ième répétition utilise les mêmes séquences de valeurs pseudo-aléatoires U(0,1) de base. On va coupler les répétitions deux à deux. Soient R_{1i} et R_{2i} les temps de réponse moyens des ièmes répétitions pour les configurations 1 et 2, et soit $D_i = R_{1i} - R_{2i}$.

Les D_i sont des variables aléatoires indépendantes et on peut utiliser leurs valeurs pour obtenir un intervalle de confiance pour la différence d entre les temps de réponse des deux systèmes. Le fait d'avoir obtenu R_{1i} et R_{2i} en utilisant des valeurs aléatoires communes devrait, si R_{1i} et R_{2i} sont positivement corrélées, réduire la variance des D_i et la largeur de l'intervalle de confiance pour d.

Le programme de la figure 3.7 permet d'effectuer cette simulation. Pour des fins de simplicité, les données spécifiques à chaque configuration sont fixées dans le programme. Les deux configurations sont caractérisées, respectivement, par (20, 0.1, .001, 5, ½, 1) et (20, 0.2, .001, 5, ½, 1). Le programme fixe aussi les variables suivantes : N0 = 100, N = 1100, et R = 10 répétitions.

Chaque terminal est vu comme un processus. Le serveur est vu comme une ressource avec une capacité égale à 1. Les générateurs **genThink** et **genServ** de type **Random1** servent à générer les valeurs aléatoires, respectivement, pour le temps de réflexion de l'utilisateur d'un terminal et le temps de réponse requis pour une tâche. Le bloc statistique **meanInRep** sert à calculer, pour chaque répétition, la moyenne des temps de réponse de N-N0 tâches. Le bloc **statDiff** de type **Tally** est utilisé pour collecter les

```
import java.io.*;
import kernel.*;
import statistics.*;
import random.*;
public class TimeShared {
   int nbTerminal = 20;      // Number of terminals.
   double quantum;           // Quantum size.
   double overhead = 0.001; // Amount of overhead (h).
   double meanThink = 5.0;  // Mean thinking time.
   double alpha  = 0.5;
   double lambda = 1.0;      // Parameters of the Weibull service needs.
   int N      = 1100;        // Total number of tasks to simulate.
   int N0     = 100;         // Number of taks for warmup.
   int nbRep = 10;           // Number of simulation runs (R).
   int nbTasks;              // Number of tasks ended so far.
   double q1 = 0.1;          // Quantum size for first config.
   double q2 = 0.2;          // Quantum size for second config.

   Random1 genThink = new Random1 ("Gen. for thinking times");
   Random1 genServ  = new Random1 ("Gen. for service requirements");
   Resource server  = new Resource (1, "The server");
   Tally meanInRep  = new Tally ("Average for current run");
   Tally statDiff   = new Tally ("Differences on mean response times");

   double meanConf1[];       // To store the results for first config.

   public static void main (String[] args) { new TimeShared (); }

   public TimeShared () {
      meanConf1 = new  double[nbRep];
      for (int conf = 1; conf <= 2; conf++) {
         if (conf == 1) quantum = q1;
         else {             // Start second configuration: reset seeds.
            quantum = q2;
            genThink.resetStartStream ();
            genServ.resetStartStream ();
            }
         for (int rep = 0; rep < nbRep; rep++) {
            simulOneRun ();
            if (conf==1) meanConf1[rep] = meanInRep.average();
            else statDiff.update(meanConf1[rep] - meanInRep.average());
            genThink.resetNextSubstream ();
            genServ.resetNextSubstream ();
            }
         }
      statDiff.printConfIntStudent (0.9);
      }

   private void simulOneRun () {
      Sim.init();   server.init();
      meanInRep.init();
      nbTasks = 0;
      for (int i=1; i<=nbTerminal; i++) new Terminal().schedule (0.0);
      Sim.start();
      }
```

```
class Terminal extends Process {
    public void actions () {
       double arrivTime;     // Arrival time of current request.
       double timeNeeded;    // Server's time still needed for it.
       while (nbTasks < N) {
          delay (genThink.expon (meanThink));
          arrivTime = Sim.time ();
          timeNeeded = genServ.weibull (alpha, lambda);
          while (timeNeeded > quantum) {
             server.request (1);
             delay (quantum + overhead);
             timeNeeded -= quantum;
             server.release (1);
             }
          server.request(1);  // Here, timeNeeded <= quantum.
          delay (timeNeeded + overhead);
          server.release(1);
          nbTasks++;
          if (nbTasks>N0) meanInRep.update (Sim.time() - arrivTime);
                        // Take the observation if warmup is over.
          }
       Sim.stop();       // N tasks have now completed.
       }
    }
}
```

Figure 3.7 : Simulation d'un système à temps partagé

différences D_i des moyennes de temps de réponse entre les deux configurations, dans le but de calculer l'intervalle de confiance pour d. Le tableau **meanConf1** est utilisé pour mémoriser les valeurs R_{1i} correspondantes aux résultats de la première configuration.

Le constructeur de la classe **TimeShared** lance la simulation pour les deux configurations. La variable **conf** indique le numéro de la configuration courante. Pour chaque configuration, on fait **nbRep** répétitions. Lorsque la simulation de la première configuration est terminée, la variable **quantum** prend la valeur de q2 et les générateurs de nombres aléatoires sont réinitialisés à leurs points de départ, pour que les deux configurations utilisent les mêmes valeurs aléatoires. Pour chaque répétition, les générateurs sont remis à leur prochain « substream ». Ce qui permet d'assurer que les générateurs démarrent exactement du même point pour les deux répétitions.

Un terminal, lorsqu'il exécute sa méthode **actions()**, génère des tâches. Pour chaque tâche, le terminal attend un certain temps égal au temps de réflexion de l'utilisateur, mémorise le temps d'arrivée de cette tâche, et génère le temps de réponse requis pour cette tâche. Ensuite, tant que le temps restant pour que la tâche termine son travail, est supérieur au quantum, la tâche demande le serveur, attend un temps égal à **quantum+overhead** et puis libère le serveur pour rejoindre encore la file d'attente. Si le temps restant est inférieur au quantum, la tâche demande le serveur, attend un certain temps égal au temps **timeNeeded+overhead** et libère le serveur. La simulation est stoppée lorsque N tâches ont terminé leur travail.

```
REPORT on Tally stat. collector ==> Differences on mean response times
        min         max       average     standard dev.   nb. obs
      -0.134        0.369       0.168         0.175          10

  90.0% confidence interval for mean (      0.067,      0.269 )
```

Figure 3.8 : Résultat du programme **TimeShared.**

La figure 3.8 (résultat du programme **TimeShared**) donne la différence entre les temps de réponse moyens pour les deux configurations caractérisés par q=0.1 et q=0.2.

3.4 Simulation continue : un système proies-prédateurs

Cet exemple est tiré de L'Ecuyer [1988]. Le texte et le programme de cet exemple sont écrits par P. L'Ecuyer. Il s'agit d'un système classique constitué de deux types d'animaux : les proies et les prédateurs, les premiers étant la nourriture des seconds. Soient x(t) et z(t) les nombres respectifs de proies et de prédateurs au temps t. On suppose que l'évolution de ces variables se fait selon les équations différentielles suivantes :

$$x'(t) = rx(t) - cx(t)z(t)$$

$$z'(t) = -sz(t) + dx(t)z(t)$$

Avec pour valeurs initiales $x(0) = x_0 > 0$. À remarquer ici que nous faisons une approximation d'un système à espace d'états discret (dans le monde réel, le nombre

d'animaux de chaque type est entier) par un système à espace d'états continu. Il s'agit ici d'un système d'équations différentielles de Lotka-Volterra, que l'on pourrait en principe résoudre de façon analytique. Cependant, pour illustrer l'utilisation de la simulation continue en SSJ, nous allons le résoudre numériquement.

```
import kernel.*;
public class PreyPred {
    double r = 0.005, c = 0.00001, s = 0.01, d = 0.000005, h = 5.0;
    double x0 = 2000.0, z0 = 150.0;
    double horizon = 501.0;
    Continuous x = new Preys ();
    Continuous z = new Preds ();

    public static void main (String[] args) { new PreyPred (); }

    public PreyPred () {
        Sim.init();
        new EndOfSim().schedule (horizon);
        new PrintPoint().schedule (h);
        Continuous.selectIntegMethod (Continuous.RUNGEKUTTA4, h);
        x.startInteg (x0);    z.startInteg (z0);
        Sim.start();
    }

    public class Preys extends Continuous {
        public double deriv (double t) {
           return (r * value() - c * value() * z.value());
        }
    }

    public class Preds extends Continuous {
        public double deriv (double t) {
           return (-s * value() + d * x.value() * value());
        }
    }

    class PrintPoint extends Event {
        public void actions () {
            System.out.println (Sim.time() + "  " + x.value() + "  "
                                                + z.value());
            this.schedule (h);
        }
    }


    class EndOfSim extends Event {
        public void actions () { Sim.stop(); }
    }
}
```

Figure 3.9 : Simulation d'un système proies-prédateurs

La figure 3.9 permet d'effectuer la simulation de ce modèle. Les données d'entrées sont placées directement dans le programme. La méthode **deriv**() des classes **Preys** et **Preds** calculent la dérivée en fonction du temps des variables continues x et z. Les variables x et z sont des instances des classes **Preys** et **Preds.** L'événement de type **PrintPoint** s'exécute à chaque pas d'intégration, et permet d'imprimer les triplets (t, x(t), z(t)). Dans ce programme, on utilise la méthode Runge Kutta d'ordre 4.

Chapitre 4

Autres Logiciels de Simulation en Java

L'émergence de Java comme langage de programmation à usage universel, portable, sécurisé et gratuit, et le dynamisme qu'il a engendré au sein de la communauté informatique, ont incité les programmeurs de simulation à réécrire de nouveaux logiciels de simulation pour tirer profit des avantages offerts par ce langage.

Plusieurs outils de simulation à événements discrets basés sur Java ont été développés récemment. Ils sont dans la plupart des cas des versions Java d'outils déjà existants basés sur d'autres langages. Chacun de ces outils implémente les concepts de la simulation à sa manière. Les critères importants à prendre en considération dans le choix de ces logiciels sont la flexibilité, la simplicité, la facilité d'utilisation, la robustesse et les performances.

Dans ce chapitre, nous présentons quelques librairies de simulation basées sur Java. Nous abordons la librairie Silk avec un peu plus de détails que les autres librairies (notons que Silk est un cadre d'application plus qu'une librairie, mais nous préférons le terme librairie pour garder les mêmes appellations utilisées dans les publications qui lui sont consacrées). Nous donnons quelques exemples d'utilisation de cette librairie. Le but est de faire une comparaison entre Silk et SSJ. Le choix de Silk est justifié par le fait qu'il y a eu beaucoup de publications pour cette librairie par rapport aux autres d'une part. D'autre part, nous disposons d'une version académique pour faire et tester des exemples. La version académique de Silk1.2 est gratuite pour les universités, ce qui nous a permis d'en avoir une copie. Toutefois, nous avons eu beaucoup de difficultés à faire des exemples en Silk, malgré de nombreuses tentatives pour contacter Threadtec (Entre-prise conceptrice de Silk).

4.1 Simulation en Silk

Dans cette section, nous allons d'abord présenter la librairie Silk accompagnée de quelques exemples, puis faire une comparaison entre Silk et SSJ. Silk est un système utilisant Java pour la simulation. Il est constitué d'une librairie de classes pour la simulation à événements discrets supportant l'approche d'interaction de processus. Il a été développé et commercialisé par ThreadTec (Kilgore [2000], Healy et Kilgore [1997], [1998]). Il fournit des objets d'animation que l'utilisateur peut lier avec d'autres objets afin d'animer la simulation. Des classes pour la construction des modèles graphiques sont aussi fournies. Les programmes Silk peuvent être exécutés en application ou en **Applet**.

Les classes de base de Silk sont **Entity, Resource, Queue,** en plus de quelques classes pour la génération de variables aléatoires selon différentes lois de probabilités. La classe **Entity** correspond à la classe **Process** de SSJ et la classe **Resource** correspond à la classe portant le même nom en SSJ. Deux classes pour le recueil de statistiques, appelées **Observational** et **TimeDependent** sont fournies. Ces dernières classes correspondent, respectivement, aux classes **Tally** et **Accumulate** de SSJ.

Pour écrire un programme de simulation en Silk1.2, l'utilisateur doit disposer, en plus de la librairie Silk, d'un compilateur Java et d'une machine virtuelle compatible avec JDK 1.1. De plus, pour développer un programme de simulation sous forme d'application, l'utilisateur doit :

- Créer une classe **Simulation** qui hérite de la classe **Silk** prédéfinie en redéfinissant la méthode **run()**. Cette classe constitue le point de départ où la simulation débute. Elle est généralement utilisée pour l'instanciation et la planification du temps d'arrivé de la première entité, l'initialisation des variables globales de Silk et la déclaration des variables et objets publics qui doivent être disponibles pour toutes les autres classes.
- Créer pour chaque type d'entité une classe héritant de la classe **Entity** prédéfinie en redéfinissant la méthode **process()** pour cette nouvelle classe. Cette méthode sert de point de départ pour l'exécution des entités de ce type. Elle doit décrire le

comportement de l'entité dans le système. Cette méthode correspond à la méthode **actions()** en SSJ qui doit être redéfinie par l'utilisateur pour chaque type d'entité.

- Créer une classe séparée qui contient la méthode **main().** Cette classe doit définir une instance de la classe **Silk** et invoquer par la suite la méthode **begin()** sur cette instance (voir les exemples ci dessous).

4.1.1 Simulation d'une file d'attente M/M/1.

Le code source donné à la figure 4.1 est le programme Silk de l'exemple de la file d'attente M/M/1 (voir section 3.1). Le programme est inspiré de Healy et Kilgore [1997], avec adaptation. La classe **Customer** représente les clients. Les autres classes (**QueueProc** et **Simulation**) sont nécessaires pour chaque programme écrit en Silk. Chaque client est vu comme un processus dont les activités sont décrites par la méthode **process()**.

Les instructions **import** font référence aux paquetages Silk utilisés dans le programme. La classe **Customer** est déclarée comme une extension de la classe **Entity** prédéfinie. Elle déclare une variable qui contient le temps d'arrivée des entités de ce type. Les variables déclarées **static** sont des objets partagés par tous les objets entités de type **Customer**. Le serveur est vu comme une ressource. La file d'attente associée à la ressource est déclarée explicitement comme une instance de la classe prédéfinie **Queue**; le temps de service et le temps des inter-arrivées sont des instances de la classe **Exponential** qui sert à générer des variables aléatoires suivant la loi exponentielle. Quatre blocs statistiques sont crées, deux de type **Observational** et les deux autres sont des instances de **TimeDependent**. Ces blocs mesurent, respectivement, le temps moyen de séjour des clients dans le système, le temps moyen d'attente des clients, la longueur moyenne de la file et le taux d'occupation du serveur.

Un client, lorsqu'il débute son cheminement dans la méthode **process()**, crée et planifie l'arrivée du prochain client en utilisant la méthode **sample()** de la variable aléatoire représentant les inter-arrivées. Il mémorise son temps d'arrivé avant d'aller rejoindre la

```
import com.threadtec.silk.Silk;
public class QueueProc  {
   public static void main ( String args[] ) {
      Silk mySilk = new Silk();
      mySilk.begin( );
   }
}
```

```
import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

public class Customer extends Entity {
   double arrivalTime;
   static Resource    server     = new Resource ( "Server" );
   static Queue       inputQueue =   new Queue ( "Input Queue" );
   static Exponential servTime   = new Exponential ( 0.8 );  // mean
   static Exponential betweenArr = new Exponential ( 1.0 ); // mean
   static Observational sysTime  = new Observational("Time in system");
   static Observational timeInQueue=new Observational("Time in queue");
   static TimeDependent queueLength = new
                  TimeDependent(inputQueue.length,"Queue length");
   static TimeDependent serverUtil  = new
                  TimeDependent(server.numBusy, "Server utilization");

   //Customer Class process method used to model Entity behavior.
   public void process ( )  {
      create ( betweenArr.sample( ) );  // create next Customer
      arrivalTime = time;               // record time in attribute
      queue ( inputQueue );             // insert Customer into queue
      while ( condition( server.getAvailability( ) == 0 ) );
      seize( server );                  // decrease availability
      dequeue( inputQueue );            // remove Customer from queue
      timeInQueue.record( time - arrivalTime); // record status delay
      delay ( servTime.sample( ) );     // delay for service
      release ( server );               // increase availability
      sysTime.record( time - arrivalTime); // record time in system
      dispose( );
    }
}
```

```
import com.threadtec.silk.*;     // references Silk methods
public class Simulation extends Silk {
   public void init ( ) {
      Customer entCustomer;
      entCustomer = (Customer)newEntity( Customer.class );
      entCustomer.start( 0.0 );
   }
   public void run ( ) {
      //  initialize Silk variables
      setReplications(1); //End simulation at the end of 1 replication.
      setRunLength(1000000.0);//End of Simulation at 1000000.
    }
}
```

Figure 4.1 : Simulation du modèle de la file d'attente en Silk

file d'attente; puis il attend dans une boucle **while (condition)** tant que le serveur est occupé. Ensuite, il décrémente le nombre d'unités libres, sort de la file d'attente, donne une nouvelle observation au bloc mesurant la longueur de la file d'attente et occupe le serveur pour un certain temps égal à son temps de service. Par la suite, il libère le serveur et donne une nouvelle observation au bloc statistique mesurant la durée de séjour des clients dans le système avant de se détruire.

La classe **Simulation,** qui étend la classe prédéfinie **Silk**, est le point de départ de l'exécution du programme. La méthode **run()**, automatiquement appelée par Silk au lancement du programme, initialise les variables globales du programme. Le nombre de répétitions est initialisé à 1 et la fin de simulation est initialisée à 10^6. La méthode **init()** crée la première instance de type **Customer** et planifie son temps d'arrivée à l'instant 0. Cette méthode est appelée automatiquement au début de chaque répétition.

La classe **QueueProc** est la classe qui contient la méthode **main().** Elle sert à identifier le programme et démarrer la simulation. La méthode **main()** crée une instance de type **Silk** et invoque la méthode **begin()** sur cette instance**.** La structure de cette classe est toujours la même pour tous les programmes écrit en Silk.

```
Run number 1 over at time 1000000.0000000

Elapsed time  0:6:52.45

Observational Variables:

                                Standard
Identifier            Average   Deviation       Minimum       Maximum         Final       Count
Time in system      4.0362702   4.0919494     0.0000161    54.4242305     0.6247543      999262
Time in queue       3.2361256   4.0118105     0.0000000    53.7223276     0.3206053      999262


Time Dependent Variables:

                                  Standard
Identifier              Average   Deviation       Minimum       Maximum         Final     Time Period
Queue length          3.2337374   4.3947676     0.0000000    66.0000000     0.0000000 1000000.0000000
Server utilization    0.7995540   0.4003341     0.0000000     1.0000000     0.0000000 1000000.0000000
```

Figure 4.2 : Résultat du programme **QueueProc** en Silk

La figure 4.2 montre les résultats du programme **QueueProc** en Silk. À l'instant 10^6, Il y a eu 999262 clients qui ont terminé leur attente et leur service. Le temps de séjour moyen dans le système est de 4.036 et le temps moyen d'attente est de 3.236. La longueur moyenne de la file d'attente est de 3.233.

4.1.2 Simulation du modèle d'un serveur par lot

Le programme de simulation donné dans la figure 4.3 permet d'effectuer la simulation en Silk du modèle du serveur par lot tel qu'il est énoncé dans le chapitre 3 (sect. 3.3). Ce programme est repris de Healy et Kilgore [1998] avec adaptation.

Les clients sont vus comme des entités qui transitent par le système. Le premier serveur est modélisé comme une ressource; le second serveur est vu comme une entité active qui exécute une boucle infinie jusqu'à la fin de la simulation.

La classe **MyEntity** représente les clients qui transitent par le système et le comportement qui leur est associé pour le premier serveur. Elle déclare le temps d'arrivée de cette entité, le générateur des temps d'inter-arrivées, le générateur des durées de service, une file d'attente qui sera associée au premier serveur, une ressource pour modéliser le premier serveur, et deux blocs statistiques pour recueillir des données sur la longueur de la file et le temps de séjour des différents clients dans le système. Les attributs déclarés **static** sont partagés par tous les objets clients de type **MyEntity**.

Un client, lorsqu'il débute son cheminement dans la méthode **process()**, crée et planifie l'arrivée du prochain client en utilisant la méthode **sample()** de la variable aléatoire représentant les inter-arrivées. Il mémorise son temps d'arrivé avant d'aller rejoindre la file d'attente; puis il attend dans une boucle **while (condition)** tant que le serveur est occupé. Ensuite, il décrémente le nombre d'unités libres, sort de la file d'attente et occupe le serveur pour un certain temps égal à son temps de service. Lorsqu'il quitte le premier serveur, le client rejoint la deuxième file associée au deuxième serveur et active la méthode **halt()** qui suspend l'entité jusqu'au moment où elle sera réactivée par le deuxième serveur. Un client, après avoir été réactivé, continue son cheminement en

mettant à jour le bloc statistique mesurant le temps de séjour dans le système, ensuite il se détruit par l'appel à la méthode **dispose**().

