SIMOD

Définition fonctionnelle et guide d'utilisation

Première version : août 1987
Version courante du guide : septembre 1993

Pierre L'Ecuyer[1]
Département d'I.R.O., Université de Montréal

Résumé

SIMOD est un progiciel de simulation de systèmes stochastiques à événements discrets, avec vision par événements ou par processus. Il supporte aussi la simulation continue, pour laquelle l'évolution de certaines variables du modèle se fait selon des équations différentielles. Le progiciel est implanté sous forme de modules pré-compilés, écrits en Modula-2. Ces modules comprennent des types de données prédéfinis, des fonctions, des procédures, et supportent une partie de l'exécution des programmes utilisateurs. Ces outils permettent d'exprimer un modèle de façon relativement concise, dans un langage lisible. Il ne s'agit pas d'un nouveau langage; en réalité, un programme SIMOD est un programme Modula-2 qui utilise les modules de SIMOD. Ceux qui connaissent déjà la syntaxe de Modula-2 (qui ressemble en fait beaucoup à celle de Pascal) n'ont donc pas de nouvelle syntaxe à apprendre pour utiliser SIMOD.

Le progiciel gère l'horloge et la liste des événements d'une simulation. Il fournit des outils pour gérer des listes, générer des valeurs pseudo-aléatoires selon différentes lois de probabilité et recueillir des statistiques durant la simulation. Les objets "actifs" dans le système peuvent être représentés par des processus, et SIMOD s'occupe de la synchronisation entre les processus, implantés sous forme de co-routines. Différentes formes de synchronisation entre les processus sont disponibles, telles que l'exclusion mutuelle par des ressources à capacité limitée, la relation de type producteur/consommateur par le biais d'un tampon (une pile de jetons), l'attente sur des conditions, et les relations de type maître/esclave.

Présentement, SIMOD est implanté sous VAX/VMS, MS-DOS, et SUN/OS.

[1]Denis Alain, Jean Bélanger, Michel Duclos, Nataly Giroux, Bruno Pelletier et Gaétan Perron ont participé au développement de SIMOD.

Table des matières

Avant-propos

La modélisation est une activité fondamentale, qui se situe au coeur de l'évolution de la science et de l'humanité. C'est en construisant des modèles que l'homme (au sens générique) peut mieux comprendre son environnement, anticiper l'avenir, et développer des stratégies pour réagir à cet environnement. Grâce à l'avènement d'ordinateurs de plus en plus puissants et d'outils logiciels de haut niveau, on peut développer et simuler sur ordinateur des modèles très complexes à des coûts de plus en plus bas. Mais malgré tout, parmi les langages spécialisés les plus populaires pour la simulation des systèmes stochastiques à événements discrets (tels que GPSS, SIMCRIPT II.5, SLAM, SIMAN,...), aucun n'est un langage de programmation moderne.

Ce guide décrit SIMOD, un progiciel servant à faciliter l'écriture des programmes de simulation de systèmes à événements discrets. SIMOD est basé sur Modula-2, un langage moderne et puissant malgré sa relative simplicité. Ainsi, l'utilisateur a accès à une "boîte à outils" bien garnie pour la simulation, tout en conservant les avantages d'utiliser un langage populaire et qui encourage l'application des principes modernes du génie logiciel. Ce progiciel a été implanté avec un certain souci d'efficacité, de façon à ce qu'il soit vraiment utilisable en pratique, même pour des applications de grande taille.

Je tiens à remercier tous ceux dont les commentaires et critiques ont permis d'améliorer SIMOD et/ou le présent guide, en particulier les étudiants du cours IFT-10557 de l'Université Laval, de même que Jules Desharnais, Marcel Dupras, Clermont Dupuis, Olivier Roux et Thien Vo-Dai. Je remercie également le CRSNG, le programme Défi 85 d'Emploi et Immigration Canada, de même que le programme d'innovation pédagogique 1986-87 de la faculté des sciences et de génie de l'Université Laval, dont les subventions ont permis de payer quelques-uns des étudiants qui ont travaillé à la conception et à l'implantation de SIMOD. La publication du présent document a été supportée en partie par le fonds FCAR.

L'utilisation de SIMOD exige un compilateur Modula-2 (voir les annexes B et C). Ce compilateur ne vient pas avec le progiciel SIMOD et c'est la responsabilité de l'utilisateur de se le procurer.

1 Introduction

La simulation sur ordinateur est un outil très utile aux preneurs de décisions. Il s'agit de modéliser un système et de reproduire son comportement sur ordinateur. Les modèles les plus réalistes sont habituellement *dynamiques* et *stochastiques*, c'est-à-dire que leur état évolue dans le temps et que cette évolution peut se faire de façon aléatoire. SIMOD permet de simuler des modèles de ce genre. Il a été conçu surtout pour la simulation à événements discrets [4, 5, 11, 12], mais permet aussi la simulation continue.