La classe **BatchServer,** qui étend la classe **Entity,** représente le deuxième serveur. Elle déclare une variable d'état qui indique si le serveur est occupé ou non, une file d'attente qui lui sera associée, une file temporaire pour référencer les clients en service, une variable aléatoire pour générer les temps de service et deux blocs statistiques pour mesurer la longueur de la deuxième file et le taux d'occupation du serveur.

```
import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;

public class BatchServer extends Entity {

    int batchSize, IDLE=0, BUSY=1;
    IntStateVar isvStatus       = new IntStateVar( IDLE );
    public static Queue queue2  = new Queue( "Stage 2 Input Queue" );
    public static Queue queTemp = new Queue( "Temporary Queue" );
    Exponential genServ2        = new Exponential( 12.0 );
    TimeDependent tdQueue       = new TimeDependent( queue2.length,
                                    "Stage 2 Input Queue" );
    Observational obsBatchSize  = new Observational( "Batch Size" );
    Entity entTemp;

    public void process ( ) {
        while ( true ) {
           while( condition ( queue2.getLength( ) < 10 ) );
           batchSize = Math.min( 20, queue2.getLength( ) );
           obsBatchSize.record( batchSize );
           for ( int i=1;  i<=batchSize;  ++i ) {
               entTemp = (Entity)queue2.remove(1);
               queTemp.insert( entTemp );
           }
           isvStatus.setValue( BUSY );
           delay( genServ2.sample( ) );
           while( queTemp.getLength( ) > 0 ) {
                entTemp = (Entity)queTemp.remove(1);
                entTemp.activate( );
           }
           isvStatus.setValue( IDLE );
        }
    }

}
```

```
import com.threadtec.silk.*;
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;
public class MyEntity extends Entity {
    double attArvTime;
    static Exponential genArriv  = new Exponential( 1.0 );
    static Exponential genServ1  =  new Exponential ( 0.8 );
    static Queue queue1          = new Queue( "Stage 1 Input Queue" );
    static Resource server1      = new Resource ( " server 1" );
    static TimeDependent tdQueue = new TimeDependent(queue1.length,
                                   "Stage 1 Input Queue" );
    static Observational sysTime = new Observational("Time in System");

    public void process ( ) {
        create( genArriv.sample( ) );
        attArvTime = time;
        queue( queue1 );
        while ( condition( server1.getAvailability( ) == 0 ) );
        dequeue( queue1 );
        seize( server1 );
        delay( genServ1.sample( ) );
        release( server1 );
        queue ( BatchServer.queue2 );
        halt( );
        sysTime.record( time - attArvTime );
        dispose( );
    }
}
```

```
import com.threadtec.silk.*;
public class Simulation extends Silk {
   public void run ( ) {
      setReplications( 1 );
      setRunLength(10000);
      setControlConsole( true );
   }
   public void init( ) {
      MyEntity first = new MyEntity( );
      first.start( 0.0 );
      BatchServer only= new BatchServer( );
      only.start( 0.0 );
   }
}
```

```
import com.threadtec.silk.*;
public class Example {

   public static void main(String args[]){
      Silk mySilk = new Silk( );
      mySilk.begin();
    }
}
```

Figure 4.3 : Simulation en Silk du modèle serveur par lot

La méthode **process()** de la classe **BatchServer** décrit le comportement du second serveur. Elle exécute une boucle infinie jusqu'à la fin de la simulation. À chaque itération, le serveur attend dans une boucle de type **while (condition)** jusqu'à ce qu'il y ait au minimum 10 entités dans la file qui lui est associée. S'il y a plus que 10 entités dans la file, le serveur prend le minimum entre 20 et le nombre d'entité en attente. Ensuite, il active les entités une par une pour qu'elles poursuivent leur cheminement à partir de l'endroit où elles étaient suspendues par la méthode **halt()** dans la classe **MyEntity.**

```
Run number 1 over at time   10000.0000000

Elapsed time  0:4:18.19

Observational Variables:

                                Standard
Identifier              Average Deviation       Minimum       Maximum         Final Count
Time in System        8.3149190 4.6236506     0.0010392    36.3478669     0.3421304  9930
Batch Size           10.0100806 0.3175003    10.0000000    20.0000000    10.0000000   992


Time Dependent Variables:

                                      Standard
Identifier                 Average    Deviation       Minimum       Maximum         Final   Time Period
Stage 1 Input Queue      2.9328018    3.7033907     0.0000000    23.0000000     2.0000000  10000.0000000
Stage 2 Input Queue      4.4964299    2.8676772     0.0000000    20.0000000     9.0000000  10000.0000000
```

Figure 4.4 : Résultat de simulation du modèle de serveur par lot en Silk

La figure 4.4 montre le résultat du programme de simulation du modèle du serveur par lot. À la fin de la simulation il y a eu 9930 clients qui ont terminé leurs parcours et le deuxième serveur a servi 992 groupes de clients.

4.1.3 Comparaison entre Silk et SSJ

En étudiant ces deux exemples et en les comparant avec ceux de SSJ (voir chapitre 3), on trouve les différences suivantes :

- Performances : Pour comparer les performances entre SSJ et Silk, nous avons exécuté les exemples de la file d'attente et celui de serveur par lot sur un Celeron à 400 Mhz et 128 Mo de mémoire vive sous Linux Red Hat 6.2. Le tableau 4.1 présente les résultats de comparaison. Nous nous sommes servis de quelques fonctions natives écrites en langages C pour calculer les temps CPU consommés par

les exemples faits en SSJ. Pour les exemples de Silk, nous nous sommes basés sur le temps CPU calculé par Silk lui-même (voir plus haut les résultats des exemples). Ce temps ne calcule que le temps CPU pour l'exécution du modèle sans prendre en considération les instructions graphiques.

Exemples	Silk (en seconde)	SSJ (en seconde)
QueueEv (10^6 unités de temps)	-	15.02
QueueProc (10^6 unités de temps)	412.45	47.76
BatchServer (10^3 unités de temps)	2.99	0.51
BatchServer (10^4 unités de temps)	258.19	0.87

Tableau 4.1 : Comparaison de performance entre SSJ et Silk

Notons que les exemples faits en Silk s'exécutent toujours en mode graphique. Si on prend en considération le temps consommé par les instructions graphiques, le temps d'exécution sera un peu plus élevé que celui affiché dans le tableau 4.1.

- Approches de simulation supportées : Silk supporte seulement la vision par processus. SSJ supporte, en plus de la vision par processus, la vision par événements et la simulation continue. Les trois visions peuvent être combinées dans un même programme. La vision par événements est utile dans le cas où les performances sont primordiales. La possibilité de combiner les trois visions donnent plus de souplesse et permet de simuler des systèmes complexes qui ne peuvent pas être simulés facilement par la vision par processus toute seule.

- Utilisation de méthodes de Java dites «deprecated» : Dans la version Silk1.2 dont nous disposons, la synchronisation entre processus utilise les méthodes Java

dépréciées telles que **suspend()**, **resume()** et **stop().** Par contre, SSJ implémente les mécanismes nécessaires pour remplacer ces méthodes (voir chapitre 5).

- Synchronisation des processus : Silk dispose du concept de **Resource** pour la synchronisation de processus. SSJ dispose, en plus de ce même concept, de deux autres concepts pour la synchronisation de processus, à savoir le concept de **Bin** et de **Condition** (voir annexe A pour plus de détail). Ces deux concepts permettent de simuler des situations difficiles à simuler avec le concept de **Resource** (voir section 3.2 pour l'exemple de serveur par lot).

- Fonctions stochastiques : Silk implémente l'algorithme congruentiel linéaire multiplicatif pour générer des nombres aléatoires et la génération de variable aléatoire selon des lois de probabilités se base sur ce même générateur. Indépendamment de la qualité de ce générateur qui laisse à désirer, l'utilisateur ne peut pas utiliser un autre générateur. SSJ dispose, pour le moment du générateur de nombres aléatoires **Mrg32k3** qui est de qualité reconnue (voir L'Ecuyer [1999]). D'autres générateurs seront éventuellement disponibles, et l'utilisateur peut choisir très facilement le générateur qu'il désire. Notons que l'utilisateur peut aussi définir son propre générateur de nombre aléatoire et l'utiliser avec SSJ sans aucun effort spécifique. En plus, les générateurs de nombres aléatoires utilisables avec SSJ disposent de plusieurs «streams» que l'utilisateur peut manipuler facilement (voir annexe A). Ceci permet de comparer deux systèmes avec des variables aléatoires communes dans le cas où on voudrait que les différences entre ces deux systèmes ne soient pas influencées par les fluctuations aléatoires (voir l'exemple du système à temps partagé à la section 3.3). Cette comparaison est difficile à faire avec Silk.

En plus de ces différences importantes, il y en a d'autres moins importantes, parmi lesquelles :

- En Silk, les files d'attente associées aux ressources sont créées et gérées par l'utilisateur. Nous jugeons qu'il est plus naturel et plus simple de considérer les files d'attentes et les files des processus en service comme des propriétés de la ressource

elle-même et de gérer ces files d'une façon transparente vis-à-vis de l'utilisateur (voir les exemples en Silk plus haut et ceux de SSJ dans le chap. 3).

- L'appel de la méthode **seize**(), l'équivalent de la méthode **request()** en SSJ, doit être précédé par une boucle **while (condition)** pour s'assurer que les unités demandées sont libres. On ne comprend pas pour quelle raison l'utilisateur doit, à chaque fois, faire une boucle vide pour bloquer les processus alors qu'il est beaucoup plus facile et plus simple de réaliser ceci d'une façon complètement transparente vis-à-vis de l'utilisateur. En SSJ, les entités qui veulent obtenir des unités de service n'ont qu'à invoquer une seule méthode **request(int nb).**
- En SSJ, on peut demander une collecte automatique de statistiques sur une ressource, alors qu'en Silk, l'utilisateur doit prendre en charge la collecte de données statistiques.

4.2 Autres logiciels de simulation en Java

Cette section donne un bref aperçu de quelques librairies de simulation basées sur Java et que nous avons consultées et étudiées (brièvement) tout au long de notre recherche. Notons que la plupart de ces librairies sont fournies gratuitement et dans la plupart des cas le code source est disponible.

Simjava est une librairie de simulation à événements discrets basée sur Java et développée par Howell et McNab [1998a, 1998b]. Elle est écrite en Java et basée, conceptuellement, sur HASE++ (Coe et al. 1998) qui est lui-même basé sur Sim++, une autre librairie de simulation basée sur le langage C++ (Jade Simulation International [1992]). L'objet principal dans Simjava est **Sim_system** qui gère la simulation et coordonne l'exécution des autres entités. Il fait avancer le temps de la simulation et entretient la liste d'événements et la trace d'exécution est mémorisée dans un fichier. Les classes de type entité définies par l'utilisateur doivent hériter de la classe **Sim_entity** (prédéfinie) en redéfinissant la méthode **body().** Cette dernière méthode doit décrire le cheminement de l'entité en question. La communication entre les entités se fait par l'envoi d'objets de type **Sim_event** via un mécanisme dit « port » établi d'avance entre les entités. Écrire un programme en Simjava consiste à développer une

collection de classes d'entités héritant de la classe prédéfinie **Sim_entity**. Chaque entité s'exécute dans son propre Thread. Les entités sont reliées par le mécanisme des ports.

JSIM est un outil de simulation qui supporte la vision par processus et celle par événements (Nair et al. [1996], Miller et al.[1997, 1998]). Il supporte la simulation basée sur le Web (en anglais «Web-Based Simulation ») en permettant à l'utilisateur d'exécuter les modèles de simulation sur le Web en utilisant les **Applet** Java. Il dispose d'un environnement graphique pour les modèles d'attente et une base de données Java pour le stockage des modèles de simulation et les résultats associés. Quelques outils pour animer la simulation sont aussi fournis. Un modèle de simulation en JSIM est décrit par un graphe formé d'un ensemble de nœuds et de liens. Les nœuds représentent les notions de serveur et de file d'attente rencontrés dans les systèmes d'attentes. Les liens sont formés en utilisant une classe appelée **Transport** que les entités actives dans le système empruntent pour transiter dans le système. Chaque entité s'exécute dans son propre thread. En plus, JSIM fournit un paquetage appelé QDS («Query Driven Simulation») (voir Nair et al [1996]). Ce paquetage contient des outils pour stocker les modèles et résultats de simulation dans une base de données pour une utilisation ultérieure. Lorsqu'un utilisateur veut exécuter un modèle existant dans la base de données, les résultats seront retrouvés et présentés sans exécuter la simulation de nouveau.

JavaSim est un ensemble de classes Java supportant la vision par processus pour la simulation des systèmes discrets (Little 1999). Il est une implémentation Java de C++SIM (Arjuna [1994]) qui est lui-même basé sur le langage C++. *Simkit* (Buss [2001]) est une librairie pour la simulation à événement discret supportant uniquement la vision par événements. Il se base sur la méthodologie des graphes d'événements et il implémente les événements comme des méthodes au lieu des classes, en utilisant la technique de réflexion de Java pour déterminer la méthode à exécuter. *SimProd* (Kapuno et al. [1999]) est un outil de simulation des systèmes de production basé sur Simjava, il fournit les objets qui représentent les entités retrouvées dans les systèmes de production comme les machines, convoyeurs et autres. Les modèles de SimProd

peuvent être implémentés comme des **Applet** et exécutés sur le Web. *Simien* (Busse et Stork [1996]) fournit un ensemble de classes Java basées sur MODSIM II avec une bonne collection de classes statistiques mais sans outil d'animation. *DEVS-Java* (Zeigler [1997]) est une version Java de l'environnement DEVS-C++.

À titre comparatif, nous avons exécuté l'exemple de la file d'attente (une version de **QueueEv**) avec la librairie Simkit. Cet exemple prend 181.55 secondes sur une machine AMD Athlon à 750 Mhz fonctionnant sous Linux 7.0 et utilisant JDK-1.2.2 avec un compilateur JIT. Le même exemple, sur la même machine et le même JDK, prend 9.45 secondes avec SSJ. Cela montre que SSJ est approximativement 20 fois plus rapide que Simkit. Le choix de ce dernier est motivé par le fait que Simkit est très récent par rapport aux autres librairies et utilise la vision par événements qui devrait être rapide. En plus, le programme de l'exemple de la file d'attente est fourni avec la librairie (voir Buss [2001]).

L'étude de ces librairies (citée ci-dessus), permet de relever les faits suivants :

- Aucune libraire ne supporte la simulation continue.
- On ne peut pas combiner événement et processus dans le même modèle. Il va de soi que l'utilisation des événements avec la simulation continue n'est pas possible.
- Toutes ces librairies utilisent un générateur de nombres aléatoires fixe. Indépendamment de la qualité du générateur utilisé, l'utilisateur ne peut pas utiliser un autre générateur ou développer son propre générateur. Idem pour la structure de la liste d'événements.
- Toutes les libraires qui supportent la vision par processus utilisent les threads de Java pour implémenter la vision par processus en associant à chaque processus un thread indépendant. Nous n'avons pas trouvé de mesures particulières pour réduire les coûts liés à l'utilisation des threads de Java, surtout la création et la destruction des threads, qui sont des opérations très coûteuses en terme de performance (voir chapitre 5).

- La plupart de ces librairies ne permettent pas d'associer des priorités aux processus. Le mécanisme de synchronisation de processus se limite à un seul mécanisme, qui ne serait en aucun cas assez puissant pour répondre à tous les besoins.

4.3 Conclusion

Comparé à d'autres librairies, l'utilisation de SSJ pour écrire des programmes de simulation est très avantageuse. En effet SSJ :

- Est très performant, aussi bien avec la vision par événements qu'avec la vision par processus.
- Utilise un générateur de nombres aléatoires de bonne qualité composé de plusieurs «streams». Des routines sont fournies pour une manipulation facile de ces «streams».
- Dispose d'outils puissants pour simuler diverses catégories de systèmes réels (discrets, continus, mixtes) et supporte l'approche par événements et celle par processus, ainsi que la simulation continue. Les trois approches peuvent être combinées dans le même programme.
- Est plus flexible que beaucoup d'autres librairies de simulation en se basant sur une architecture ouverte surtout en ce qui concerne le choix de générateur de nombres aléatoires et le choix de la structure de la liste d'événements (voir annexe A).
- Est très simple et très facile à utiliser, étant donné que les programmes SSJ sont des programmes Java ordinaires sans aucune contrainte. SSJ se base sur peu de concepts faciles à comprendre et à mettre en œuvre.

Chapitre 5

Conception et Implémentation de SSJ

Dans ce chapitre, nous abordons l'élaboration de certaines parties importantes de SSJ. Nous décrivons l'architecture et l'organisation de ce dernier, et nous mettons l'accent sur quelques particularités ou difficultés d'implémentation. Nous discutons quelques alternatives d'implantation auxquelles nous avons été confrontés tout au long du développement de SSJ (le lecteur peut se référer, au besoin, à la description fonctionnelle des classes et des méthodes de SSJ dans l'annexe A).

Au cours de l'élaboration de SSJ, nous avons respecté l'esprit d'une bonne conception orientée objet et le style de programmation qui lui est associé. En effet, toute classe SSJ est une transcription d'un type d'objet réel ou abstrait. Les attributs d'une classe sont toujours encapsulés, excepté dans le cas où il serait plus efficace de déclarer un attribut public pour un accès rapide et direct. Les conventions d'écriture de programmes en langage Java ont été suivies.

Dans beaucoup de cas, nous avons fait des tests de vitesse pour estimer le coût de différentes alternatives de solution. Tous les tests de ce chapitre ont été effectués sur un AMD Athlon à 750 Mhz et 512 Mo de mémoire vive fonctionnant sous Linux Red-Hat 7.0. Nous nous sommes servis de quelques fonctions natives écrites en langages C pour calculer les temps d'exécution. Ces fonctions natives permettent de calculer le temps CPU consommé par un programme ou juste par une partie de programme. Sans indication contraire, tous les tests de ce chapitre sont effectués sur l'exemple de la file d'attente **QueueProc** qui sert approximativement 10^6 clients (voir chapitre 3 sect.3.1 pour la définition de cet exemple). Le temps des tests de performance sera donné toujours en secondes.

5.1 Le cadre d'application SSJ

SSJ est composé de quatre paquetages (voir figure 5.1) :

- **Paquetage kernel** : constitue le noyau de la simulation, l'inclusion de ce paquetage est nécessaire pour le fonctionnement de tout programme écrit en SSJ.
- **Paquetage random** : englobe les classes et les interfaces relatives à la génération des nombres pseudo-aléatoires et des variables aléatoires selon différentes lois de probabilités.

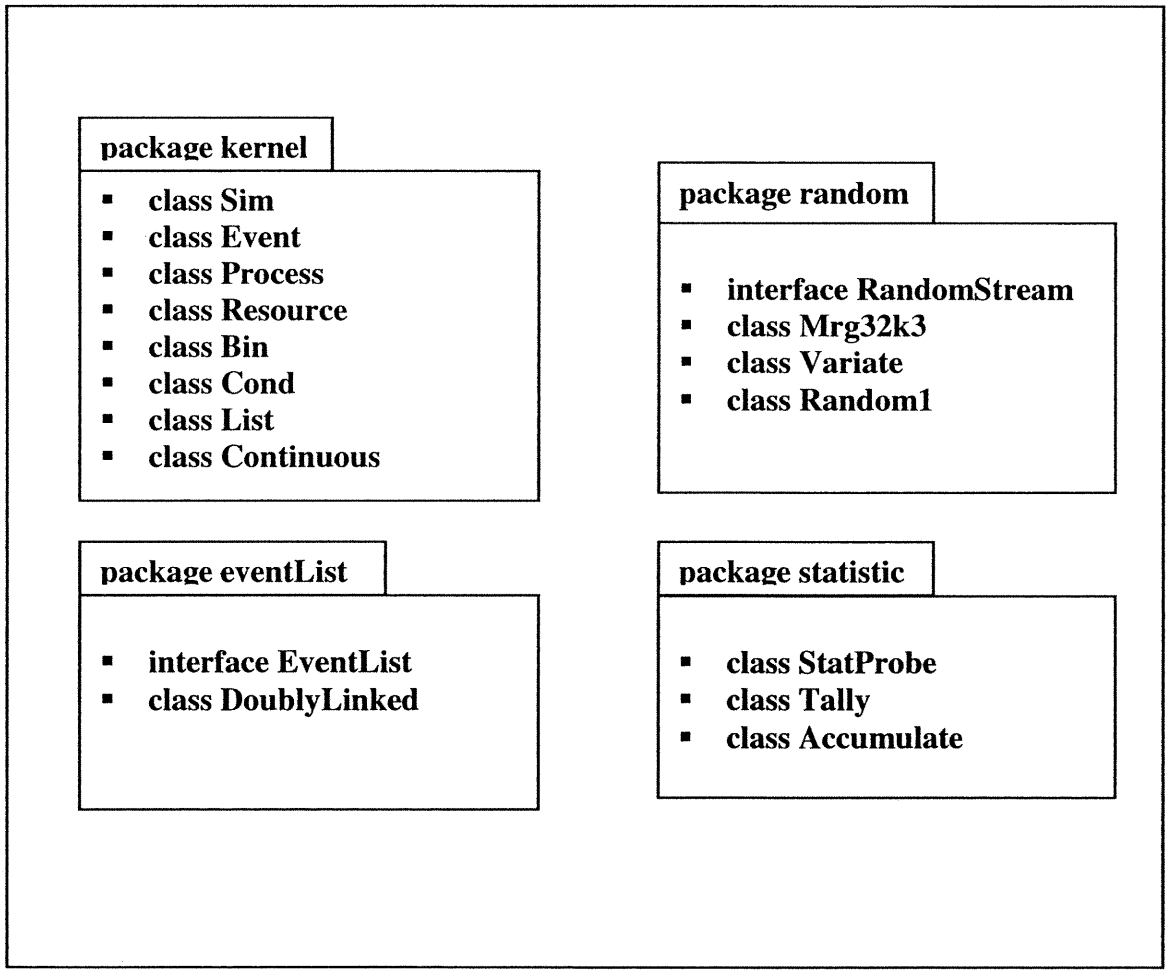


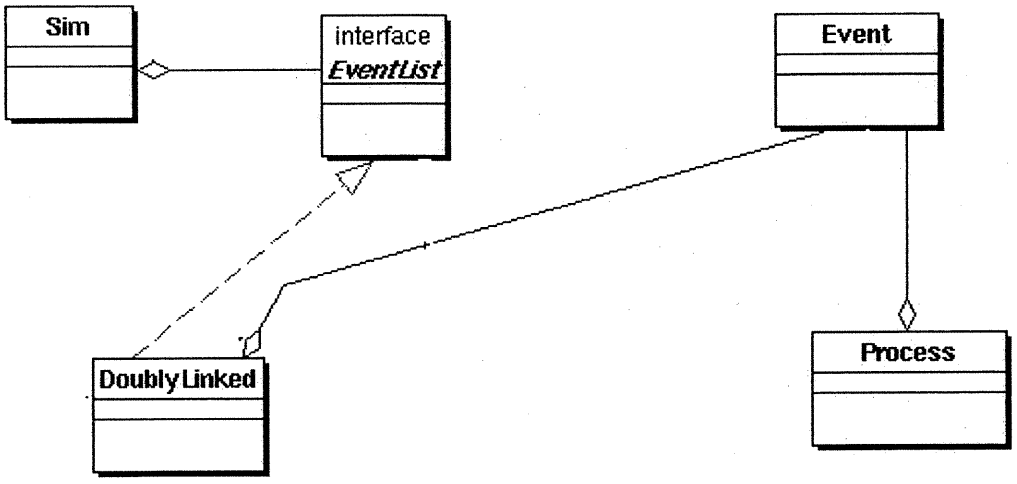
Figure 5.1 : Différents paquetages de SSJ

- **Paquetage eventList** : englobe les classes et les interfaces relatives à l'implémentation des structures de données pour la liste d'événements.
- **Paquetage statistics** : contient des classes pour la collecte et la présentation des données statistiques.

Le mécanisme de paquetage est utilisé en SSJ pour plus de clarté et pour suivre les conventions du langage Java.

5.2 Organisation globale des classes de SSJ

La figure 5.2 présente une organisation simplifiée des classes de SSJ et les liens qui leurs sont associés en UML (diagramme de classe). Les liens sont de type généralisation/spécialisation, implémentation ou agrégation. Une généralisation/ spécialisation permet d'exprimer l'héritage, elle est notée avec un triangle dont la pointe est orientée vers la classe la plus générale. Une implémentation (une interface est implémentée par une ou plusieurs classes) est représentée par une droite pointillée avec une flèche pointant sur l'interface à implémenter. L'agrégation est un type d'association non symétrique, elle est représentée par un petit losange du coté de l'agrégat.



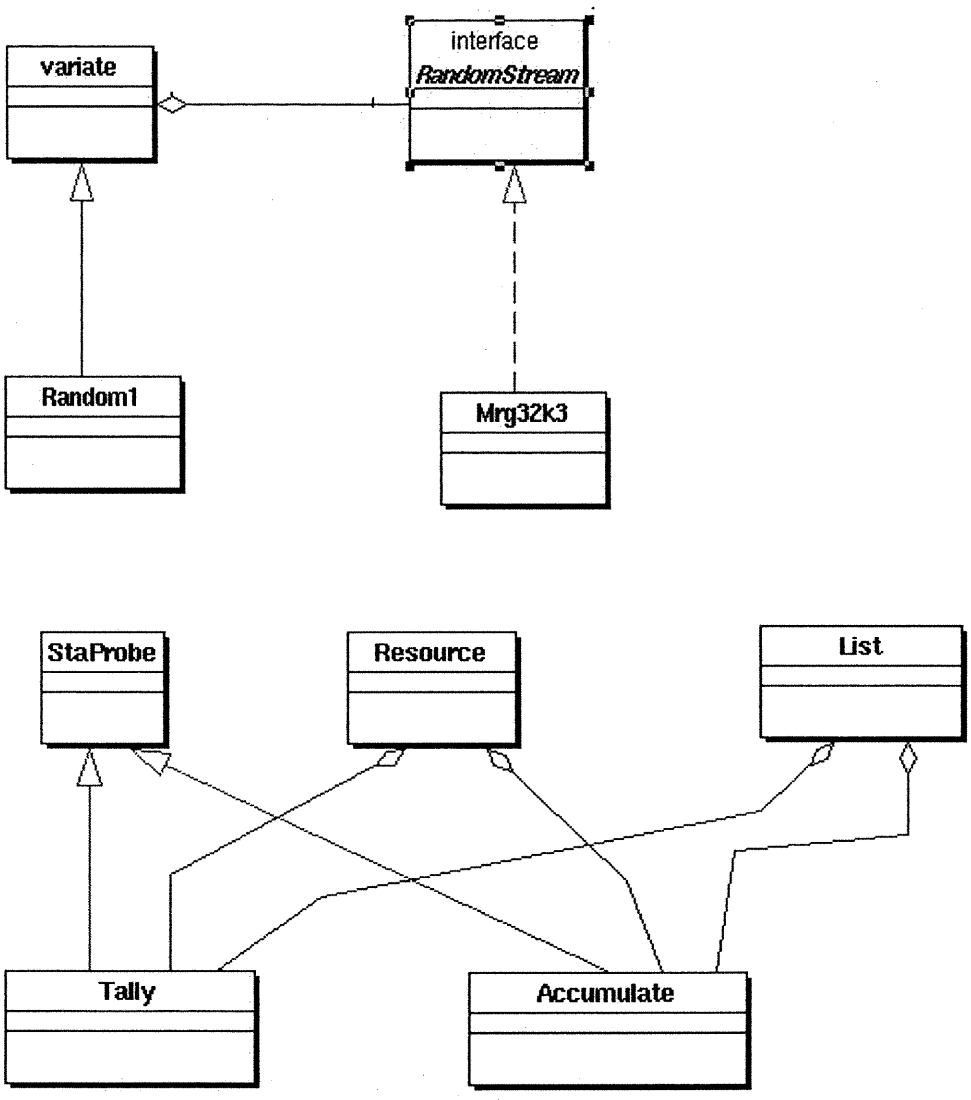


Figure 5.2 : Organisation globale simplifiée de SSJ

5.3 Performances de SSJ

SSJ se base sur le langage Java, qui, malgré ses innombrables avantages, est considéré comme un langage très lent et peu performant. Ceci est dû, principalement, à son mécanisme de machine virtuelle (qui exécute un code semi interprété), à son système de sécurité, son «Garbage Collector» et son système «multi-thread». Néanmoins, nous le répétons encore une fois, cette faiblesse peut être surmontée. Ceci dépend de la façon de

programmer et aussi du choix de la machine virtuelle Java (ou compilateur). Ceci dit, les performances de SSJ ont été influencées considérablement par les deux facteurs suivants :

1. Effet de quelques mesures de programmation : Suite à une intervention de la part de J. Vaucher, nous sommes arrivés à des résultats très satisfaisants en terme de performances. Même si chaque instruction en SSJ est bien étudiée et bien réfléchie pour donner les meilleures performances possibles, l'implantation de quelques mesures ont eu un impact important pour les performances. Parmi ces mesures :
   - Recycler les processus et, surtout, les threads associés aux processus. Ceci est très important, étant donné que la création et la destruction des threads sont, toutes les deux, très coûteuses (voir sect. 5.4).
   - Implanter le processus principal d'une façon efficace en confondant le processus principal avec le «main thead». Cela a un effet non négligeable pour les performances, étant donne que le processus principal exécute les événements, gère et coordonne le cheminement des autres processus (voir sect. 5.4.6).
   - Faire passer le contrôle d'un processus à un autre sans ranimer le processus principal. Cela permet d'éviter de réactiver et désactiver le processus principal à chaque fois qu'un processus termine ou suspend ses activités. Notons que l'activation et la désactivation d'un processus passe par un appel à des méthodes synchronisées qui sont des opérations coûteuses en terme de performance (voir sect. 5.4.4).
   - Recycler les événements associés aux processus, ceci est réalisé en attachant un événement à chaque thread associé à un processus. Le thread est réutilisé avec son événement (voir sect. 5.4.1)
   - Recycler les objets de la liste d'événements (les avis d'événements). Cela permet de réduire considérablement la création et la destruction des objets de la liste.
   - Recycler les éléments (noeuds) de la classe **List**. Aussi, SSJ utilise plusieurs objet de ce type et la réutilisation des objets insérés réduira le nombre d'objets crée et détruits.

- Séparer la vision par événement et celle par processus. Cela permet d'exécuter les exemples utilisant la vision par événements plus rapidement. En effet, nous avons implémenté l'exécutif de la simulation d'une façon permettant d'exécuter les exemples utilisant la vision par événements d'une manière complètement séparée vis-à-vis de celle de processus.

Mesure implantée	**Avant**	**Après**	**%**
Recyclages des threads	139.76	71.50	95 %
Passer le contrôle directement d'un processus à autre	71.50	39.02	83 %
Confondre l'exécutif avec le «main thread »	39.02	33.83	16 %
Recycler les avis d'événements	33.83	29.98	13 %
Recycler les nœuds des objets de type List	29.98	27.56	8 %
Réutilisation des événements associés aux processus	27.56	25.74	7 %
Autres mesures	25.74	23.13	11 %

Tableau 5.1 : Effet de l'implémentation de quelques mesures

Pour implémenter efficacement ces mesures, des ajustements multiples et répétitifs ont été nécessaires pour aboutir à des résultats satisfaisants en terme de performances. Le tableau 5.1 montre l'effet de l'implémentation de ces mesures qui ont donné des temps d'exécution jusqu'à 6 fois plus rapides pour l'exemple de la file d'attente **QueueProc.** La première colonne du tableau indique la mesure implantée pour améliorer les performances. Les deuxième et troisième colonnes montrent l'effet de chaque mesure en indiquant le temps pris par l'exemple, respectivement,

avant et après l'implantation de cette mesure. La dernière colonne montre l'effet de cette mesure en pourcentage de réduction du temps de CPU. La dernière ligne montre l'effet des autres mesures qui ne sont pas très importantes mais, rassemblés, donnent quand même un gain dépassant 10 %. Parmi ces mesures, on cite la façon de mémoriser et de retrouver l'état d'un processus (voir 5.4.2), la réutilisation d'objets de toutes sortes, l'implémentation des listes de threads (libres et non) d'une façon efficace, etc.

2. Effet du choix de la Machine Virtuelle Java (JVM): depuis l'apparition de Java, plusieurs versions de JVM ont vu le jour et beaucoup d'améliorations ont été réalisées surtout pour améliorer les performances. Ainsi, il y a eu l'apparition, entre autres, des compilateurs JIT (« Just In Time compiler») (Yellin [1996]) et les machines virtuelles dites « HotSpot » versus les anciennes JVM dites «Classic VM» (Sun [2001]). Un compilateur JIT, qui peut être intégré dans une «Classic VM», consiste à faire une compilation dynamique du « bytecode » Java. Cette compilation consiste à traduire dynamiquement le « bytecode » en code machine de la plate-forme en question. Chaque méthode est compilée lors du premier appel et le résultat de cette compilation sera utilisé pour les appels ultérieurs de cette même méthode. Un compilateur «HotSpot», qui est une amélioration des compilateurs JIT, se base sur le principe d'une compilation adaptative. Cette compilation consiste à identifier dynamiquement les portions du code fréquemment exécutées (dites «hot») et les compiler pour la plate-forme en question.

   Les JVM classiques, dites «Classic VM», supportent les threads simulés appelés «green thread», qui sont gérés au niveau de la machine virtuelle, et les threads natifs du système d'exploitation. Le choix de la catégorie de thread à utiliser est laissé à l'utilisateur final du programme (option -green et -native de l'interpréteur java). Il s'avère que les machines virtuelles HotSpot ne supportent que la deuxième catégorie de threads. L'implémentation JDK-1.3 de Sun supporte l'option -classic qui permet d'utiliser une JVM classique avec des «green Thread» (les options -client et -server utilisent les threads natifs avec un compilateur HotSpot).

Cependant, l'implémentation JDK-1.4 de Sun ne supporte pas l'option -classic et l'utilisateur n'a aucune façon d'exécuter un programme avec des threads simulés, qui sont très utiles pour la simulation.

Durant le développement de SSJ, nous avons installé et utilisé quelques compilateurs de type JIT et HotSpot pour les plates-formes Solaris, Linux et Windows 2000. Notamment, nous avons utilisé, pour Linux, un compilateur JIT de Borland appelé « javacomp ». Ce compilateur, que nous trouvons très fiable et très rapide, est disponible gratuitement sur le site de Borland (Borland [1998]). Pour Solaris, nous avons utilisé un compilateur de Sun Microsystèmes appelé «sunwjit». Ce compilateur gratuit, est disponible sur le site de Sun (Sun [1998]). Aussi, pour l'amélioration des performances de SSJ, nous avons installé et utilisé quelques compilateurs statiques qui compilent entièrement le code Java en code machine en générant un code exécutable. Ce type de compilateur, indépendamment des performances, permet un lancement très rapide des applications Java, qui prennent, avec les autres types de compilateurs, un certain temps non négligeable pour se lancer. Nous avons utilisé, notamment, un compilateur « freeware » appelé GCJ de la fondation GNU (Gnu [2001]) et un compilateur commercial appelé TowerJ 3.8.1 (TowerJ [2001]), pour lequel nous avons eu une licence d'utilisation pour seulement une seule journée. Ce compilateur est, selon ses concepteurs, très performant surtout pour les applications coté serveur et compatible 100% avec JDK de Sun.

Le tableau 5.2 donne les temps de CPU (en seconde) pris par les exemples **QueueEv** et **QueueProc** (voir chap. 3) sur 10^5 unités de temps, ces deux exemples utilisent 10.0 comme inter-arrivée et 9.0 comme temps de service à la place de 1.0 et 0.8, respectivement. Ces modifications étaient nécessaires pour pouvoir faire une comparaison avec le compilateur TowerJ auquel nous avons eu une licence d'utilisation très limitée. Les exemples **BankEv** et **BankProc** font références aux programmes de simulation en SSJ par événements et par processus, respectivement, d'un modèle de banque tiré de Bratley et al. [1987]. Les programmes pour ces exemples sont donnés dans le guide de l'utilisateur de SSJ (voir L'Ecuyer 2001a).

Le nombre de répétitions est donné entre parenthèse dans le tableau 5.2 (100 ou 1000 répétitions). Les compilateurs qui utilisent un JIT sont indiqués par un symbole «√» dans la deuxième ligne du tableau. Le même symbole dans la troisième ligne indique l'utilisation des threads natifs, sinon les « green threads » sont utilisés. Dans le cas de GCJ (symbole «?»), nous n'avons trouvé aucune indication au sujet de type de thread utilisé. Notons que l'exemple de **BankProc** avec 1000 répétitions, qui manipule beaucoup de threads simultanément, ne donne pas de résultats avec les compilateurs utilisant des threads natifs parce qu'il demande plus de threads que ce que l'on peut créer.

Exemples	Sun JDK1.2.2_006 Sans JIT	Sun JDK1.2.2_006 Avec javacomp	Sun JDK 1.3.1 Classic VM	Sun JDK1.3.1 client (HotSopt)	GCJ	TowerJ
Compilateur JIT		√		√		
Threads natifs				√	?	
QueueEv (10^5)	0.31	0.10	0.33	0.07	0.21	0.069
BankEv(100)	0.72	0.17	0.76	0.12	0.34	-
BankEv (1000)	7.27	1.73	7.47	1.01	2.02	1.45
QueueProc (10^5)	0.76	0.33	0.82	0.97	0.69	0.24
BankProc(100)	1.64	0.73	1.81	1.82	1.52	-
BankProc(1000)	19.95	7.37	18.25	-	-	6.78

Tableau 5.2 : Choix de la JVM : tests de performance

En examinant Le tableau 5.2, nous pouvons tirer les conclusions suivantes :

- L'utilisation des compilateurs JIT (spécialement javacomp) permet d'exécuter les exemples de SSJ qui utilisent la vision par processus

(**QueueProc** et **BankProc**) de deux à trois fois plus rapidement, et ceux utilisant la vision par événements (**QueueEv** et **BankEv**) de trois a cinq fois plus rapidement.

- L'utilisation d'un compilateur HotSpot a permet d'exécuter les exemples utilisant la vision par événements jusqu'à 10 fois plus rapidement qu'avec une JVM classique sans JIT (voir tableau 5.2). Cependant, les exemples utilisant la vision par processus ne s'exécutent pas toujours d'une façon normale avec ce type de compilateur. Lorsqu'ils s'exécutent correctement, ils donneront les mêmes résultats (parfois inférieures), en terme de performance, qu'une JVM classique sans JIT. Ceci est dû à l'utilisation des threads natifs au lieu des threads simulés.
- Les threads de Java ne sont pas adaptés pour faire de la simulation par processus. Même si l'utilisation des threads simulés donne des résultats corrects et relativement satisfaisant en terme de performances, l'utilisation des threads natifs pour la simulation est lente et donne des résultats erronés dans plusieurs cas. En effet, un programme de simulation utilisant les threads natifs peut facilement demander plus de threads qu'il y en dans le système, et par conséquent ne peut pas donner de résultats. En plus, un programme de simulation, même s'il ne nécessite pas beaucoup de processus, ne s'exécute pas toujours d'une façon correcte sur une machine multi-processeurs. Les choses s'empirent avec les développements récents de Java où les threads simulés ne sont plus supportés à partir de JDK1.3.0 sur Solaris et à partir de JDK-1.4.0 sur les autres machines. Java avait introduit son mécanisme de thread pour supporter des applications multi-tâches qui ont un vrai parallélisme (par exemple : plusieurs demandes de connexions à une base de données). Ceci a beaucoup d'avantages où l'utilisation des threads natifs permet de répartir les tâches indépendantes sur des processeurs différents (pour un souci d'efficacité) dans une machine multi-processeurs. SSJ est conçu (comme la plupart des langages et librairies de simulation) pour s'exécuter d'une façon séquentielle sur un seul processeur. L'implémentation des processus concurrents n'est en fait qu'un

pseudo-parallélisme. En effet, à tout moment, seulement un seul processus détient le contrôle et à chaque occurrence d'un événement, on décidera quel processus va prendre le contrôle tout en passivant le processus qui avait le contrôle (si ce n'est pas le même).

- Le compilateur GCJ est plus rapide, avec un lancement instantané, qu'une JVM classique sans JIT, mais moins rapide qu'un compilateur JIT ou HotSpot. L'utilisation du compilateur TowerJ apporte un gain important en terme de performance surtout pour les exemples utilisant la vision par processus. Il surpasse les autres compilateurs que nous avons utilisés excepté les compilateurs HotSpot avec vision par événements. Cependant, javacomp, qui est gratuit, donne des résultats satisfaisants, et il semble le meilleur choix même si TowerJ (un produit commercial) donne des résultats un peu plus rapides.

Pour des fins de comparaison avec le langage C, nous avons exécuté l'exemple de la file d'attente **QueueEv** (voir chapitre 3) en utilisant la librairie SSC. Cette dernière est une librairie similaire à SIMOD supportant seulement la vision par événements et implémentée en ANSI-C (voir L'Ecuyer [2002b]). L'exécution de cet exemple prend 2.8 secondes en utilisant le compilateur gcc avec l'option d'optimisation –O3. Le même exemple en SSJ et sur la même machine s'exécute en 3.6 secondes. Ainsi, sur cet exemple, l'utilisation du langage C avec un niveau d'optimisation très élevé donne un programme qui n'est que 30 % plus rapide que SSJ.

5.4 Implémentation des processus en SSJ

Les processus utilisent des threads Java pour s'exécuter. Notons que l'environnement de la machine virtuelle Java est multi-processus (en anglais « multi-threaded»). Il dispose de la classe **Thread** et de l'interface **Runnable** pour créer des threads. La méthode **start**() permet de démarrer effectivement le thread sur lequel elle est invoquée. La méthode **setDaemon(boolean)** permet de spécifier si un thread doit s'exécuter en arrière plan (tâche de fond) ou non. Un thread qui s'exécute en arrière plan n'existe, en général, que pour rendre des services aux autres threads. Quand il n'y a plus que des

threads qui tournent en tâche de fond dans le système, la Machine Virtuelle Java s'arrête. Les premières versions de Java disposent d'autres méthodes utiles pour la synchronisation de processus, telles que **stop**(), **suspend**() et **resume**(), respectivement, pour stopper, suspendre et réveiller un thread. Malheureusement ces méthodes ont un comportement insécurisé et peuvent causer des problèmes d'inter-blocage. À partir de la version 1.2 de Java, l'utilisation de ces méthodes est fortement déconseillée (en Anglais, «deprecated»), voir Sun [2001] pour plus de détail.

Contrairement à un événement qui s'exécute d'une façon instantanée, un processus a son existence propre; il prend naissance, vit dans un contexte donné en interaction avec d'autres processus, exécute ses instructions, collabore avec d'autres processus, peut se bloquer, puis il meurt éventuellement. Pour réaliser cela, il faut implémenter les mécanismes nécessaires pour la gestion et la synchronisation des processus.

À Chaque processus est associé un thread appelé **myThread2** dans lequel ce processus doit s'exécuter. Ce thread est défini dans une classe appelée **Thread2** accessible seulement à la classe **Process**. Etant donné que la création et la destruction des threads sont deux opérations très coûteuses en terme de performance, nous avons expérimenté plusieurs façons différentes pour implémenter une solution performante basée sur la réutilisation des objets de type **Process** et leurs threads associés. Selon les tests que nous avons effectués avec JDK-1.2.2 sous Linux Red-Hat 6.0, la création d'un **Thread** est presque 80 fois plus lente que la création d'un objet de type **Object**. Dans ce qui suit, nous discutons quelques alternatives d'implémentation de la classe **Process** :

1. Pas de réutilisation : on considère chaque processus comme un thread séparé en faisant hériter la classe **Process** de la class prédéfinie **Thread**. À chaque création d'un processus, un thread est automatiquement créé et à chaque destruction d'un processus, ce thread est automatiquement détruit. Cette solution est la plus évidente, mais elle est désastreuse pour les performances.
2. Réutilisation des threads : cette technique, suggérée par J. Vaucher, est connue dans la littérature sous le nom de « Thread Pool » (voir Holub [2000], Hyde [1999], Modi [2000, 2001], Bulka [2000]). Elle est très

puissante et sauve beaucoup de temps. Elle consiste à créer un réservoir (une liste référencée par **lastFree**) de threads libres (on ne détruit pas les threads créés mais on les réutilise). À chaque création d'un nouveau processus, on lui attribue un thread de type **Thread2**. Ceci est réalisé par un appel à une méthode **static getThread(Process p)** (donnée ci-bas). Cette méthode associe un thread au processus **p** passé en paramètre. Ce thread ne sera créé que s'il n'y a pas de thread libre (**lastFree = null**). Quand un processus termine son cheminement, on récupère son thread associé et on l'ajoute à la pile des threads libres. Les variables **lastFree** et **nextFree** servent à manipuler la pile des threads libres.

```
public static final Thread2 getThread (Process p) {
    // if there is no free thread available, a new thread will be created and started.
    // p is the process associated to this thread.
    if (lastFree == null) return new Thread2 (p);
    // there is at least one free thread available
    Thread2 th = lastFree;
    lastFree = lastFree.nextFree;
    th.myProcess = p;
    return th;
   }
```

Un thread créé ne sera jamais détruit avant la fin de la simulation. Ce thread se bloque quand il n'y a pas de processus associé à lui. Il se bloque aussi dans le cas où le processus associé ne serait pas encore activé pour la première fois. Ceci est utile lors de la création d'un nouveau thread qui sera automatiquement démarré mais il se bloque en attente d'être réveillé par l'activation de son processus. La méthode **run**() de ce thread (appelé **Thread2**) est définie comme suit :

```
public final void run () {
while (true) {
    try {
      passivate ();   // free or the process associated is not activated yet.
      myProcess.actions ();   // myProcess starts its life.
      myProcess.dispatch (); // Give control to another process.
    } catch (Error e) {}     // An Error exception is thrown because
                             // the process is killed.
    myProcess.myThread2 = null;
    myProcess = null;        // the associated process is DEAD.
    NextFree = lastFree;  lastFree = this;
}
```

Cette solution apporte un gain de performance considérable, en effet, l'exemple de la file d'attente **QueueProc** s'exécute jusqu'à trois fois plus vite par rapport à la précédente alternative (voir tableau 5.3). Notons que le gain apporté dépend du nombre de threads réutilisés.

3. Réutilisation des processus avec leurs threads associés : cette idée consiste à réutiliser les objets de type **Process** (qui implémentent l'interface **Runnable**) avec leurs threads associés, au lieu de réutiliser seulement les threads. La méthode **run**() de la classe **Process** ressemble à celle de la classe **Thread2** définie dans le paragraphe précédent. Cette façon consiste à créer une liste pour chaque type de processus. Chaque fois, quand on veut créer une nouvelle instance d'un type de processus donné, on appelle la méthode **create**() (donnée ci-bas) qui cherche, dans la liste des instances libres de même type (référencée par la variable **myList**), s'il y en a une de libre. Sinon, on crée une nouvelle instance identique par la méthode **clone**() (définie ci-bas). L'utilisation de la méthode **create**() est illustrée dans les rubriques suivantes.

```
public Process create () {
      // If there is a free instance in myList, we reuse it.
      // else we create another instance by cloning this object.
      Process p;
      if (myList.empty()) p = (Process)clone ();
      // no free instance is available.
      else p = (Process)myList.remove (List.FIRST);
      // at least one free instance is available
      return p;
}

protected Object clone () {
       // create an identical object of this object
       Process proc = (Process)super.clone ();
      // the associate thread will be created and started
       proc.myThread = new Thread (proc); // create the associated thread
       proc.myThread.setDaemon (true); // set the thread as daemon thread
       proc.myThread.start ();          // start the associated thread
       proc.n = 0;                      // reset the semaphore indicator
       return proc;                     // the process created is returned
     }
```

Chaque objet de type **Process** contient une référence **(myList)** à la liste des instances libres de même type. La liste de type de processus est accessible par la variable **first**. Lorsqu'on fait appel à l'opérateur **new** pour créer une nouvelle instance au lieu de la méthode **create**(), le constructeur de la classe **Process** cherche d'abord s'il y a un nœud crée pour ce type d'instance, sinon un nœud pour ce type sera crée. La variable **next** permet de se déplacer dans la liste des types.

Cette solution est un peu plus rapide par rapport à la précédente pour quelques types d'exemples (qui créent beaucoup d'objets de type **Process**). En effet, il y a beaucoup moins d'objets créés et détruits de type **Process**, ceci signifie moins de «garbage collection». Toutefois, cette solution n'est pas toujours rapide et elle avantage quelques types d'exemples par rapport aux autres. Elle ne serait pas efficace dans le cas suivant : prenons un exemple contenant deux types de processus. Supposons qu'à un moment donné, il y a beaucoup d'instance libre de premier type et on vaudrait créer des instances de deuxième type. Cette solution ne permet pas la réutilisation des threads associés aux instances libres du premier type, alors que la précédente solution le permet.

4. Réutilisation des processus et des threads séparément : cette façon, qui est une combinaison des deux précédentes solutions, donne de résultats rapides pour tout genre d'exemple. Elle consiste à réutiliser les objets de type **Process** et les threads associés séparément pour remédier aux inconvénients constatés dans la dernière solution. Ainsi, la méthode **create**() devient :

```
public Process create () {
   Process p;
   if (myList.empty()) p = (Process)clone ();
   // no free instance is available.
   else  p = (Process)myList.remove (List.FIRST);
   // a free instance is available
   p.myThread = Thread2.getThread (p) ;
   // look for a free thread if there is any, if not a new thread will be created and started.
   return p;
   }
```

La méthode **clone**() est très simple et consiste à appeler tout simplement la méthode **clone**() de la classe **Object** dont la classe **Process** hérite.

```
protected Object clone () {
     try {
          return super.clone ();  //create an object
     }catch (CloneNotSupportedException e) {}
     return null;
}
```

Le constructeur de la classe **Process** (voir ci-dessous) manipule la liste de types de processus. À chaque appel au constructeur (voir ci-bas) de cette classe pour créer une nouvelle instance au lieu de la méthode **create**() (les deux façons sont possibles), le constructeur vérifie si un nœud est déjà créé pour ce type. Sinon un nœud sera crée pour ce type.

```
public Process () {
    // create a node for a free instance of this type of process,
    // if this node was created before, only the myList variable will be set.
    // the create method is more appropriate for creating another instances
    // of the same type of process.
    if (first == null) {myList = new List();  first = this; }
    else {
         Process ptr=first;
         while (ptr.next!=null && ptr.getClass()!=getClass())
               ptr = ptr.next;
         if ( ptr.getClass () != getClass () ) {
              myList = new List();
              ptr.next = this;
         }
         else myList = ptr.myList; // List of free instances of this type
    }
    myThread = Thread2.getThread (this);
}
```

Notons que la méthode **create(),** qui est une méthode de la classe **Process**, ne peut être invoquée que sur un objet déjà crée ou bien à l'intérieur d'une classe qui hérite de la classe **Process** (typiquement à l'intérieur de la méthode **actions**()). Pour illustrer l'utilisation de **create**(), on reprend l'exemple de la file d'attente donné à la sect. 3.1.2 (figure 3.3). Les seuls changements à faire figurent dans la méthode **actions**() de la classe **Customer** (voir ci-bas).

```
class Customer extends Process {
        public void actions () {
            create(genArr.expon (meanArr));
            server.request (1);
            delay (genServ.expon (meanServ));
            server.release (1);
        }
}
```

L'utilisateur peut aussi utiliser l'opérateur Java **new** à la place de **create**(), sauf que cela va ralentir beaucoup le système étant donné que le constructeur de la classe **Process** (dans ce cas) entretient une liste par type de processus.

Le tableau 5.3 montre une comparaison de performances pour les quatre alternatives d'implémentations de la classe **Process**. La troisième alternative et la quatrième donnent les mêmes résultats parce qu'il y a un seul type de processus.

Alternative d'implémentation de la classe Process	**Temps en secondes**
1. pas de réutilisation	59.20
2. réutilisation uniquement des threads	23.13
3. réutilisation des thread et des processus ensemble	21.96
4. réutilisation des thread et des processus séparément	21.96

Tableau 5.3 : La classe **Process** : comparaison de performance

Même si la dernière solution apporte un gain en terme de performance (pour des exemples qui créent beaucoup d'objets de type **Process**), SSJ adopte la deuxième solution parce qu'elle est simple et la création des objets se fait toujours d'une façon uniforme (l'opérateur **new**). En plus, le gain en performance de la dernière méthode n'est pas toujours significatif (sauf dans le cas où il y aurait un nombre énorme d'objets crées).

5.4.1 Les avis d'événements associés aux processus

À chaque processus correspond un événement qui sera placé dans la liste d'événements pour réveiller ce processus. Plusieurs façons étaient à envisager : associer un événement à chaque processus, lier un événement à chaque thread associé au processus, faire hériter la classe **Process** de la classe **Event**, faire hériter la classe **Thread2** de la classe **Event** ou bien créer une interface appelée **EventNotice** qui sera implémentée par la classe **Event** et la classe **Process**. Dans ce dernier cas les objets insérés dans la liste d'événements seraient de type **EventNotice**. Dans les paragraphes suivants, nous discutons seulement deux alternatives qui sont, à notre avis, simples et performantes :

- La première façon consiste à associer à chaque thread (et non pas chaque processus) un événement de type particulier appelé **pevent** (donné ci-bas). Cet événement est défini dans une classe anonyme et interne dans la classe **Thread2**:

```
Event pevent = new Event () {
      public void actions () {
            eventTime = EXECUTING;
            // the state of the associated process is memorized in the eventTime variable.
            current = this.Process;      // update the current process .
            wakeup ();   // wake up this thread..
      }
}
```

- La deuxième façon, adoptée dans SSJ, consiste à faire hériter la classe **Thread2** de la classe **Event** (chaque thread est un événement) et redéfinir la méthode abstraite **actions**() (donnée ci-bas) de la classe **Event.** Cette méthode, qui sera exécutée seulement une seule fois au maximum au cours d'une simulation, transférera le contrôle de l'exécutif à un processus. Une fois le contrôle passé à un processus, il sera passé d'un processus à l'autre via la méthode **dispatch**() (voir sect. 5.4.4). Un processus qui a le contrôle exécutera les événements imminents, s'il y en a, avant de donner le contrôle à autre processus. L'exécutif ne rependra le contrôle qu'à la fin de la simulation.

```
public void actions () {
      // This method will be executed only once.
      // It transfers the control from the executive to this thread.
      // The control will then be passed from process to process,
      // which will execute the events if any.
      // Control will be returned to the executive only at the end of simulation.
      Process.current = myProcess;
      activate (); // Notify this thread to be ready to take control
                   // as soon as the caller's thread is passivated.
      Sim.passivate (); // Passivate the main thread (executive).
}
```

Un objet de type **Thread2**, lorsqu'il est crée, est toujours dans l'un des deux états suivant : **WAITING** ou **RUNNING**, suivant qu'il a un processus associé à lui ou non. Il est dans l'état **RUNNING** s'il y a un processus associé à lui et que ce dernier avait déjà commencé son cheminement sinon il est dans l'état **WAITING**. La méthode **run**() (voir ci-dessous) de ce thread consiste à donner la main à la méthode **actions**() du processus associé. Une fois que ce processus termine son cheminement, la méthode **dispatch**(), définie à la sect.5.4.4, est exécutée pour donner le contrôle à un autre thread. Avant que l'autre thread ne prenne le contrôle, cette méthode ajoute le thread en cours d'exécution à la pile des threads libres pour une utilisation ultérieure.

5.4.2 Les différents états d'un processus

Un processus est toujours dans l'une des états suivants : **INITIAL, SUSPENDED, EXECUTING, DELAYED** ou **DEAD.** À tout moment, un et un seul processus se trouve dans l'état **EXECUTING,** c'est celui qui est en train d'exécuter ses instructions; on dit qu'il a le contrôle. Lorsque ce n'est pas un processus de l'usager qui a le contrôle, c'est l'exécutif de la simulation qui l'a. L'état d'un processus peut être mémorisé de deux façons différentes :

- On attribue à chaque processus un indicateur d'état ("**state**") donnant, à tout moment, l'état de ce processus. Cet indicateur prend toujours l'une des valeurs suivantes : **INITIAL, SUSPENDED, EXECUTING, DELAYED** ou **DEAD**. Cette façon est adoptée dans SIMOD. Notons qu'en SIMOD, un processus peut être seulement dans l'une des trois états (**SUSPENDED, EXECUTING, DELAYED**) au lieu de cinq états différents.

- On se sert du thread, qui hérite de la classe **Event**, associé au processus et sa variable **eventTime** pour obtenir d'une façon implicite l'état de ce processus. Un processus qui n'a pas de thread associé est considéré dans l'état **DEAD**, le processus courant (référencé par la variable **static current**) est considéré dans l'état **EXECUTING**. Quand la variable **eventTime** du thread associé est supérieure ou égale à 0, le processus est dans l'état **DELAYED**. Un processus dont le thread associé est dans l'état **WAITING** (constante définie dans la classe **Thread2**) est considéré dans l'état **INITIAL**. La méthode **getState**() renvoie l'état du processus (voir ci-bas).

```
public int getState ()   {
     //The state of a process is not kept explicitly, but obtained from
     // the state of the associated thread (myThread) and its eventTime variable.
     // When there is no associated thread (myThread==null), the process is
     // considered to be in the DEAD state. When the variable current == this, the
     // process is in EXECUTING state. Otherwise, a process is considered to be:
     // - DELAYED  if eventTime of the myThread >= 0.0,
     // - In INITIAL state if eventTime of myThread is set to Thread2.STARTING (-10.0)
     // Otherwise, the processus is in SUSPENDED state.
     if (myThread == null)       return DEAD;
     elseif (current == this)    return EXECUTING;
     elseif (myThread.eventTime >= 0.0)   return DELAYED;
     elseif (myThread.eventTime == Thread2.INITIAL)
                                 return INITIAL;
     else return SUSPENDED;
}
```

Cette dernière façon, adoptée dans SSJ, est très simple et un peu plus rapide parce qu'on a une variable en moins à manipuler et à entretenir.

La figure 5.3 montre les différents états d'un processus et la transition d'un état à un autre à l'invocation d'une méthode. Lorsqu'un nouveau processus est créé (appel de **new** ou **create**()), il est dans l'état **INITIAL**, il passera à l'état **DELAYED** à l'appel de la méthode **schedule()** ou **scheduleNext();** un processus est dans l'état **DELAYED** si un événement est prévu pour lui donner le contrôle. L'appel de la méthode **reschedule**() sur un processus en état **DELAYED** modifie seulement sa date d'activation sans changer son état (qui reste **DELAYED**). Il passera à l'état **EXECUTING** si la date d'occurrence de cet événement est atteinte (lorsqu'il débute ou reprend son exécution).

La méthode **suspend**() fait passer un processus de l'état **EXECUTING** à l'état **SUSPENDED** ; il est dans l'état **SUSPENDED** s'il est en attente qu'on lui redonne le contrôle sans qu'il y ait d'événement de prévu à cet effet. Un processus dans l'état **DELAYED** passe à l'état **SUSPENDED** à l'appel de la méthode **cancel**(). La méthode **kill**() tue le processus en le mettant dans l'état **DEAD**.

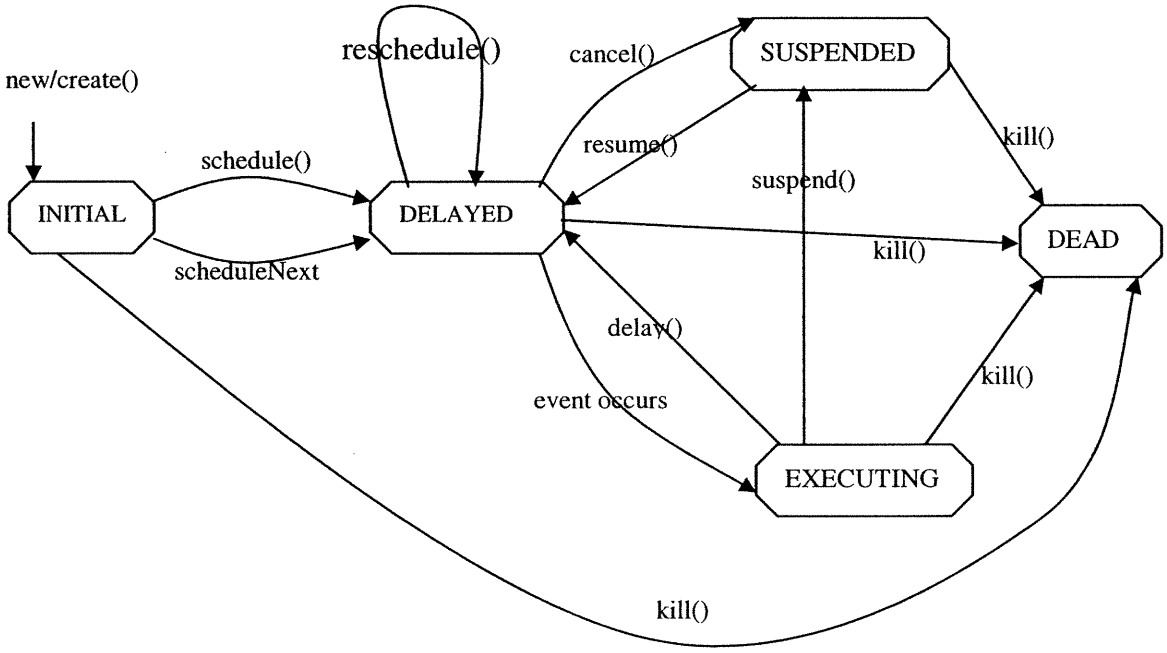


Figure 5.3 : Les différents états d'un processus

5.4.3 Le processus courant

Plusieurs façons étaient à envisager pour obtenir le processus qui détient le contrôle en ce moment. La façon la plus simple et la plus performante consiste à mettre à jour la variable **current** (variable statique de la classe **Process**) juste avant d'activer le processus qui va prendre le contrôle. La méthode **currentProcess**() ne fait que retourner la valeur de cette variable. Notons que la méthode prédéfinie **currentThread**() n'est pas appropriée dans notre cas, car la class **Process** n'hérite pas de la classe **Thread**. Il y a une autre façon, suggéré par J. Vaucher, qui consiste à garder toujours l'élément se trouvant en tête de la liste d'événements. Cet élément référence le processus courant. Dans ce dernier cas, on devrait traiter le cas où la tête de liste référence un événement au lieu d'un processus.

5.4.4 Synchronisation des processus

Pour la synchronisation des processus, on a besoin de suspendre, redémarrer et arrêter complètement un processus. Comme l'utilisation des méthodes **suspend()**, **resume()** et **stop()** de la class **Thread** a été déconseillée, nous avons été amenés à implémenter les mécanismes nécessaires de synchronisation des processus (suspendre, réveiller et stopper un processus) et remplacer les méthodes déconseillées. Notons que la synchronisation de processus en SSJ se fait d'une façon transparente vis-à-vis de l'utilisateur. Elle est effectuée derrière la scène par un moniteur qui s'occupe de passer le contrôle d'un processus à autre, de suspendre ou de réveiller un processus. Par souci de performance, et sur une idée proposée par J. Vaucher, un processus qui détient le contrôle ne réveille pas l'exécutif, mais, passe directement le contrôle à un autre processus après avoir exécuté les événements, s'il y en a. Ceci est réalisé par un appel à la méthode **dispatch**() définie ci-après. L'exécutif ne reprendra le contrôle qu'à la fin de la simulation (l'effet de l'implémentation de cette mesure est montré dans le tableau 5.1).

```
public static void dispatch () {
     Event ev;
     while ((ev = Sim.eventList.removeFirst()) != null &&
                                        Sim.stopped == false) {
          Sim.currentTime = ev.eventTime; // upadate the current time
          ev.eventTime = -10;
          if (ev.process) { // this is a process, the control will transferred to it.
               Current = ((Thread2)ev).p;      // the current process is updated
               current.myThread.activate (); // activate the associated thread
               return;
          }
          else ev.actions (); // this is an event which is executed by the caller process.
     }
     Sim.activate ();   // Simulation is over, the executive will be reactivated
}
```

La méthode **dispatch**(), appelée par le processus courant, exécute les événements s'il y en a. Une fois qu'un événement de type **Thread2** est rencontré (la variable **process == true**) le contrôle est passé à ce thread après avoir mis à jour la variable **current**.

5.4.4.1 Suspendre et réveiller un processus

Plusieurs solutions étaient à envisager pour implémenter les outils permettant de suspendre et d'activer un processus (**suspend**() et **resume**()) d'une part; et d'autre part, pour tuer un processus ou éventuellement tous les processus (**kill**() et **killAll**()). Dans les paragraphes suivants, nous discutons ces solutions tout en justifiant nos choix.

- La première solution, adoptée dans SSJ, consiste à associer à chaque objet de type **Thread2** une variable entière appelée **s**. Une fois que la valeur de **s** devient négative le thread se bloque. Cette variable est manipulée par deux primitives **activate**() et **passivate**() (voir ci-dessous). La première méthode incrémente la valeur de **s** et envoie un signal de notification à ce thread pour le réveiller. La deuxième méthode décrémente la valeur de **s** et bloque ce thread si jamais **s** prend une valeur négative. Notons que ces deux méthodes sont déclarées **synchronized** pour la simple raison que le langage Java impose que l'appel aux méthodes **notify**() ou **wait**() soit synchronisé.

```
public synchronized final void passivate () {
     try {
            s--;
            if (s<0)  wait ();
     }
     catch (InterruptedException e) {
     s++;  throw error; // errorException;
     // This thread is waken up and interrupted because
     // the kill() method was called on its associated process.
     // Throws an Error Exception that
     // will be catched by run() method.
     }
 }

public synchronized final void activate () {
      // Notify this thread to be ready to take control.
      // This thread doesn't take the control till the caller thread
      // calls the passivate() method.
     s++;
     notify ();
}
```

Cette façon de faire est une implémentation du concept de sémaphore, qui était l'un des premiers mécanismes proposés pour résoudre le problème de synchronisation entre processus (voir Burn et Davies [1993], Banâtre [1991]). Le concept de sémaphore, introduit par Dijkstra [1968], est implémenté dans les noyaux de la

plupart des systèmes d'exploitation. L'implémentation de ce concept en Java est expliquée dans beaucoup de livres, le lecteur intéressé est référé à Lea [2000], Holub [2000], Magee et Kramer [2000] et Chiao [2001]

La variable **s** peut prendre trois valeurs différentes : -1 veut dire que le thread est en état bloqué, une valeur 0 ou 1 veut dire que ce thread est en exécution. Cette variable prendra la valeur 1 lorsqu'un processus en exécution veut donner le contrôle à un autre processus en appelant la méthode **dispatch**(), et que cette dernière méthode trouve en tête de la liste d'événements un événement qui va redonner le contrôle à ce même processus (appelant). Ceci peut arriver quand un processus appelle la méthode **delay**() qui va placer un avis d'événement à la tête de la liste d'événements. Notons que l'utilisation de la variable **s** n'est pas obligatoire. On pourrait arriver au même résultat avec un appel à **wait**() (voir méthode **passivate**() ci-dessus) seulement dans le cas où le processus qui va prendre le contrôle est différent du processus courant.

- La deuxième solution pour suspendre et réactiver un processus, consiste à créer un objet central appelé **token** (jeton) dont l'accès est verrouillé par des blocs de code à exclusion mutuelle. Ces blocs sont protégés par le mot clé Java **synchronized** qui permet d'éviter que plusieurs processus aient accès en même temps à ce même jeton. Le fonctionnement de **synchronized** est comme suit : quand l'un des blocs **synchronized** est invoqué sur le jeton, le système pose un verrou sur le jeton puis le bloc est exécuté normalement. Quand le bloc est terminé, le verrou est retiré sur le jeton. Si le jeton est déjà verrouillé par un autre processus, le système met le processus courant l'état bloqué.

  La communication entre les processus se fait par les méthodes **wait()** et **notifyAll()** de la classe **Object**. Ces méthodes permettent de mettre en attente un **Thread** sur un objet (dans notre cas un jeton), et de prévenir les processus en attente sur ce jeton que celui-ci est libre. Quand **wait()** est invoquée sur l'objet jeton, le processus courant perd le contrôle, se met en attente et le verrou sur le jeton est retiré. Le

processus courant est ajouté à la liste des processus en attente du jeton. Seulement le processus dont l'indicateur d'état est **EXECUTING** peut acquérir le jeton; tous les autres processus doivent rester bloqués. Lorsque l'exécutif de la simulation est actif, il met l'indicateur du processus à réveiller dans l'état **EXECUTING** et son indicateur à lui dans l'état **SUSPENDED ;** ensuite, il fait une notification aux autres processus en attente du jeton, avant d'aller dormir. Ceci est réalisé par un appel à la méthode **notifyAll()** suivi de **wait()**. Tous les processus se réveillent mais seulement le processus dont l'indicateur d'état est positionné à **EXECUTING** va acquérir le jeton pour pouvoir poursuivre son cheminement. Tous les autres processus vont se bloquer de nouveau.

```
private void wakeUp () {
   try {
        synchronized (token) { // section critique : un seul processus à la fois
            token.notifyAll ();//envoie un signal à tous les processus en attente du jeton
            while (executive.state != EXECUTING) token.wait ();
            //l'exécutif doit rester bloqué tant que son état n'est pas en EXECUTING.
         }
     } catch ( InterruptedException e ) {}
}
```

Lorsqu'un processus veut libérer le jeton (pour suspendre son exécution ou lorsqu'il se termine), il change l'état de son indicateur et met l'état de l'indicateur de l'exécutif de la simulation à **EXECUTING**; ensuite, il invoque la méthode **notifyAll()** pour réveiller tous les processus en attente du jeton. Seulement l'exécutif va pouvoir acquérir le jeton parce que son indicateur est dans l'état **EXECUTING**. Tous les autres processus vont dormir.

```
state = DELAYED
// il change son état : state=DELAYED à l'appel de delay() sinon state=SUSPENDED à l'appel de suspend()
executive.state = EXECUTING; // il met l'état de l'exécutif en exécution
synchronized (token) {       // section critique : un seul processus à la fois
     token.notifyAll ();     // envoi un signal à tous les processus qui attendent le jeton
     while (state != EXECUTING)
              token.wait (); // il se bloque tant que son état n'est pas en EXECUTING.
}
```

Pour envoyer un signal de notification à un processus, nous avons utilisé **notifyAll()** qui réveille non seulement un processus, mais tous les processus. Ça prend un peu plus de temps que la méthode **notify(),** de fonctionnement un peu similaire, qui réveille un seul processus parmi ceux en attente du jeton. Cette dernière méthode n'est pas appropriée dans ce cas, parce qu'elle notifie arbitrairement un processus quelconque, choisi au hasard, parmi ceux en attente du jeton. Cependant, on aurait pu appeler **notify()** successivement autant de fois qu'il y ait de processus en attente, mais ça n'apporte aucun gain.

Cette solution utilise des mécanismes coûteux en terme de temps d'exécution (surtout l'utilisation de **synchronized** et **notifyAll**()). Notons que les performances sont dépendantes de nombres de processus dans le système. Lorsque le nombre de processus augmente, les performances diminuent. Ceci est dû à l'utilisation de l'instruction **notifyAll**() qui réveille, inutilement, tous les processus au lieu d'un seul processus.

5.4.4.2 Implémentation des méthodes kill() et killAll()

Un processus est tué lorsqu'il termine l'exécution de sa méthode **actions()** ou lorsqu'on invoque la méthode **kill()** sur lui. La méthode **killAll()** tue tous les processus. Étant donné que l'utilisation de la méthode **stop()** de la classe **Thread** est déconseillée, nous avons été amenés à réfléchir sur les mécanismes nécessaires à l'implémentation de ces deux méthodes.

Deux solutions ont été envisagées :

1. La première solution consiste à mettre tous les processus comme des démons qui s'exécutent en tâche de fond (arrière plan), par appel à la méthode **setDaemon(true)** au moment de la création de chaque thread (associé à un processus). La machine virtuelle Java s'arrête automatiquement lorsqu'il ne reste plus que des processus démons, ce qui entraîne l'arrêt de tous les processus encore actifs. Cette solution, même si elle permet d'arrêter un programme de simulation correctement, ne permet

pas de stopper un processus particulier sans arrêter la simulation. Par conséquent, elle ne peut pas servir à implémenter les méthodes **kill()** et **killAll()**.

2. La deuxième méthode consiste à tirer profit du système d'exception mis en œuvre par la machine virtuelle Java et d'utiliser la méthode **interrupt()** de la classe **Thread**. La méthode **interrupt()** a été implémentée en Java à partir de la version 1.1. Cette méthode, lorsqu'elle est invoquée sur un processus, n'interrompt pas ce processus. Mais, elle déclenche seulement une exception de type **InterruptedException**. De ce fait, les processus bloqués vont continuer leurs cheminements même si l'on invoque la méthode **interrupt()** sur eux.

   Notre idée consiste de soulever une exception à tous les endroits où les processus sont susceptibles d'être bloqués, à savoir dans la méthode **delay()** et **suspend()**. Ces deux dernières méthodes soulèvent une exception de type **InterruptedException**. L'utilisateur doit intercepter ou soulever cette même exception dans toutes les méthodes qui font appel à l'une de ces deux méthodes (voir les exemples dans le chapitre 3). Mentionnons que le compilateur Java exige que cette exception soit interceptée ou levée sinon la compilation n'aura pas lieu.

   La figure 5.4 montre un scénario typique de fonctionnement de cette technique. Supposons qu'un processus p1 est bloqué dans la méthode **delay()** (l'exécutif a invoqué **p1.actions**() qui à son tour invoque **p1.delay()**); et qu'un autre processus p2 appelle la méthode **kill()** sur le processus p1, la méthode **kill**() appelle la méthode **interrupt()** qui déclenche une exception de type **InterruptedException**. Cette exception est transmise à la méthode **delay**() dans laquelle le processus est bloqué. Cette dernière méthode interrompt l'exécution de ses instructions en soulevant la même exception. L'exception sera transmise à la méthode définie par l'utilisateur et dans laquelle le processus est bloqué (la méthode **actions**() dans notre cas). A la réception de cette exception, la méthode de l'utilisateur met immédiatement fin à son exécution en passant l'exception au « framework » SSJ qui va l'attraper en récupérant le thread associé et l'objet **Process** (voir la méthode

run() de la classe **Thread2**). Ainsi, le processus en question est tué parce qu'il a terminé (interrompu) l'exécution de sa méthode **actions()**.

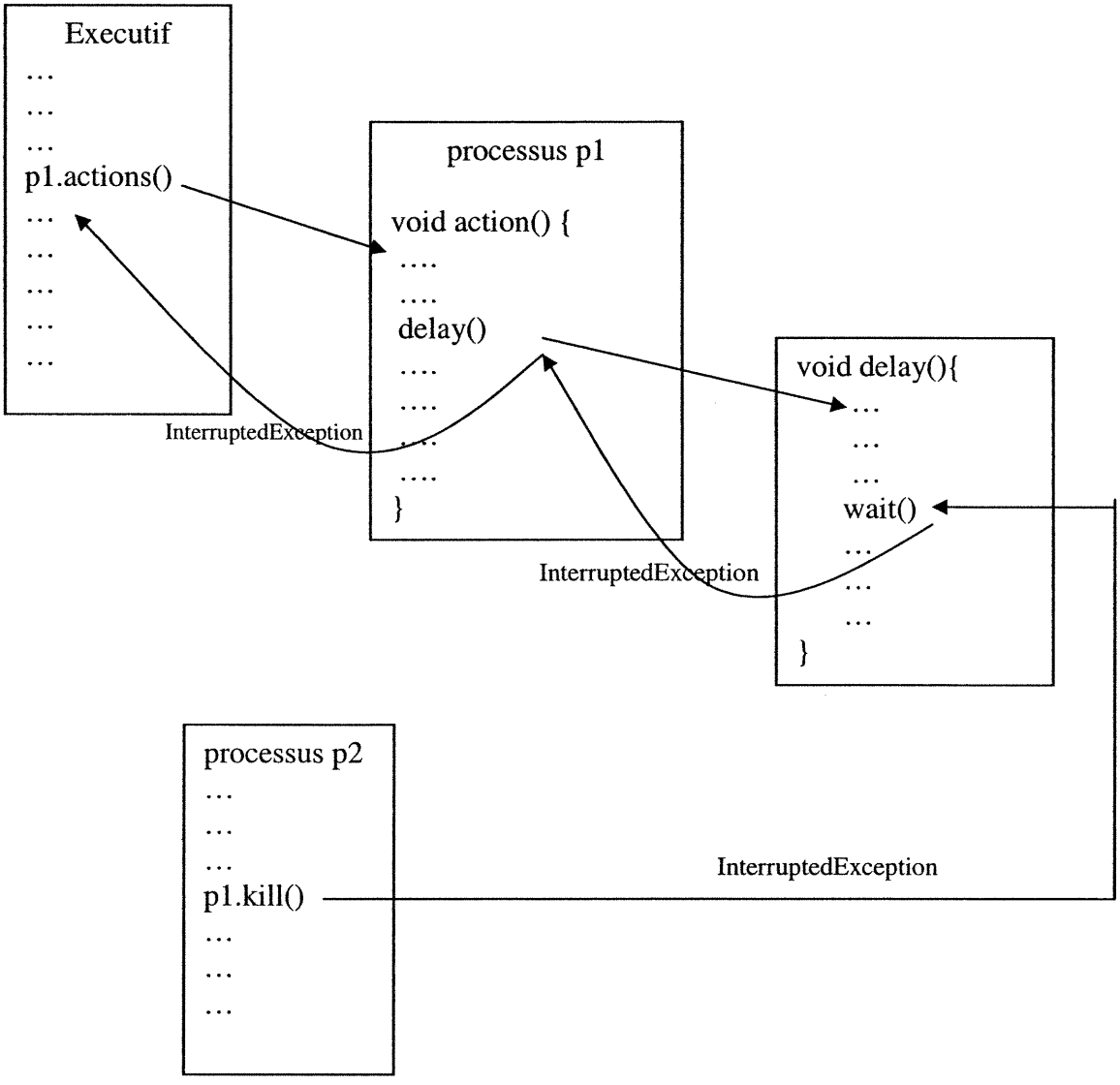


Figure 5.4 : Stopper un processus via la méthode **interrupt()**

La deuxième solution, adoptée dans SSJ, permet de mettre fin à la vie d'un processus d'une façon convenable. Ainsi, les méthodes **kill**() et **killAll**() (données ci-bas) sont bien implémentées et peuvent être appelées à n'importe quel moment pour stopper un processus particulier ou éventuellement tous les processus de l'utilisateur.

```
protected static void killAll () {
    // This method look for all processes still alive and kill them.
    // A process is alive if there is a thread associated to it.
    // A process is killed if there is no thread associated to it.
    // we can not kill a process blocked somewhere only by setting its associated thread to null. But
    // we have to make it came over its actions() method. this is done by interrupting the associated
    // thread and throwing an exception on it.
    Thread2  th2 = last;
    while (th2 != null) {
          if (th2.myProcess != null) th2.kill ();
          th2 = th2.next;
     }
}

  protected void kill ()   {
       if (eventTime >= 0.0)    // DELAYED state
                   Sim.eventList.remove (this);
       myThread.interrupt ();
       // this method does not interrupt the thread but it throws only an InterruptedException.
}
```

Suite à une proposition de J. Vaucher, nous avons amélioré cette solution en attrapant l'exception **InterruptedException** dans la méthode **passivate**() (le seul endroit où les processus sont susceptibles d'être bloqués) et en soulevant une autre exception de type **Error** que l'utilisateur n'est pas sensé prendre en compte. Cette derrière exception est attrapée par SSJ dans la méthode **run**() de la classe **Thread2** (voir la définition de la méthode **run**() dans la sect.5.4 et la définition de la méthode **passivate**() dans la sect.5.4.4.1)

5.4.5 Processus avec priorité

SSJ implémente le concept de ressource comme un mécanisme de synchronisation de processus (voir définition de la classe **Resource** dans l'annexe A). Ce concept est utile dans une vision par processus où les entités actives (processus) demandent un certain nombre d'unités de la ressource, utilisent ces unités, puis les libèrent après en avoir terminé. Un processus peut obtenir, simultanément, des unités de plusieurs ressources différentes et il peut aussi demander des unités avec priorité. L'implémentation de la classe **Resource** ressemble au module **Res** de SIMOD (voir L'Ecuyer [1988]).

Cependant, nous avons adopté une façon différente pour la gestion des priorités associées aux processus.

À chaque objet de type ressource sont associées deux listes (des instances de la classe **List**), la liste des processus en attente pour la ressource **(waitingList)** et la liste des processus en service (**serviceList**). La première liste sert comme une file d'attente de capacité infinie dont la politique de service est fixée par l'utilisateur au moment de la création de la ressource. Cette politique peut être **FIFO** (premier arrivé, premier servi) ou **LIFO** (dernier arrivé, premier servi). Par défaut, cette politique est de type **FIFO**. Au départ, tous les processus qui demandent des unités d'une ressource via la méthode **request(int nbUnit)** ont une priorité égale à 0. Cependant, un processus peut demander une ressource avec priorité par le biais de la méthode **request(int nbUnité, double priority).** En SIMOD, la politique de gestion de la file d'attente associée à une ressource peut être **Fifo**, **Lifo** ou **Prior** (avec priorité). Si la politique est **Prior**, l'utilisateur doit appeler la procédure **RequestWithPrior()** au lieu de **Request().**

5.4.6 L'implémentation du processus principal

L'exécutif de la simulation (dit processus principal) est un processus de type particulier, qui s'occupe de gérer l'horloge de la simulation, de trouver l'événement imminent et le déclencher et de synchroniser les autres processus ou, éventuellement, transférer le contrôle à un processus. Dans ce dernier cas, le processus qui est en train d'exécuter ses instructions s'occupe de passer le contrôle à un autre processus et d'exécuter les événements s'il y en a. À la fin de la simulation, l'exécutif reprendra le contrôle (voir l'implémentation de la méthode **dispatch**() à la sect.5.4.4). La façon d'implémenter ce processus influence les performances. De ce fait, nous avons expérimenté plusieurs façons :

- Considérer l'exécutif comme un processus s'exécutant dans son propre thread. Ce thread est défini dans une classe interne et anonyme qui implémente l'interface Java **Runnable**. La méthode **run()** de cette classe définit le comportement de ce processus.

- La deuxième consiste à considérer le processus principal comme un processus particulier des processus usagers (voir l'implémentation de la classe **Process** plus loin). La méthode **actions**() de ce processus définit son comportement.
- La troisième façon, adoptée dans SSJ, consiste à confondre le programme principal («main thread») et l'exécutif de la simulation. C'est à dire que le programme principal joue le rôle de l'exécutif de la simulation en appelant la méthode **static start**() de la classe **Sim** (donnée ci-après).

```
public static void start () {
      if (eventList.isEmpty ())
            error ("Sim.start with empty event list");
      Event ev;
      while (!stopped && (ev = eventList.removeFirst())!=null) {
            currentTime = ev.eventTime;
            ev.eventTime = -10;
             ev.actions ();
      }
}
```

Cette méthode cherche l'événement se trouvant à la tête de la liste d'événements et l'exécute. Lorsque **ev** est un événement pour activer un processus, la méthode **actions**() de cet événement sert à transférer le contrôle à cet objet et à bloquer l'exécutif (voir sect.5.4.1). La méthode **start**() s'arrête lorsque la simulation sera stoppée ou bien lorsqu'il n'y a plus d'événements dans la liste **eventList**.

Cette façon d'implémenter le processus principal apporte un gain en terme de performance (voir tableau 5.1). En effet, nous n'avons plus besoin de créer un nouveau thread et lui passer le contrôle, puis de détruire ce thread à la fin de la simulation et repasser le contrôle au programme principal. Notons que la création et la destruction d'un thread, ainsi que le transfert du contrôle d'un thread à un autre, sont des opérations coûteuses en terme de performances.

5.5 Génération des variables aléatoires selon des lois de probabilités.

Pour la génération de valeurs aléatoires non uniformes, nous avons été confrontés à plusieurs alternatives que nous avons, d'ailleurs, toutes implémentées. Toutefois, nous avons adopté celle qui offre le plus de souplesse et qui est très simple à utiliser et à étendre. Dans ce qui suit, nous discutons de ces alternatives.

Première alternative : Chaque loi de probabilité est implémentée comme une classe héritant de la classe appelée **Variate**. Cette dernière classe contient une référence à un « stream », qui est, par défaut, de type **Mrg32k3**. Ce « stream » peut être changé au moment de la création de la loi ou plus tard en invoquant une méthode qui prend en paramètre le nouveau « stream ». La classe **Variate** contient une méthode abstraite appelée **nextValue(),** qui sert à générer la valeur désirée. Elle doit être redéfinie dans toutes les sous-classes implantant des lois de probabilité. En outre, la classe **Variate** contient quelques méthodes pour la manipulation du générateur sous-jacent. Toutefois, l'utilisateur a la possibilité de manipuler directement ce générateur en utilisant la méthode **getStream().** Toutes les classes qui implantent une loi de probabilité doivent hériter de la classe **Variate** en redéfinissant la méthode abstraite **nextValue().**

Cette manière d'implémenter les générateurs non uniformes a l'avantage d'être simple et flexible. Les paramètres de la classe ne sont fixés qu'une seule fois au moment de la création de l'objet. Toutefois, si on veut implanter beaucoup de lois de probabilité, on est obligé de créer autant de classes que de lois. De plus, cette façon de voir les choses est moins intuitive étant donnée que la génération de variables aléatoires selon différentes lois de probabilités se base sur des «streams». Il est plus intuitif de les considérer comme étant des méthodes d'une classe qui contient le « stream ».

Deuxième alternative : Cette deuxième solution consiste à créer un petit nombre de classes utilitaires, chacune implémentant un groupe de lois comme des méthodes statiques d'une classe abstraite qui ne peut être instanciée. Ces méthodes reçoivent en paramètre le générateur de nombres aléatoires (« stream ») à utiliser et les paramètres de la loi de probabilité.

Cette façon de faire est la même que celle utilisée dans SIMOD (L'Ecuyer [1988]). Tous les membres de cette classe sont statiques. L'utilisateur peut utiliser les méthodes de cette classe directement sans créer un objet. Les «streams» doivent être crées avant l'appel de la méthode, et ils doivent être passés en argument avec les autres paramètres de la loi à chaque appel de cette fonction.
Cette façon d'implémenter les lois de probabilité n'est pas extensible d'une part, et d'autre part, elle n'est pas conforme au style de conception orientée objet qui favorise l'identification et la représentation des liens sémantiques entre les objets pour augmenter la réutilisation et l'extensibilité.

Troisième alternative : Cette solution, adoptée dans SSJ, est une combinaison des deux méthodes précédentes; elle consiste à créer une classe de base appelée **Variate** et une sous classe **Random1** qui implante les lois de probabilité les plus usuelles comme étant des méthodes membres. La classe **Random1** hérite de la classe **Variate** qui contient un lien vers l'interface **RandomStream,** de type agrégation. Elle implémente quelques lois de probabilité usuelles. Notons que le «stream» utilisé est référencé dans la super classe **Variate**.

Cette manière d'implémenter les variables aléatoires est très flexible et très facile à utiliser et à comprendre. L'utilisateur a deux façons d'implanter d'autres lois :

- Par dérivation de nouvelles classes à partir de la classe **Random1**. Les lois de probabilités seront les fonctions membres de ces nouvelles classes.
- Par extension de la classe **Variate.** Chaque nouvelle fonction de distribution peut être implémentée comme une sous classes de la classe **Variate** ou bien comme une fonction membre d'une classe qui regroupe un ensemble de lois; cette nouvelle classe hérite de la classe **Variate.**

5.6 Comparaison entre SSJ et SIMOD

Même si SSJ reproduit beaucoup de fonctionnalités de SIMOD, il diffère de ce dernier sur plusieurs plans.

- La conception et l'implémentation de SSJ sont complètement orientées objet. Par conséquent, plusieurs méthodes et concepts de SIMOD ne trouvent plus de place en SSJ.
- SSJ est aussi performant que SIMOD. Le tableau 5.4 présente les résultats de comparaison de vitesse avec SIMOD (les chiffres indiquent le temps CPU, en secondes, pris par les exemples). Les exemples **QueueProc** et **QueueEv** sont exécutés avec 10^6 clients (approximativement), l'exemple **BankEv** est exécuté avec 1000 répétitions. Le tableau montre que SSJ est très rapide avec la vision par événements et dépasse SIMOD. La vision par processus donne des résultats comparables à SIMOD en terme de performances.

Exemples	**SIMOD**	**SSJ avec javacomp JDK 1.2.006**	**SSJ avec HotSpot JDK1.3.1**
QueueEv (10^6)	9.36	9.32	4.09
QueueProc(10^6)	19.07	23.13	-
BankEv (1000 répétitions)	1.96	1.92	1.35

Tableau 5.4: Comparaison de performances entre SSJ et SIMOD

- L'architecture de SSJ est plus ouverte et facile à étendre que celle de SIMOD. L'enrichissement et l'extension de SSJ s'effectueront très facilement dans le futur. Ceci inclut spécialement l'implémentation des nouvelles structures pour la liste d'événements et l'ajout des nouveaux générateurs de nombres aléatoires ou des générateurs de variables aléatoires selon différentes lois de probabilité.
- SSJ adopte une façon efficace pour synchroniser les processus en ne réanimant plus l'exécutif une fois que le contrôle est transféré à un processus. En SIMOD c'est

toujours l'exécutif qui synchronise les autres processus en reprenant le contrôle à chaque fois qu'un processus le libère.

- SSJ adopte une façon très simple et intuitive pour demander une ressource avec priorité qui consiste à associer une priorité par défaut égale à 0 aux processus qui n'ont pas de priorité. La politique de la gestion des files d'attentes (associées aux objets de type **Resource)** est de type **FIFO** ou bien **LIFO**. SIMOD dispose de deux fonctions différentes pour demander une ressource avec priorité ou sans priorité et de trois politiques de gestion des files d'attentes à savoir **FIFO**, **LIFO** et **PRIOR**. L'utilisateur de SIMOD doit invoquer la bonne méthode selon la politique choisie.
- Le paquetage **random** est implémenté d'une façon différente de SIMOD. Ce paquetage est extensible, simple et très flexible.
- En plus de ces différences, il y a d'autres différences moins importantes. À titre d'exemples, SSJ implémente le processus principal d'une façon simple et efficace qui consiste à faire exécuter les instructions de ce processus par le «main thread». Il utilise une technique simple pour trouver le processus courant en mettant à jour une variable juste avant de donner le contrôle à un autre processus. Les états des processus ne sont plus conservés dans une variable comme SIMOD utilise, mais sont déduits automatiquement à partir de la variable **eventTime** associée aux événements. SSJ se sert du thread associé à un processus pour activer ce dernier.

Chapitre 6

Conclusion

Dans le cadre de ce travail, nous avons, d'abord, examiné en détails tous les modules de SIMOD, une librairie de style procédurale. Ensuite, nous avons étudié d'autres librairies de simulation basées sur le langage Java. Notre objectif était de concevoir et de développer un cadre d'application qui reproduit toutes les fonctionnalités de SIMOD dans un style orienté objet, simple, flexible, extensible et aussi performant que SIMOD. Pour ce faire, plusieurs façons étaient à envisager, la plus facile aurait été de convertir directement les modules de SIMOD en Java sans toucher à la conception ni à l'implémentation. Mais cette manière de faire aurait donné un programme volumineux, compliqué, très lent et peu extensible. Ceci nous a amené à explorer d'autres alternatives et possibilités, qui permettent une bonne conception orientée objet et une implémentation efficace. Dans plusieurs cas, nous avons apporté des solutions différentes de celles adoptées par SIMOD. Nous citons à titre d'exemple, l'implémentation du concept des priorités associées aux processus.

Tout au long de la réalisation de SSJ, une attention particulière était accordée aux points suivants :

- Efficacité : pour augmenter les performances, nous avons expérimenté plusieurs alternatives et exécuté des tests pour déterminer les coûts liés à ces différentes alternatives. L'implantation de quelques mesures particulières avait un effet considérable sur les performances (voir sect. 5.3). Parmi ces mesures, nous citons : le recyclage des threads, le passage du contrôle directement d'un processus à un autre sans ranimer l'exécutif, le recyclage des avis d'événements et les nœuds de tous les objets de type **List**, l'utilisation du «main thread» pour exécuter les

instructions du processus principal, etc. Selon nos tests de comparaison, SSJ est très rapide avec la vision par événements et surpasse SIMOD en termes de performances. Les performances avec la vision par processus se comparent avec celles de SIMOD. La vision par processus est ralentie par l'utilisation des threads et des méthodes Java **notify**() et **wait**() qui doivent être synchronisées.

- Synchronisation de processus : pour la synchronisation des processus, nous avons expérimenté différentes alternatives permettant de suspendre, réactiver ou stopper un processus. Cela est pour remplacer les méthodes dites «deprecated » de Java et qui sont nécessaires pour le fonctionnement d'un programme de simulation utilisant la vision par processus. Nous avons opté pour une solution simple, efficace et fiable, qui utilise le principe de sémaphore.
- Flexibilité et extensibilité : Nous citons deux cas que nous jugeons importants : la génération des nombres (et variables) aléatoires et le choix de la liste d'événements. Pour la génération des nombres et variables aléatoires, nous avons étudié beaucoup de possibilités. Nous avons discuté les inconvénients et les avantages de chacune d'elles. Nous avons, enfin, opté pour une solution simple et extensible permettant à l'utilisateur de changer et/ou d'ajouter facilement d'autres générateurs de nombres aléatoires uniformes. Cette solution donne aussi la possibilité à l'utilisateur de définir facilement d'autres variables aléatoires selon des lois de probabilités non prévues. Pour le choix de la structure de la liste d'événements, l'utilisateur peut choisir une structure parmi celles disponibles ou bien de définir sa propre structure à utiliser.

Le développement de SSJ a montré que l'utilisation de Java pour la simulation a beaucoup d'avantages sur le plan de simplicité, flexibilité, extensibilité et même efficacité. Selon notre expérience, les nouvelles versions de Java sont rapides et compétitives avec les langages performants tel que C++, le langage C et Modula-2. Ceci grâce à l'introduction des compilateurs de type JIT et HotSpot qui se basent sur le principe d'une compilation dynamique. Cette compilation est de type adaptative dans le cas de HotSpot (voir sect. 5.3). Cependant, le développement de SSJ montre aussi que les threads de Java ne sont pas adaptés pour implémenter les processus de la simulation.

Ils (les threads de Java) sont faits pour supporter des applications qui ont de vrais parallélismes. Ceci est très utile pour pouvoir tirer profit des architectures multi-processeurs en répartissant (pour plus d'efficacité) les tâches indépendantes sur les différents processeurs. SSJ, qui utilise un pseudo-parallélisme, s'exécute correctement et plus efficacement avec les «green thread» qui ne sont plus supportés dans les versions récentes de Java. L'utilisation des threads natifs (les seuls disponibles dans les versions récentes) ne peut donner de résultats que sur une machine mono-processeur et pour des programmes manipulant un nombre limité de processus simultanément. Même dans ce dernier cas, les threads natifs sont plus lents comparativement aux «green thread».

Pour des fins de comparaison, nous avons consulté plusieurs librairies de simulation, notamment en Java. La librairie Silk a été l'objet d'une plus grande attention de notre part par rapport aux autres librairies. Nous avons développé quelques exemples en Silk. Les comparaisons ont montré que l'utilisation de SSJ est beaucoup plus avantageuse que celle des autres librairies (spécialement Silk) sur plusieurs plans : efficacité simplicité, extensibilité et puissance. Ainsi, SSJ est très rapide grâce à l'implantation de certaines mesures très efficaces (citées au deuxième paragraphe de ce chapitre). Il utilise des générateurs de nombres aléatoires de bonne qualité et fournit plusieurs «streams» faciles à manipuler. Ceci permet de comparer deux configurations différentes du même système en utilisant les mêmes séquences de nombres aléatoires, chose qui n'est pas facile avec les autres librairies. Pour la synchronisation de processus, en plus du concept **Resource**, SSJ implémente le concept de **Bin** et de **Condition**. Il permet l'utilisation conjointe des événements, processus et la simulation continue. Ceci montre la puissance et la richesse de SSJ comparativement aux autres librairies. En plus, SSJ donne la possibilité à l'utilisateur de choisir facilement aussi bien le générateur de nombres aléatoires uniforme que la structure à utiliser pour la liste d'événements.

Il est illusoire de vouloir prétendre que ce travail ne nécessitera pas de futurs développements. SSJ pourrait être enrichi de multiples façons dans les versions futures, éventuellement :

- SSJ dispose de quelques outils de base pour le recueil des statistiques. Toutefois, ces outils peuvent être enrichis en implémentant d'autres mécanismes de recueil et de présentation de statistiques.
- Développer des outils d'analyse et d'animation des résultats de simulation. Ainsi que des outils d'analyse de la réduction de la variance.
- Implémenter les autres structures de la liste d'événements comme arbre binaire, arbre rouge et noir, splay tree, etc.
- Pour la génération des nombres aléatoires uniformes et les variables aléatoires selon les différentes lois de probabilité, SSJ dispose du générateur **Mrg32k3** défini par L'Ecuyer [2001, 2002a] et quelques lois de probabilité très usuelles. Il reste à développer d'autres générateurs uniformes et des générateurs de variables aléatoires selon d'autres lois de probabilité.
- Il serait intéressant de développer des environnements graphiques de simulation par domaine d'application en utilisant SSJ comme un noyau.
- Il serait aussi intéressant de développer des outils pour permettre aux utilisateurs distants d'écrire des programmes de simulation en SSJ et les exécuter via l'Internet.

Bibliographie

Arjuna, [1994] : "C++SIM User's Guide", Arjuna project, http://cxxsim.ncl.ac.uk/.

Banâtre, J. P. [1991] : "La Programmation Parallèle : Outils, Méthodes et Éléments de Mise en Œuvre", Eyrolles, Paris.

Banks, J. [1998a] : "Software for Simulation", chapitre 25 de Handbook of Simulation, ed: J.Banks, John Wiley & Sons, New York, pp. 813-835.

Banks, J. [1998b] : "The Future of Simulation Software: A Panel Discussion", Proceedings of the 1998 Winter Simulation Conference, ed. D. J. Medeiros, E. F. Watson, J. S. Carson et M. S. Manivannan, pp. 7-14.

Banks, J. [2000] : "Getting Started With AutoMod", AutoSimulations, Inc.

Banks, J., J. S. Carson II, B. R. Nelson et D. M. Nicol [2000] : "Discrete-Event System Simulation", 3ème Edition, Prentice-Hall, New Jersey.

Bapat, V. et N. Swets [2000] : "The Arena Product Family : Entreprise Modeling Solutions", Proceedings of the 2000 Winter Simulation Conference, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 163-169.

Biles, W. E. [1995] : "Discrete-Event Systems" dans Systems Modeling and Computer Simulation, 2ème Edition, Naim A. Kheire.

Birtwistle, G. [1979] : "DEMOS – A System for Discrete Event Modeling on Simula", MacMillan, 1979.

Booch, G. [1995] : "Object Solutions : Managing the Object-Oriented Project", Addison-Wesley.

Borland [1998] : http://www.borland.com/jbuilder/jb3/linux/download/jit_steps.html

Bratley, P., B. L. Fox et L. E. Schrage [1987] : "A Guide to Simulation", Springer-Verlag, New York, 2ème Edition.

Bryant, R. M. [1980] : "SIMPAS, a Simulation language Based on Pascal", Proceedings of the 1980 Winter Simulation Conference, pp. 25-40.

Bulka, D. [2000] : "Java Performance and Scalability, Server-Side Programming Techniques", Volume 1, Addison Wesley.

Burns, A. et G. Davies [1993] : "Concurrent Programming", Addison-Wesley.

Buss, A. H. et K. A. Stork [1996] : "Discrete Event Simulation on the World Wide Web using Java", Proceedings of the 1996 Winter Simulation Conference, ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, pp.780-785.

Buss, A [2001] : "Discrete Event Programming with Simkit", Simulation News Europe (32/33), pp. 15-26.

Chiao, H.-T. et S.M. Yuan [2001] : "An Enhanced Thread Synchronization Mechanism for Java", Software - Practice and Experience, 31. pp. 667-695.

Coe, P. S., F. W. Howell, R. N. Ibbett et L. M. Williams [1998] : "A Hierarchical Computer Architecture Design And Simulation Environment", ACM Transactions on Modeling and Computer Simulation, 8(4): 431-446.

Côté, S. [1990] : "Un progiciel de Simulation Basé sur ADA", Mémoire de Maîtrise Dirigé par L'Ecuyer, P., École des gradués, Université Laval.

Dahl, O. J. et K. Nygaard [1967] : "SIMULA : A language for Programming and Description of Discrete Event Systems", Oslo 3, Norway, Norwegian Computing Center, Forskningveien 1B, 5ème Edition, September 1967.

Delfosse, C. M. [1976] : "Continuous and Combined Simulation in SIMSCRIPT II.5", C.A.C.I,, Inc. Arlington, Va.

Diskstra, E.W. [1968] : "Cooperating sequential processes", dans F.Genuys (ed.) Programming languages, pp. 43-112, New York, Academic Press.

Fayad, M. E., D. C. Schmidt et R. E. Johnson [1999] : "Building Application Frameworks, Object-Oriented Foundations of Framework Design", John Wiley & Sons, Inc, New York.

Frantz, F. K. et K. C. Trott [1984] : "Extension to Pascal for Discret Event Simulation", dans Simulation in Strongly Typed Languages : ADA, PASCAL, SIMULA..., Simulation Serie 13(2), pp. 32-36.

Gnu [2001] : "The GNU Compiler for the Java™ Programming language", http://gcc.gnu.org/java/index.html.

Gordon, G. [1978] : "The Development of the General Purpose Simulation System (GPSS)", ACM SIGPLAN Conference on History of Programming Languages, SIGPLAN Notices 13(8), pp. 183-198.

Gordon, G. [1979] : "The Design of GPSS Language", dans Current Issues in Computer Simulation, N.R. Adams et A. Dogramaci, John Wiley, New York, pp.15-26.

Grady, B., J. Rumbaugh et I. Jacobson [1999] : "The Unified Modeling Language User Guide", Addison-Wisley.

Harrel, C. R. et R. N. Price [2000] : "Simulation Modeling and Optimisation using ProModel", Proceedings of the 2000 Winter Simulation Conference, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 197-202.

Harrel, C. R. et R. N. Price [2000] : "Healthcare Simulation Modeling and Optimisation Using MedModel", Proceedings of the 2000 Winter Simulation Conference, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 203-207.

Healy, J. J. et R. A. Kilgore [1997] : "Silk : A Java-Based Process Simulation Language", Proceedings of the 1997 Winter Simulation Conference, ed. S. Andradottir, K. J. Healy, D. H. Withers, and B. L. Nelson, pp. 475-482.

Healy, K. J. et R. A. Kilgore [1998] : "Introduction to Silk and Java-Based Simulation", Proceedings of the 1998 Winter Simulation Conference, ed. D. J. Medeiros, E. F. Watson, J. S. Carson and M. S. Manivannan, pp. 327-334.

Helsgaun, K. [1980] : "DISCO – A SIMULA-Based Language for Continuous Combined and Discrete Simulation", Simulation, Juillet 1980, pp. 1-12.

Holub, A. [2000] : "Taming Java Threads", Apress, (distribué par Springer-Verlag, NY).

Howell, F. et R. McNab [1998a] : "Simjava : A Discrete Event Simulation Package for Java with Application in Comuper Systems Modeling", First International Conference on Web-based Modeling and Simulation, San Diego, CA, Society for Computer Simulation.

Howell, F. et R. McNab [1998b] : "A Guide to the `Simjava` Package", dans : http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.2/doc/simjava_guide/index.html.

Hyde, P. [1999] : "Java Thread Programming", SAMS.

ImagineThat, Inc. [1997] : "Extend User's Manual, Version 4", San Jose, California.

Jade Simulations International [1992] : "Sim++ User Manual", Jade Simulations International Corporation.

Johnson, E. R et B. Foote [1988] : "Designinig Reusable Classes", Journal of Object-Oriented Programming, juin/juillet 1988, pp. 22-35.

Joines, J. A. et S. D. Roberts [1996] : "Design of Object-Oriented Simulations in C++", Proceedings of the 1996 Winter Simulation Conference, ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, pp.65-72.

Joines, J. A. et S. D. Roberts [1998] : "Object-Oriented Simulation", dans Handbook of Simulation, ed. J.Banks, John Wiley & Sons, Inc. New York [1998], pp. 397-427.

Joines, J. A et S. D. Roberts [1999] : "Simulation in an Object-Oriented World", Proceedings of the 1999 Winter Simulation Conference, ed. P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, pp. 132-140.

Kalasky, D. R. et G. A. Levasseur [1997] : "Using SiMPLE++ for Improved Modeling Efficiences and Extending Model Life Cycles", Proceedings of the 1997 Winter Simulation Conference, ed. S. Andradottir, K. J. Healy, D. H. Withers, and B. L. Nelson, pp. 611-618.

Kapuno, Jr., R. Raymond et N. N. Nagarur [1999] : "SimProd : A Web-Based Flexible Simulation Package for Production Systems", EJSAT, July 1999, http://www.sat.ait.ac.th/ej-sat/articles/1.2/ng.html.

Karr, H. W., H.Kleine et H. M. Markowitz [1965] : SIMSCRIPT I.5., CACI-65-INT-1, California Analysis Center, Inc.

Kilgore, R. A. [2000] : "Silk, Java and Object-Oriented Simulation", Proceedings of the 2000 Winter Simulation Conference, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 246-252.

Kiviat, P. J., R Vilanueva et H. M. Markowitz [1971] : "The SIMSCRIPT II Programming Language", Prentice-Hall.

Kreutzer, W. [1986] : "System Simulation Programming Styles and Languages", Addison-Wesley.

Lapalme, G. et J. Vaucher [1989] : "Programmation Orientée Objet Depuis Simula-67 : Progrès et Reculs", Journée sur les Bases de Données Orientées Objet, Montréal, ACFAS.

Law, A. M. et W. D. Kelton [2000] : "Simulation Modeling and Analysis", 3ème Edition, McGraw-Hill, New York.

Lea, D. [2000] : "Concurrent Programming in Java : Design Principles and Patterns", Addison Wesley.

L'Ecuyer, P. [1987] : "SIMPascal 2.0 : Guide de l'usager pour VAX/VMS", Rapport Technique DIUL-RT-8804, Département d'Informatique, Université Laval.

L'Ecuyer, P. [1988] : "SIMOD : Définition Fonctionnelle et Guide D'Utilisation (version 2.0)" Technical-Report DIUL-RT-8804, Département d'Informatique, Université Laval, Sept 1988.

L'Ecuyer, P. [1999] : "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators", Operations Research, 47(1) pp. 159-164.

L'Ecuyer, P. et C. Lemieux [2000] : "Variance Reduction via Lattice Rules", Management Science 46 (9) pp.1214-1235.

L'Ecuyer, P. [2001a] : "SSJ : A Java Library for Stochastic Simulation, Software user's guide".

L'Ecuyer, P. [2001b] : "Software for uniform random number generation : Distinguishing the Good and the Bad", Proceedings of the 2001 Winter Simulation Conference, pp.95-105.

L'Ecuyer, P., R. Simard, E. J. Chen, et W. D. Kelton [2002a] : "An Objected-Oriented Random Number Package with Many Long Streams and Substreams", Manuscript.

L'Ecuyer, P. [2002b] : "SSC : A Library for Stochastic Simulation in C, Software user's guide".

Little, M. C. [1999] : "JavaSim : User's Guide" dans : http://javasim.ncl.ac.uk/.

Lomow, G et B. Unger [1982] : "A Process View of Simulation in Ada", Proceedings of the 1982 Winter Simulation Conference, pp. 77-86.

Magee, J. et J. Kramer [1999] : "Concurrency : State Models & Java Programs", John Wiley & Sons.

Mangione, C. [1998] : "Performance Tests Show Java as Fast as C++", http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html.

Matthew, W. R. [2000] : "AutoMod Tutorial", Proceedings of the 2000 Winter Simulation Conference, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 170-176.

Meyer, B. [2000] : "Object-Oriented Software Construction", 2ème Edition, Prentice Hall.

McNab, R. et F. W. Howell [1996] : "Using Java for Discrete Event Simulation, in Proceedings of the Twelfth UK Computer and Telecommunications Performance Engineering Workshop, pp. 219-228, University of Edinburgh, UK.

Miller, J. A., R. S. Nair, Z. Zhang et H. Zhao [1997] : "JSIM : A Java-Based Simulation and Animation Environment" Proceedings of the 30th Annual Simulation Symposium, pp. 31-42.

Miller, J. A., Y. Ge et J. Tao [1998] : "Component-Based Simulation Environments : JSIM as a Case Study Using Java Beans", Proceedings of the 1998 Winter Simulation Conference, ed. D. J. Medeiros, E. F. Watson, J. S. Carson and M. S. Manivannan, pp. 373-381.

Mills, C. R. [1984] : "QSIM : A Language to Support Modular Large Scale Simulation", dans Simulation in Strongly Typed Languages : ADA, PASCAL, SIMULA..., Simulation Série, 13(2), pp. 22-24.

Modi T. [2000] : "Why Thread Pools are Important in Java : Use Thread Pools to Create better Performing, Multithreaded Java Applications", dans http://www.java-zone.com/upload/free/features/javapro/2000/10oct00/tm0010/tm0010.asp

Modi, T. [2001] : "Improve Java Performance", dans http://www.devx.com/premier/mgznarch/javapro/2001/01jan01/tm0101/tm0101.asp

Murphy, B. L. [2000] : "High Resolution Satellite Communication Simulation", Proceedings of the 2000 Winter Simulation Conference, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, pp. 1043-1049.

Nair, R. S., J. A. Miller et Z. Zhang [1996] : "Java-Based Query Driven Simulation Environment", Proceedings of the 1996 Winter Simulation Conference, ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, pp.786-793.

Nance, R. E. [1995] : "Simulation Programming Languages : An Abridged History", Proceedings of the 1995 Winter Simulation Conference, ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, et D. Goldsman, pp. 1307-1313.

Nikoukaran, J., V. Hlupic et R. J. Paul [1998] : "Criteria for Simulation Software Evaluation", Proceedings of the 1998 Winter Simulation Conference, D. J. Medeiros, E. F. Watson, J. S. Carson et M. S. Manivannan, pp. 399-406.

Pegden, C. D. [1985] : "Introduction to SIMAN Version 3.0", System Modeling Corporation, State College, Pennsylvania.

Pegden, C. D., R. E. Shannon et R. P. Sadowski [1995] : "Introduction to Simulation Using SIMAN", 2ème edition, McGraw Hill College Div.

Petersen, N. D. [1972] : "MIMIC, An Alternative Programming Language for Industrial Dynamics", J. Socio-Economic Plann. Sci. Vol. 6, pp. 319-327.

Pooch, U.W. et J. A. Wall [1993] : "Discrete Event Simulation : a Practical Approach", CRC Press.

Pooley, R. J. [1987] : "An Introduction to Programming in SIMULA", Oxford, Blackwell Scientific Publications.

Pritsker, A. A. B. [1974] : "The GASP IV Simulation Language", John Wiley & Sons.

Pritsker, A. A. B. [1986] : "Introduction to Simulation and SLAM II", John Wiley , New York.

Pritsker, A. A. B. et J. J. O'Reilly [1999] : "Simulation With Visual Slam and Awesim", 2ème edition, John Wiley & Sons.

Pugh, A. [1963] : "DYNAMO User's Manual", The M.I.T press, Cambridge, Massachusetts.

Russel, E. C. [1976] : "Simulation with Processes and Resources in SIMSCRIPT II.5", CACI Inc., Los Angeles.

Russel, E. C. [1983] : "Building Simulation Models with SIMSCRIPT II.5", CACI, Los Angeles.

Sadowski, D. et V. Bapat [1999] : "The Arena Product Family : Entreprise Modeling Solutions", Proceedings of the 1999 Winter Simulation Conference, ed. P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, pp. 159-166.

Sawhney, A., J. Manickam, A. Mund et J. Marble [1999] : "Java-Based Simulation of Construction Processes using Silk", Proceedings of the 1999 Winter Simulation Conference, ed. P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, pp. 985-991.

Schriber, T. J. [1974] : "Simulation Using GPSS", John Wiley & Sons, N.Y.

Schriber, T. J. [1991] : "An Introduction to Simulation using GPSS/H", John Wiley & Sons, New York.

Schwetman, H. [1996] : "CSIM18 – The Simulation Engine", Proceedings of the 1996 Winter Simulation Conference, ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, San Diego, CA, pp. 387-396.

Shannon, R. E. [1998] : "Introduction to the Art and Science of Simulation", Proceedings of the 1998 Winter Simulation Conference, ed. D. J. Medeiros, E. F. Watson, J. S. Carson et M. S. Manivannan, pp. 7-14.

Shirazi, J. [2000] : "Java Performance Tuning", O'Reilly.

Sun [1998] : http://www.sun.com/software/solaris/jit/download/.

Sun [1999] : « Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.run FinalizersOnExit Deprecated ? », http://java.sun.com/j2se/1.3/docs/guide/misc/threadPrimitiveDeprecation.html.

Sun [2001] : "The Java HotSopt Virtual Machine : Technical White Paper", http://www.sun.com/solaris/java/wp-hotspot/.

TowerJ [2001] : "TowerJ 3.8.1 User's Guide", http://www.towerj.com.

Vaucher, J. [1979] : "BETA: Un Successeur à SIMULA pour la Programmation des Systèmes", BIGRE 15, pp. 4-10.

Vaucher, J. [1984] : "Process-Oriented Simulation in Standard PASCAL", dans Simulation in Strongly Typed Languages : ADA, PASCAL, SIMULA..., Simulation Serie 13(2), pp. 37-43.

Vaucher, J. [1998] : "The Simula Web Site", à : http://www.iro.umontreal.ca/~simula/.

Vaucher, J. et B. Magnusson [1999] : "Simula Frameworks : The Early Years", chapitre 4 de Building Applications Frameworks : Object Oriented Foundations of Framework Design. M. Fayad, D. Schmidt et R. Johnson, John Wiley, pp. 89-142.

Wood, B. et K. Tumay [1999] : "MODSIM III and CACI's Applications", Proceedings of the 1999 Winter Simulation Conference, ed. P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, pp. 234-238.

Yellin, F. [1996] : "The JIT Compiler API", ftp://ftp.javasoft.com/docs/jit_interface.pdf.

Zeigler, B. P. [1997] : http://www-ais.ece.arizona.edu/SOFTWARE/software

Annexe A

Définition fonctionnelle des classes et méthodes de SSJ

paquetage kernel

Sim

public classe Sim

La classe **Sim** contient le processus principal de la simulation et les méthodes permettant de démarrer, arrêter ou initialiser la simulation.

public static double time ( )

Renvoie la valeur courante du temps de simulation.

public static void init ( )

Initialise la simulation en :

- tuant tous les processus existant s'il y a lieu,
- remettant à zéro l'horloge de la simulation,
- vidant la liste d'événements.

public static void init (EventList evlist)

Identique à la méthode **init()**, mais utilise l'instance **evlist** comme liste d'événements.

public static void start ( )

Démarre l'exécutif de la simulation. Il faut avoir, au moins, un événement dans la liste d'événements lorsque cette méthode est invoquée.

Public static void stop ( )

Demande à l'exécutif d'arrêter la simulation quand il reprend le contrôle et rend le contrôle au programme ayant appelé **Sim.start()**.

paquetage kernel

Event

public abstract class Event

La classe **Event** dispose des outils nécessaires pour la gestion des événements dans le contexte d'une simulation utilisant l'approche événementielle. Plusieurs types d'événements peuvent se produire et un même type d'événement peut survenir plus qu'une fois. Les événements sont utilisés aussi bien dans la construction des modèles de simulation utilisant la vision par événements que dans la simulation des modèles utilisant la vision par processus (pour réveiller les processus). Un événement se produit à un instant donné, provocant un changement d'état dans le système simulé ou donnant le contrôle à un processus pour débuter ou reprendre son exécution. Un type d'événement peut avoir des caractéristiques spécifiques à chaque occurrence d'événement de ce type.

La classe **Event,** qui est une classe abstraite**,** contient les outils nécessaires pour la gestion d'événements, mais elle n'est pas un type d'événement particulier. Pour chaque type d'événement, l'utilisateur doit créer une nouvelle classe héritant de la classe **Event** et redéfinissant la méthode **actions()** qui doit contenir les instructions décrivant les changements d'états effectués par cet événement. Ces instructions seront exécutées à chaque occurrence d'un événement de ce type. Les classes dérivées, représentant les événements, sont des classes Java ordinaires. Elles peuvent contenir de nouveaux attributs pour décrire les propriétés de l'événement et de nouvelles méthodes agissant sur ces attributs.

public Event ( )

Créer une nouvelle instance de type **Event** (un événement).

public Event (double delay)

Construit un nouvel événement et le place dans la liste d'événement. Si **delay** ≥ 0.0, l'événement est ordonnancé pour se produire dans un délai égal à **delay**.

public void schedule (double delay)

Planifie l'occurrence de l'événement à l'instant : **Sim.time()+delay** en insérant cet événement dans la liste d'événements. Lorsque deux ou plusieurs événements sont prévus pour le même instant, ils seront placés dans la liste (et exécutés) dans l'ordre selon lequel ils ont été prévus.

public void scheduleNext ( )

Place l'événement en tête de la liste d'événements (le prochain à exécuter). Il doit s'exécuter à l'instant courant. Si d'autres événements sont aussi prévus pour l'instant courant, l'événement que l'on prévoit maintenant s'exécutera en premier.

public void scheduleBefore (Event other)
Planifie l'occurrence de l'événement appelant juste avant l'événement **other**. Les deux événements auront la même date d'occurrence.

public void scheduleAfter (Event other)
Planifie l'occurrence de l'événement appelant juste après l'événement **other**. Les deux événements auront la même date d'occurrence.

public void reschedule (double delay)
Change la date d'occurrence de l'événement en planifiant son occurrence après un temps égal à **delay** à partir de l'instant courant.

public boolean cancel ()
Annule (enlève de la liste) l'occurrence de cet événement avant qu'il ne se produise. Retourne **true** si l'occurrence de l'événement appelant était déjà planifiée (et vient d'être annulée par cet appel), sinon **false** est retourné.

public double eventTime ( )
Renvoie le temps d'occurrence prévu pour cet événement.

public abstract void actions ( )
Décrit la logique de changement d'état lorsque cet événement se produira. Toutes les classes héritant de la classe **Event** doivent implémenter cette méthode.

public boolean cancel (String type)
Cherche dans la liste d'événement la première occurrence de l'événement de classe **type,** et l'enlève de la liste (annule son occurrence). Cette méthode retourne **true** si la recherche a réussi, sinon elle retourne **false**.

paquetage kernel

Process

public abstract class Process

La classe **Process** est une classe abstraite qui dispose des outils nécessaires pour la construction des modèles de simulation utilisant le paradigme d'interaction de processus. Elle offre des mécanismes pour la gestion et la synchronisation des processus. Pour chaque type d'entité active dans le système, l'utilisateur doit créer une nouvelle classe héritant de la classe **Process** et redéfinissant la méthode **actions()**. Cette dernière méthode doit décrire le comportement de l'entité en question.

public static final int INITIAL = 0
public static final int EXECUTING = 1
public static final int DELAYED = 2
public static final int SUSPENDED = 3
public static final int DEAD = 4

Ces constantes font référence aux différents états possibles d'un processus. **INITIAL** : le processus est crée mais son activation n'est pas encore planifiée; **EXECUTING** : le processus est en exécution; **DELAYED** : un événement est prévu pour lui donner le contrôle ultérieurement; **SUSPENDED** : l'exécution du processus est suspendue provisoirement; **DEAD** : le processus a terminé son cheminement.

public Process ( )

- Construit un nouveau processus et le place dans l'état **INITIAL**.
- Lui associe un thread.

Public Process (double delay)

- Construit un nouveau processus et le place dans l'état **DELAYED**.
- Planifie son activation dans un délai égal à **delay** unités de temps en insérant l'événement (le thread) qui lui associé dans la liste d'événement.

public void schedule (double delay)

- Planifie l'activation de ce processus dans **delay** unité de temps à partir de l'instant courant, en insérant un événement dans la liste d'événements.
- Met le processus dans l'état **DELAYED**

public void scheduleNext ( )

- Permet au processus de démarrer à l'instant courant de la simulation (dès que le processus appelant suspend ses activités) en plaçant son événement d'activation en tête de la liste des événements.
- Met le processus dans l'état **DELAYED.**

public static Process currentProcess ( )

Renvoie le processus en cours d'exécution s'il y a lieu. Sinon la valeur **null** est renvoyée.

public int getState ( )

Renvoie l'état du processus.

public double getDelay ( )

Si le processus est dans l'état **DELAYED**, cette méthode renvoie le temps restant pour l'activation de ce processus; sinon, le programme est arrêté avec un message d'erreur.

public static void delay (double delay)

Seulement un processus en exécution peut appeler cette méthode qui :

- Suspend l'exécution du processus en cours.
- Planifie un événement pour reprendre l'exécution dans un délai égal à **delay.**
- Met le processus dans l'état **DELAYED.**

public void reschedule (double delay)

Si le processus est dans l'état **DELAYED**, son événement d'activation sera annulé; ensuite un nouvel événement d'activation sera planifie pour redonner le contrôle au processus dans un délai égal à **delay**.

public void suspend ( )

Cette méthode ne peut être invoquée que par un processus dans l'état **EXECUTING** ou **DELAYED**, sinon le programme est arrêté avec un message d'erreur.

- Place le processus dans l'état **SUSPENDED**
- Si le processus est en cours d'exécution, son exécution sera suspendue. Si le processus est dans l'état **DELAYED**, l'événement prévu pour son activation sera annulé.

public void resume ( )

- Si ce processus est déjà dans l'état **DELAYED**, son événement d'activation sera annulé.
- Met l'événement d'activation de ce processus en tête de la liste d'événements.
- Place le processus dans l'état **DELAYED.**

public void cancel ( )

- Annule l'événement d'activation supposé réveiller ce processus
- Met le processus dans l'état **SUSPENDED**.

public void kill ( )

Met terme à la vie du processus en mettant son état à **DEAD,** si le processus est en état **DELAYED,** l'événement d'activation qui lui est associé est annulé.

public static void killAll ( )

Met terme à la vie de tous les processus usagers (toutes les instances de la classe **Process**).

public void abstract actions ()

C'est une méthode abstraite qui doit être implémentée par toutes les classes qui héritent de la classe **Process**. Elle doit décrire le comportement du processus en question.

paquetage kernel

Ressource

public class Resource

Le concept de ressource représente un centre de service dont la capacité correspond au nombre de serveurs dans ce centre. Ce concept est utile dans une vision par processus où les entités actives (processus) demandent un certain nombre d'unités de la ressource, utilisent ces unités, puis les libèrent après en avoir terminé. Un processus peut obtenir, simultanément, des unités de plusieurs ressources différentes.

À chaque objet de type ressource sont associées deux listes (des instances de la classe **List**), la liste des processus en attente pour la ressource et la liste des processus en service. La première liste sert comme une file d'attente de capacité infinie dont la politique de service est fixée par l'utilisateur au moment de la création de la ressource. Cette politique peut être **FIFO** (premier arrivé, premier servi) ou **LIFO** (dernier arrivé, premier servi). Par défaut, cette politique est de type **FIFO**. Au départ, tous les processus qui demandent des unités d'une ressource via la méthode **request(int nbUnit)** ont une priorité égale à 0. Cependant, un processus peut demander une ressource avec priorité par le biais de la méthode **request(int nbUnité, double priority).** Les processus demandant une ressource avec une priorité plus élevée passent avant ceux qui la demandent avec une priorité plus faible. La file d'attente est ordonnée selon les priorités; les processus de même priorités sont gérés selon la politique de gestion de la liste en vigueur. Si le processus en tête de liste demande plus d'unités que ce qui est disponible, la ressource est allouée au prochain processus dans la file dont la requête peut être satisfaite (si un tel processus existe). De même, si un processus attend dans la file parce que sa requête ne peut être satisfaite, et un second processus arrive, la requête de ce dernier sera satisfaite immédiatement s'il demande un nombre d'unité ne dépassant pas ce qui est disponible.

Les objets manipulés par les deux listes (liste des processus en attente et celle des processus en service) sont des instances de la classe appelée **UserRecord**. Cette classe contient en plus du processus en question, le nombre d'unités demandé, la priorité associée et le temps d'entrée dans la liste. Chaque fois qu'un processus demande une ressource, on lui ouvre un dossier de ce type. Deux blocs statistiques de type **Accumulate** et un bloc de type **Tally** (voir plus loin pour la définition des classes **Accumulate** et **Tally**) sont associés à chaque objet de type **Resource**. Ces blocs statistiques ne sont actifs que si l'utilisateur invoque la méthode **collectStat(true)** sur la ressource. Ils servent à mesurer, respectivement, l'évolution de la capacité, l'utilisation de la ressource en fonction du temps et les durées de séjour des processus dans la ressource. Ces blocs sont accessibles à l'utilisateur par les méthodes **statOnCapacity**(), **stateOnUtil**() et **statOnSojourn**(). La méthode **report**() permet d'obtenir un rapport statistique complet sur une ressource. Les listes sont accessibles à l'utilisateur par les méthodes **getWaitingList()** et **getServiceList()**, ce qui donne la possibilité à l'utilisateur de recueillir des statistiques seulement sur la liste qui l'intéresse.

public static final int FIFO = 1
public static final int LIFO = 2

Ces constantes indiquent les différentes politiques de service de la liste d'attente associée à une ressource. FIFO (premier arrivé, premier servi): les processus, qui ont les mêmes priorités, sont placés dans la liste et servi selon leur ordre d'arrivé. LIFO (premier arrivé, dernier servi) : les processus, qui ont les mêmes priorités, sont placés dans la liste et servis selon l'ordre inverse de leur arrivé. Les processus de priorités plus élevées sont toujours placés et servis avant ceux de priorités inférieures.

public Ressource (int capacity)

Construit une nouvelle ressource avec une capacité égale à **capacity** et une politique de service **FIFO**.

public Resource (int capacity, String name)

Construit une nouvelle ressource identifiée par **name** avec une capacité égale à **capacity** et une politique de service **FIFO.**

public Resource (int capacity, int policy, String name)

Construit une nouvelle ressource identifiée par **name** avec une capacité égale à **capacity** et une politique de service **policy.**

public void collectStat (boolean flag)

- Si **flag** est vrai (**true**), cette méthode démarre un recueil automatique des statistiques sur les listes des processus en attente et en service. Ces listes sont renvoyées par **waitList()** et **servList()** de cette ressource. Elle crée aussi trois autres blocs statistiques associés à cette Ressource. Ces collecteurs sont initialisés et mis à jour automatiquement. Ils peuvent être accédés par **statOnCapacity()**, **statOnUtil()** et **statOnSojourn()**. Les deux premiers sont des instances de la classe **Accumulate** et permettent de suivre l'évolution de la capacité et le taux d'utilisation de la ressource en fonction du temps. Le troisième, est une instance de la classe **Tally** et permet de recueillir des statistiques sur les durées de séjour des processus.
- Si le **flag** est faux (**false**), cette méthode arrête le recueil des statistiques.

public void initStat ( )

Réinitialise tous les collecteurs de statistiques pour cette ressource. Cette méthode ne peut être appelée si la méthode **collectStat(true)** n'a pas été appelée auparavant.

public void init ( )

Réinitialise cette ressource en vidant la liste d'attente et la liste de service associée à cette ressource. Les processus qui étaient dans ces listes restent dans le même état. Réinitialise tous les blocs statistiques associés à cette ressource, s'il y a lieu.

public int getCapacity ( )

Renvoie la capacité courante de la ressource.

public void changeCapacity (int diff)

- Modifie la capacité de la ressource (augmente si **diff**>0, diminue si **diff**<0).
- Si **diff** >0 et s'il y a des processus en attente dont la demande peut être satisfaite, ils obtiennent la ressource.
- Si **diff**<0, il doit avoir au minimum **diff** unités de ressource disponibles, sinon le programme s'arrête avec un message d'erreur.

public void setCapacity (int newcap)

Affecte la capacité de la ressource à **newcap** unités; elle est équivalente à **changeCapacity(newcap-old)** si **old** est la capacité courante de la ressource.

public int getAvailable ( )

Renvoie le nombre d'unités disponibles, c'est-à-dire, la capacité de la ressource moins le nombre d'unités occupées.

public void request (int n)

Le processus en cours d'exécution demande **n** unités de cette ressource. Si le nombre d'unités disponibles suffit pour répondre à cette requête, on lui accorde les unités demandées et il les détient jusqu'à ce qu'il appelle la méthode **release**() pour les libérer. Ce processus est inséré dans la liste des processus en service. Par contre, si ce nombre n'est pas suffisant, le processus est placé dans la liste d'attente dans l'état **SUSPENDED** jusqu'à ce qu'il obtienne les unités demandées. Tous les processus qui invoquent la méthode **request(int n)** ont une priorité par défaut égale à 0.

public void request (int n, double priority)

Similaire à la méthode **request(int n)**, sauf que le processus appelant demande la ressource avec une priorité égale à **priority.** Les priorités plus élevées passent avant les priorités plus faibles. Les processus ayant des priorités égales sont classés selon la politique de gestion de la file d'attente en vigueur (**LIFO** ou **FIFO**).

public void release (int n)

- Le processus en cours d'exécution libère n unités de cette ressource. Si ce processus occupait exactement n unités de la ressource, il est retiré de la liste des processus en service pour cette ressource. S'il occupait moins de n unités, le programme s'arrête avec un message d'erreur.
- S'il y a d'autres processus en attente pour cette ressource, les unités libérées sont maintenant disponibles et peuvent leurs être allouées.

public Accumulate statOnUtil ( )

Renvoie le bloc statistique qui mesure l'utilisation de la ressource en fonction du temps. Ce bloc n'existe que si on a déjà appelé la méthode **collectStats(true)** précédemment. Sinon le programme s'arrête avec un message d'erreur. À noter que le bloc renvoyé par **servList().statSize()** compte le nombre d'objets de type **UserRecord** dans la liste de service, ce qui n'est pas toujours la même chose que le nombre d'unités de la ressource occupées, car un dossier peut occuper plusieurs unités à la fois.

public List waitList ( )

Renvoie la liste contenant les objets de type **UserRecord** des processus en attente pour cette ressource.

public List servList ( )

Renvoie la liste des dossiers sur les processus en service. Les objets de cette liste sont de type **UserRecord**.

public Accumulate statOnCapacity ( )

Renvoie le bloc statistique qui mesure l'évolution de la capacité de la ressource en fonction du temps. Ce bloc existe seulement si **collectStats(true)** a été invoqué précédemment. Sinon, le programme s'arrête avec un message d'erreur.

public Tally statOnSojourn ( )

Renvoie le bloc statistique qui mesure les temps de séjour des objets de type **UserRecord** pour cette ressource. Ce bloc existe seulement si **collectStat(true)** a été invoqué précédemment, sinon le programme s'arrête avec un message d'erreur. Ce bloc prélève une observation à chaque fois qu'un processus libère toutes les unités de cette ressource dont il dispose.

public void report ( )

Imprime le rapport statistique de cette ressource. La méthode **collectStat(true)** doit être appelée auparavant. Ce rapport contient des statistiques sur les temps passés en attente et en service pour cette ressource, sur les durées de séjour, sur l'évolution dans le temps de la capacité, du nombre d'unités occupées, du nombre de processus utilisant la ressource, du nombre de processus en attente, et sur le taux d'utilisation de cette ressource.

paquetage kernel

Continuous

public abstract class Continuous

La classe abstraite **Continuous** fournit les outils de base pour la simulation continue, où l'évolution de certaines variables en fonction du temps est régie par des équations différentielles. Dans un même modèle, on peut mélanger des événements, des processus, et des variables continues obéissant à des équations différentielles. Pour chaque type de variable continue, l'utilisateur doit définir une classe héritant de la classe **Continuous** et redéfinissant la méthode **deriv**() qui retourne la dérivée de cette variable en fonction du temps. Les classes de l'utilisateur peuvent aussi redéfinir la méthode **afterEachStep**() qui sera immédiatement exécutée après chaque pas d'intégration pour cette variable. À chaque variable continue est associée un événement de type particulier qui sera placé dans la liste d'événements. La fonction membre **actions**() de cet événement déclenche la méthode d'intégration appropriée.

public static final int EULER = 1
public static final int RUNGEKUTTA2 = 2
public static final int RUNGEKUTTA4 = 4

Ces constantes font référence aux méthodes d'intégration utilisées. **EULER** veut dire la méthode Euler; **RUNGEKUTTA2** et **RUNGEKUTTA4** veulent dire la méthode Runge-Kutta d'ordre 2 et 4, respectivement.

public continuous ( )
Construit une nouvelle variable continue, sans l'initialiser.

public continuous (String name)
Construit une nouvelle variable continue avec un descripteur **name**. Celui-ci peut être utilisé pour identifier la variable continue dans les rapports.

Public void init(double initval)
Initialise ou réinitialise la variable continue avec la valeur **initval**.

public double value ( )
Renvoie la valeur courante de cette variable continue.

public static void selectIntegMethod (int method, double h)
Sélectionne la méthode d'intégration **méthode** à utiliser avec toutes les variables continues et la longueur du pas d'intégration **h**.

public void startInteg ( )
Démarre le processus d'intégration qui va changer l'état de cette variable à chaque pas d'intégration.

public void startInteg (double val)

Démarre le processus d'intégration qui va changer l'état de cette variable à chaque pas d'intégration après avoir initialisé la variable avec la valeur **val**.

public void stopInteg ( )

Arrête le processus d'intégration pour cette variable. La variable garde la valeur prise au dernier pas d'intégration, juste avant l'appel de **stopInteg**.

public abstract double deriv (double t)

Renvoie la dérivée de la variable continue par rapport au temps à un moment donné **t**. La méthode utilise **t** qui n'est pas forcément égal au temps courant. Toutes les classes qui héritent de la classe **Continuous** doivent implémenter cette méthode qui doit renvoyer la dérivé en fonction du temps.

public void afterEachStep ( )

Cette méthode est exécutée à chaque pas d'intégration de cette variable continue (juste après avoir effectué ce pas). Ici, la méthode ne fait rien, mais les sous-classes de **Continuous** peuvent l'implémenter.

paquetage kernel

Bin

public class Bin

La classe **Bin** fournit un autre mécanisme de synchronisation de processus. Un objet de type **Bin** correspond à une pile de jetons identiques et une liste des processus en attente des jetons lorsqu'il n'y a plus de jetons disponibles. La méthode **put**() permet d'ajouter des jetons à la pile. Un processus qui veut demander un jeton doit invoquer la méthode **take**(). Les objets de type **Bin** permettent d'établir des relations de type producteur/consommateur entre les processus. En effet, un appel à la méthode **put**() peut être vu comme une production de jetons et un appel à **take**() comme une consommation de ces jetons. Un processus qui demande un nombre de jetons plus grands que ce qui est disponible se voit bloqué et placé dans la file d'attente associée à l'objet **Bin.** La gestion de la file d'attente est semblable à celle utilisée dans la classe **Resource.** À chaque objet de type **Bin** est associée une file d'attente unique, de capacité infinie, et dont la politique de service peut être **LIFO** (premier arrivé, premier servi) ou **FIFO** premier arrivé, dernier servi). Les processus peuvent demander les jetons avec priorité. La file d'attente est une instance de la classe **List**. La méthode **waitList**() retourne cette liste, ce qui permet d'accéder aux blocs statistiques qui lui sont associés.

public static final int FIFO = 1
public static final int LIFO = 2

Ces constantes référencent les politiques de service de la liste d'attente des processus qui attendent pour des jetons d'un objet de type **Bin**. **FIFO** (premier arrivé, premier servi) : les processus seront placés dans la liste (et servis) selon leur ordre d'arrivée. **LIFO** (dernier arrivé, premier servi) : les processus seront placés dans la liste (et servis) selon l'ordre inverse de leurs arrivées.

public Bin (String name)

Construit un nouveau **Bin**, initialement vide, avec une politique **FIFO** et un identificateur **name**.

public Bin (int policy, String name)

Construit un nouveau **Bin**, initialement vide, avec une politique **policy** et un identificateur **name**.

public void init ( )

Réinitialise ce **Bin** par l'élimination de sa pile de jetons et en vidant sa liste d'attente. Les processus dans cette liste restent dans le même état.

public int getAvailable ( )

Renvoie le nombre de jetons disponibles pour ce **Bin**.

public void take (int n)

Le processus en exécution invoquant cette méthode demande **n** jetons de ce **Bin**. S'il y a suffisamment de jetons, le nombre de jetons disponibles est réduit de n et le processus continue son exécution. Sinon, le processus est mis dans la liste d'attente de ce **Bin** et son exécution est bloquée jusqu'à ce qu'il obtienne les jetons demandés.

public void put (int n)

Ajoute **n** jetons dans la pile des jetons associée à ce **Bin**. S'il y a des processus en attente de jetons et que leurs demandes peuvent être satisfaites, ils obtiennent les jetons et reprennent leurs exécutions.

public List waitList ( )

Renvoie la liste des dossiers sur les processus en attente pour ce **Bin**. Les objets de cette liste sont de type **UserRecord**.

paquetage kernel

Condition

public class Condition

La classe **Condition** offre un autre mécanisme de synchronisation de processus. Un objet de type **Condition** est un indicateur booléen associé à une liste qui contient les processus en attente pour cette **Condition**. Ces derniers restent bloqués jusqu'à ce que l'état de l'indicateur booléen devienne vrai. Un processus qui appelle la méthode **wait**() d'un objet de type **Condition** se voit bloqué si indicateur d'état de cet objet est positionné à faux. Il reprend son exécution lorsque cet indicateur devient vrai.

public Condition (boolean val, String name)

Construit une nouvelle condition avec une valeur initiale égale à **val.** Le paramètre **name** sert à identifier cette condition**.**

Public void init (boolean val)

Réinitialise cette condition en vidant sa liste d'attente et en affectant la valeur **val** à son état. Les processus de cette liste restent dans les mêmes états.

public void state ( )

Renvoie l'état (vrai ou faux) de la condition.

public boolean set (boolean val)

Attribue la valeur **val** à la condition. Si **val** est vrai, tous les processus bloqués derrière cette condition reprennent leurs exécutions dans l'ordre de l'appel d'attente de cette condition.

public void waitFor ( )

Le processus en exécution invoquant cette méthode doit attendre que cette condition devienne vraie. Si elle l'est déjà, le processus continue son exécution immédiatement, sinon, il est placé dans la liste d'attente de cette condition et est suspendu jusqu'à ce que la condition soit mise à vrai.

public List waitList ( )

Renvoie la liste des dossiers sur les processus en attente pour cette condition. Les objets de cette liste sont de type **UserRecord.**

paquetage kernel

List

public class List

Cette classe implémente une structure de liste doublement chaînée, elle contient des méthodes pour insérer des objets dans la liste, supprimer et obtenir des objets de la liste. Pour chaque objet de type **List**, on mémorise la position du dernier objet référencé ou inséré. Cet objet s'appelle l'objet courant. On peut retirer ou consulter l'objet courant, son successeur ou son prédécesseur, de même que le premier ou le dernier objet de la liste. Par un souci d'efficacité, les nœuds de tous les objets de type **List** sont réutilisés. Lorsqu'on veut supprimer un élément de la liste, on ne l'efface pas vraiment mais on l'ajoute à la pile des nœuds libres en manipulant seulement quelques variables. On peut demander un recueil automatique de statistiques sur un objet de type **List** en invoquant la méthode **collectStat(true)**. À chaque objet de type List est associé deux blocs statistiques de type **Accumulate** et **Tally**. Le premier mesure l'évolution de la taille de la liste en fonction du temps et le deuxième mesure les durées de séjour des objets dans la liste. Ces blocs statistiques ne sont créés que si on demande le recueil automatique de statistiques sur cette liste.

public static final int FIRST = 1;
public static final int LAST = 2;
public static final int PREVIOUS = 3;
public static final int NEXT = 4;
public static final int CURRENT = 5;

Ces constantes font référence à la position où l'on peut observer, retirer ou insérer un objet dans la liste. **FIRST** signifie en tête de liste; **LAST** signifie à la fin de la liste; **PREVIOUS** et **NEXT** signifient juste avant et juste après l'élément courant, respectivement; **CURRENT** signifie à la position courante.

public List ( )

Construit une nouvelle liste, initialement vide.

public List (String name)

Construit une nouvelle liste, initialement vide. Le paramètre **name** sert à identifier cette liste.

public void collectStat (Boolean flag)

- Si **flag = true**, cette méthode déclenche un recueil automatique de statistiques pour cette liste. Elle crée et initialise deux blocs. L'un, de type **Accumulate,** sert à mesurer l'évolution de la taille de la liste en fonction du temps, et l'autre, de type **Tally,** échantillonne les durées de séjour des objets dans la liste pendant la période d'observation, c'est-à-dire à partir de la dernière initialisation de ce bloc jusqu'à l'instant courant. Ces deux blocs sont accessibles, respectivement, par les méthodes **statSize()** et **statSojourn()**. La

méthode **collectStat**() appelle automatiquement **initStat**() pour initialiser ces deux blocs.

- Si **flag = false**, la méthode arrête la collecte des statistiques.

public void initStat ( )

Réinitialise les deux blocs statistiques crées par **collectStat()** et met à jour le bloc statistique mesurant l'évolution de la longueur de la liste. À noter que ces deux blocs doivent être créés auparavant par un appel à **collectStat(true).**

public void init ( )

Vide cette liste et initialise ses blocs statistiques si **collectStat(true)** a été déjà appelé auparavant.

public Object view (int where)

Renvoie l'objet se trouvant à la position **where** dans la liste sans le détruire, la position de cet objet devient la position courante. Si l'objet n'existe pas (si on tente d'observer l'objet précédent le premier de la liste, ou celui suivant le dernier de la liste, ou encore si la liste est vide), la valeur **null** est retournée.

public boolean belongs (Object obj)

Si **obj** est dans la liste, renvoie la valeur **true**, sinon retourne la valeur **false**. Lorsque la méthode retourne **true**, **obj** devient l'objet courant, sinon l'objet courant demeure inchangé.

public Object remove (int where)

Retourne et retire l'objet dont la position est indiquée par le paramètre **where**. Elle retourne la valeur **null** si cet objet n'existe pas. Si l'objet retiré est l'objet courant et que la liste n'est pas vide, le nouvel objet courant sera le suivant de celui retiré (ou le dernier de la liste, si l'objet enlevé était le dernier).

public int size ( )

Renvoie le nombre d'objets se trouvant dans la liste.

public void insert (Object obj, int where)

Insère l'objet **obj** dans cette liste. Selon la valeur du paramètre **where**, l'objet est inséré au début de la liste (**FIRST**), à la fin (**LAST**), avant l'objet courant (**PREVIOUS**) ou après l'objet courant (**NEXT**). L'objet **obj** devient l'objet courant.

public boolean empty()

Renvoie la valeur **true** si la liste est vide et **false** sinon.

public Tally statSojourn ( )

Retourne le bloc statistique qui mesure les durées de séjour des objets dans la liste. Il s'agit d'un bloc de type **Tally**. Ce bloc n'existe que si on a déjà appelé **collectStat(true)** pour cette liste. Sinon un message d'erreur est affiché.

public Accumulate statSize ( )

Renvoie le bloc statistique qui mesure l'évolution de la longueur de la lise en fonction du temps. Il s'agit d'un bloc de type **Accumulate**. Ce bloc n'existe que si on a déjà appelé **collectStat(true)** pour cette liste. Sinon un message d'erreur est affiché.

public void report ( )

Fournit un rapport statistique complet sur cette liste. On doit avoir auparavant appelé **collectStat(true)** pour cette liste.

paquetage kernel

UserRecord

public class UserRecord

Les objets de cette classe servent à mémoriser les informations des processus (nombre d'unité, priorité, etc) qui demandent des unités d'une ressource (**Resource**). Les objets de cette classe sont aussi utilisés dans le cas d'utilisation des mécanismes de type **Bin** ou **Condition**.

public UserRecord(int n, Process p);

Crée un dossier pour le processus **p** qui demande **n** unités (le nombre d'unités d'une ressource ou le nombre de jetons demandé pour un objet de type **Bin**). Par défaut, la priorité associée au processus est égale à 0.

public UserRecord(int n, Process p, double priority);

Similaire à **UserRecord(int n, Process p)** sauf que le processus demande les unités avec une priorité égale à **priority.**

public Process getUser();

Retourne le nom du processus pour lequel ce dossier a été ouvert.

public int getNbUnit();

Retourne le nombre d'unités demandé par le processus pour lequel ce dossier a été ouvert.

public double getTimeIn();

Retourne le temps d'entrée de processus (pour lequel ce dossier a été ouvert) dans la liste.

public double getPriority();

Retourne la priorité associée au processus référencé par ce dossier.

Paquetage statistics

StatProbe

public abstract class StatProbe

La classe abstraite **StateProbe** constitue la classe de base pour les autres classes servant à collecter des statistiques. Elle contient les attributs et les fonctionnalités communs aux classes statistiques. C'est une classe abstraite qui ne peut pas être instanciée. Les classes qui héritent de cette classe (classes **Tally** et **Accumulate**) implémentent les méthodes abstraites de cette classe.

abstract public void init ( )
Initialise le bloc statistique. Cette méthode doit être définie dans les sous classes.

abstract public void update (double x)
Donne une nouvelle observation au bloc statistique. Doit être définies dans les sous classes.

public double min ( )
Renvoie la plus petite valeur prise par la variable depuis la dernière initialisation du bloc statistique.

public double max ( )
Renvoie la plus grande valeur prise par la variable depuis la dernière initialisation du bloc statistique.

public double sum ( )
Renvoie la somme cumulée par le bloc statistique. Le sens de cette somme dépend de la sous-classe (**Tally** ou bien **Accumulate**).

abstract public double average ( )
Renvoie la moyenne pour ce collecteur.

abstract public void report ( )
Imprime le rapport sur ce bloc statistique.

paquetage statistics

Tally

public class Tally extends StatProbe

La classe **Tally** est instanciée quand on veut mesurer une séquence d'observations d'une variable, X1, X2,...

public Tally ( )
Construit un nouveau bloc statistique de type **Tally**.

public Tally (String name)
Construit un nouveau bloc statistique **Tally.** Le paramètre **name** sert à identifier ce bloc.

public void init ( )
Initialise le bloc statistique.

public void update (double x)
Donne une nouvelle observation **x** à ce bloc.

public int numberObs ( )
Renvoie le nombre des observations de ce collecteur statistique.

public double average ( )
Renvoie la valeur moyenne de toutes les observations.

public double variance ( )
Renvoie la variance des observations depuis la dernière initialisation. Si le nombre d'observation est inférieur à 2, un message d'erreur est affiché.

public double standardDev ( )
Renvoie l'écart-type des observations depuis la dernière initialisation.

public void confIntStudent (double level, double [] centerAndRadius)
Donne le centre et le rayon de l'intervalle de confiance à un niveau de confiance spécifié par **level** en faisant l'hypothèse que les observations suivent la loi normale.

public void printConfIntStudent (double level)
La même chose que **confIntStudent**, mais imprime l'intervalle de confiance au lieu de le retourner.

public void report ( )
Imprime un rapport sur ce bloc depuis la dernière initialisation.

paquetage statistics

Accumulate

public class Accumulate extends StatProbe

Cette classe est instanciée quand on s'intéresse à l'évolution dans le temps d'une variable { X(t), t>=0}.

public Accumulate ( )
Construit et initialise un nouveau bloc statistique de type **Accumulate**.

public Accumulate (String name)
Construit un nouveau bloc statistique de type **Accumulate.** Le paramètre **name** sert à identifier ce bloc.

public void init ( )
Initialise le collecteur statistique et met la valeur courante de la variable correspondante à 0. Un appel à cette méthode, normalement, doit être suivi immédiatement par un appel à la méthode **update**() pour donner une valeur à la variable au moment de l'initialisation.

public void init (double x)
Équivalant à un appel à **init()** suivi par **update(x)**.

public void update (double x)
Donne une nouvelle observation **x** au collecteur statistique.

public double average ( )
Renvoie la valeur moyenne de ce bloc par unité de temps depuis la dernière initialisation.

public report ( )
Imprime le rapport sur ce collecteur statistique depuis la dernière initialisation.

paquetage random

RandomStream

public interface RandomStream

Cette interface définit la structure de base et les outils nécessaires pour la gestion et l'utilisation des générateurs de nombres pseudo-aléatoires uniformes. Un générateur peut générer une longue séquence de nombres (pseudo)aléatoires. Cette séquence est découpée en plusieurs sous séquences disjointes et contiguës appelées «streams» qui peuvent être considérés comme des générateurs virtuels. Chaque «stream» est aussi découpé en plusieurs blocs appelés «substream». La longueur de ces «streams» et «substreams» dépend du type de générateur utilisé. L'état initial du générateur, est le point de départ du premier «stream» crée. Chaque fois que l'on crée un «stream» (un objet de type **RandomStream**), son point de départ est calculé automatiquement, plus loin que la position de départ du «stream» précédent de même type. On dénote par C_g l'état courant du «stream» g, I_g son état initial, B_g l'état au début du «substream» courant, et N_g l'état au début du prochain «substream».

Pour définir un nouveau type de générateur, l'utilisateur doit créer une classe implémentant l'interface **RandomStream**. Un générateur peut être utilisé directement par instanciation et peut aussi servir de base aux autres classes de génération de valeurs aléatoires suivant les lois de probabilité

public void resetStartStream ()

Réinitialise le «stream» à son état initial. ($\mathbf{C_g}$ et $\mathbf{B_g}$ sont initialisés à $\mathbf{I_g}$).

public void resetStartSubstream ()

Réinitialise le «stream» au début de son «substream» courant ($\mathbf{C_g}$ est initialisé à $\mathbf{B_g}$).

public void resetNextSubstream ()

Réinitialise le «stream» au début de son «substream» suivant. ($\mathbf{N_g}$ est calculé, et $\mathbf{C_g}$ et $\mathbf{B_g}$ sont initialisés à $\mathbf{N_g}$).

public void setAntithetic (boolean a)

Lorsque cette méthode est invoquée avec **a = true**, le «stream» commence à générer des variables antithétiques, c'est-à-dire **1-U** à la place de **U**, jusqu'à ce que cette méthode soit invoquée une deuxième fois avec le paramètre **a = false**.

public double randU01 ()

Retourne un nombre (pseudo)aléatoire selon la distribution uniforme continue dans l'intervalle (0,1) en utilisant ce «stream». L'état du «stream» est d'abord avancé par un pas.

public int randInt (int i, int j)

Retourne un nombre (pseudo)aléatoire selon la distribution discrète uniforme dans l'intervalle des entiers de i à j, en utilisant ce «stream». (Cette méthode appelle la méthode **randU01()**).

public void writeState ()

Imprime l'état courant du «stream».

Paquetage random

Mrg32k3

public class Mrg32k3 implements RandomStream

public Mrg32k3 ( )

Crée un nouveau «stream», initialise son état initial $\mathbf{I_g}$, initialise $\mathbf{B_g}$ et $\mathbf{C_g}$ égale à $\mathbf{I_g}$, et met son état antithétique à **false**. Si ce «stream» est le premier objet crée, l'état initial $\mathbf{I_g}$ est égal à l'état initial donné par la méthode **setPackageSeed()**. Sinon, l'état initial est calculé automatiquement plus loin que la position de départ du dernier «stream» crée de même type.

public Mrg32k3 (String name)

Similaire à **Mrg32k3()**, le paramètre **name** sert comme identificateur à cet objet (peut être utilisé lors de l'impression de l'état de ce «stream»).

public static void setPackageSeed (long seed [])

Met l'état initial de la classe **Mrg32k3** égal au vecteur des entiers indiqué par le paramètre **seed[]** qui va être l'état initial du premier « stream ». Ce vecteur doit avoir 6 éléments. Si cette méthode n'est pas appelée, l'état initial est égal à (12345, 12345, 12345, 12345, 12345). Lorsque cette méthode est appelée, les 3 premières valeurs du paramètre **seed** doivent être inférieures à m1 = 4294967087, et les 3 dernières valeurs doivent être inférieures à m2 = 4994944443, mais pas toutes égales à 0.

public void increasePrecis (Boolean incp)

Après avoir appelé cette méthode avec **incp = true**, chaque appel à une méthode **randU01**() (directement ou indirectement) retourne un nombre aléatoire uniforme avec 53 bits de précision au lieu de 32 bits. L'état du «stream» va être avancé par 2 pas au lieu de 1. Précisément, si **s** est une instance de la classe **Mrg32k3,** l'appel de **u = s.randU01**() équivaut à "**u = (s.randU01() + s.randU01() * fact)%1.0**" où la constante **fact** = 2^{-24} (ceci s'applique aussi lorsque l'on appel **randU01**() indirectement).

Par défaut, ou si cette méthode est invoquée une deuxième fois avec **incp = false**, chaque appel à **randU01()** avance l'état du "stream» par 1 pas et retourne un nombre avec 32 bits de précision.

public double[] getState ( )

Retourne l'état courant C_g de ce «stream». C'est un vecteur de 6 entiers représenté en format flottant.

public void advanceState (int e, int c)

Avance l'état du «stream» par k valeurs. Sans modifier les états des autres «streams». $\mathbf{B_g}$ et $\mathbf{I_g}$ associé à ce "stream» resteront aussi inchangés.
Si $e > 0$, alors $k = 2^e+c$; si $e < 0$, $k = -2^{-e}+c$; et si $e = 0$, alors $k = c$.

public void setSeed (long seed[])

Initialise l'état initial $\mathbf{I_g}$ de ce «stream» par le vecteur **seed[]** qui est composé de six éléments. Ce vecteur doit satisfaire aux même conditions que dans la méthode **setPackageSeed()**. Le «stream» est remet à son état initial. Les états courants et initiaux des autres «streams» ne sont pas changés.

public void writeStateFull ( )

Imprime le nom du "stream» et les valeurs de toutes ses variables internes.

Comme cette classe implémente l'interface **RandomStream**, elle doit implémenter toutes les méthodes de cette interface (voir l'interface **RandomStream** pour la définition de ces méthodes).

public void resetStartStream ( )

public void resetStartSubstream ( )

public void resetNextSubstream ( )

public void setAntithetic (Boolean a)

public void writeState ( )

public double randU01 ( )

public int randInt (int i, int j)

paquetage random

Variate

public class Variate

public Variate()
Construit un objet **Variate** qui référence un générateur de nombres (pseudo)aléatoires de type **Mrg32k3**.

public Variate(RandomStream stream)
Construit un objet **Variate** faisant référence au générateur de nombres (pseudo)aléatoires; nommé **stream**.

public RandomStream getStream()
Retourne le générateur de nombres (pseudo)aléatoires référencé par cet objet.

public void setStream(RandomStream stream)
À l'appel de cette méthode, l'objet **Variate** référence le générateur de nombres (pseudo)aléatoires nommé **stream**.

Les six méthodes suivantes manipulent le "stream» référencé par cet objet. Ces méthodes ne font qu'appeler les méthodes équivalentes du "stream» en question. (Voir la définition des méthodes dans l'interface **RandomStream**).

public void resetStartStream()

public void resetNextSubStream()

public void setAnthetic(boolean)

public void writeState()

public double randU01()

public int randInt(int, int)

paquetage random

Random1

public class Random1 extends Variate

Cette classe fournit quelques méthodes pour la génération des variables aléatoires selon différentes lois de probabilités. Toutes les méthodes utilisent l'inversion.

public Random1()
Construit un objet de type random1. Il se base sur un générateur de nombre aléatoire uniforme de type **Mrg32k3**.

public Random1(RandomStream stream)
Construit un objet de type Random1. Le paramètre **stream** sert comme un générateur de nombres aléatoires uniforme sur lequel se base l'objet crée.

public double uniform(double a, double b)
Renvoie une valeur aléatoire suivant la loi uniforme de paramètre **a** et **b.** La valeur produite est comprise strictement entre **a** et **b**.

public double expon (double mean)
Renvoie une valeur aléatoire suivant la loi exponentielle de moyenne μ = **mean.**

public double erlang (int k, double mu)
Renvoie une valeur aléatoire suivant la loi d'Erlang avec paramètres (k, mu), qui est la somme de k variables aléatoires indépendantes suivant la loi exponentielle de moyenne **mu**. La moyenne de cette variable est **k*mu**.

public weibull (double alpha, double lambda)
Renvoie une valeur aléatoire suivant la loi Weibull de paramètres α = **alpha** et λ= **lambda**. La fonction de répartition de cette loi est :

$$F(x) = 1 - e^{-\lambda x^{\alpha}} \quad \text{pour} \quad x > 0.$$

On doit avoir $\alpha > 0$ et $\lambda > 0$. Elle utilise l'inversion.

public double normal (double mean, double sdev)
Renvoie une valeur aléatoire suivant la loi normale de moyenne μ = **mean** et d'écart-type σ = **sdev**. Elle utilise l'inversion via la méthode **invNormalDist**().

public double invNormalDist(double u)
Retourne $y = F^{-1}(u)$, où F est la fonction de répartition de la loi normale de moyenne 0 et de variance 1. Elle utilise une approximation rationnelle donnant au moins 7 décimales de précision pour $10^{-20} < u < 1-10^{-20}$ (voir Kennedy [1980], page 85).

public double invStudentDist (int n, double u)

Renvoie $y= F^{-1}(u)$, où F est la fonction de répartition de la loi de Student à n degrés de liberté.

public double student (int n)

Renvoie une valeur aléatoire suivant la loi de Student avec **n** degrés de liberté. Elle utilise l'inversion et une approximation rationnelle.

Paquetage eventList

EventList

public interface EventList

L'interface **EventList** décrit le comportement général d'une liste d'événements. Toutes les classes implémentant cette interface doivent obligatoirement définir toutes les méthodes membres. Les objets référencés dans la liste d'événements sont de type **Event.**

public boolean isEmpty ()

Retourne **true** si et seulement si la liste est vide (aucun événement n'est planifié).

public void cleanup ()

Vide complètement la liste, c'est-à-dire annule tous les événements prévus.

public void insert (Event ev)

Insère l'événement **ev** dans la liste d'événements selon son instant d'occurrence (la valeur du champ **eventTime** de l'objet **ev**).

public void insertFirst (Event ev)

Insère l'événement **ev** en tête de la liste d'événements.

public void insertBefore (Event ev, Event other)

Insère l'événement **ev** immédiatement avant l'événement **other** dans la liste.

public void insertAfter (Event ev, Event other)

Insère l'événement **ev** immédiatement avant l'événement **other** dans la liste.

public Event viewFirst ()

Retourne le premier événement dans la liste d'événement.

public void print ()
Imprime le contenu de la liste d'événements.

public Event viewFirstOfClass (String cl)
Retourne le premier objet de type **cl** dans la liste (**cl** doit être d'une classe héritant de la classe **Event**).

public boolean remove (Event ev)
Retire l'événement **ev** de la liste d'événements. Retourne **true** si l'événement **ev** existe dans la liste, sinon **false** est retourné.

public Event removeFirst ()
Retire et retourne le premier événement de la liste.

paquetage eventList

DoublyLinked

public class DoublyLinked implements EventList

public DoublyLinked ( )
Construit une nouvelle liste linéaire doublement chaînée.

public Node findPosition (Event ev)
Cherche la position où **ev** peut être inséré dans la liste des événements.

Comme cette classe implémente l'interface **EventList**, elle doit implémenter toutes les méthodes de cette interface (Voir la l'interface **EventList** pour la définition de ces méthodes).

public boolean isEmpty ( )
public void print ( )
public void insert (Event ev)
public void insertFirst (Event ev)
public void insertBefore (Event ev, Event other)
public void insertAfter (Event ev, Event other)
public Event viewFirst ( )
public Event viewFirstOfClass (String cl)
public Boolean remove (Event ev)
public Event removeFirst ( )