Dans la pratique, les programmes de simulation ont longtemps été (et sont encore souvent) écrits dans des langages tout-usage, tels que C, Pascal ou FORTRAN [12, 4]. Dans le but d'améliorer la productivité, on a développé, dans ces langages, des librairies de sous-routines fournissant des outils de base pour la programmation de simulation. On a aussi créé, par ailleurs, des langages spécialisés pour la simulation. Par exemple, les langages GPSS [26], SIMSCRIPT [24], SLAM [23] et SIMAN [22] ont été créés pour la simulation des systèmes à événements discrets. Plusieurs langages spécialisés ont aussi été créés pour la simulation continue.

Les langages spécialisés fournissent des outils de plus haut niveau et permettent d'écrire des programmes plus courts, mais n'offrent souvent que peu de flexibilité. Dans les langages de simulation les plus populaires, on doit souvent appeler des routines FORTRAN pour effectuer certaines opérations. Les compilateurs des langages de simulation sont en général plus coûteux et moins répandus que ceux des langages tout-usage, et produisent du code moins efficace. Autre obstacle important aux langages spécialisés: pour les utiliser, il faut les apprendre. Comme ces langages ont souvent une syntaxe plutôt excentrique, cela nécessite un investissement en temps non négligeable, surtout pour un usage unique ou occasionnel. Plusieurs logiciels spécialisés pour la simulation proposent aussi une approche dite "sans programmation", où l'usager spécifie son modèle en répondant à des questions ou en manipulant des objets graphiques à l'écran. Cette approche peut être intéressante pour développer rapidement des petits modèles relativement simples, mais devient inutilisable dès que l'on veut modéliser quelque chose qui sort du cadre (souvent simpliste) fixé par le logiciel.

Pour ces raisons, dans de nombreux projets de simulation, on utilise encore des langages tout-usage tels que FORTRAN. Mais FORTRAN est un langage plutôt dépassé (voir [11]). On lui préfère maintenant des langages plus riches de concepts et plus fiables, tels que Pascal, Modula-2, ADA, etc. Ces langages plus modernes sont mieux structurés, favorisent l'écriture d'un code plus lisible, offrent des structures de données plus variées et plus puissantes, permettent l'allocation dynamique de mémoire, permettent de diminuer les erreurs en assurant la compatibilité des types, etc. Pascal est un langage qui a fait sa marque à plusieurs points de vue, mais qui a aussi ses faiblesses, surtout pour l'implantation de gros logiciels. Il ne supporte pas, par exemple, les coroutines et la compilation séparée. L'implantation d'un progiciel de simulation basé sur Pascal se fait assez bien si on se limite à la vision par événements et si on utilise une extension du langage qui permet la compilation séparée [13]. Mais pour la vision par processus, on doit ou bien modifier le langage et réécrire le compilateur, ou bien écrire un préprocesseur [9, 11, 19, 30].

Le langage Modula-2, développé par Wirth, est souvent présenté comme le successeur de Pascal [7, 8, 32]. Il a été conçu pour combler les lacunes de Pascal, tout en essayant d'en conserver la simplicité (par opposition à ADA, par exemple). Modula-2 permet d'une part l'accès à des fonctions de bas niveau, près du langage machine, et encourage d'autre part une bonne utilisation des règles du génie logiciel. Des outils de haut niveau peuvent être implantés dans des modules précompilés (séparément), qui permettent d'effectuer des opérations sur des objets abstraits dont la structure est définie à l'intérieur de ces modules, mais dont l'implantation est cachée aux utilisateurs. Le langage offre des facilités de base pour l'utilisation de co-routines, et on peut utiliser ces facilités pour implanter différents paradigmes de synchronisation entre les processus. Modula-2 est disponible et supporté sur la plupart des machines. Pour quelqu'un qui connait Pascal, apprendre Modula-2 ne pose vraiment aucune difficulté et peut se faire en quelques heures.

SIMOD est un progiciel de simulation implanté sous forme d'un ensemble structuré de modules précompilés, en Modula-2. Ces modules fournissent des procédures et des types de données pré-définis, et contiennent un exécutif qui s'occupe de gérer l'horloge de la simulation, de gérer la liste d'événements, de synchroniser les processus, de recueillir des statistiques, etc. SIMOD supporte l'utilisation de la vision par processus: chaque objet actif dans le système peut être représenté par un processus, qui possède ses propres données (locales) et dont le comportement est décrit par une procédure (les processus de même type ont la même procédure et les mêmes types de données). Durant la simulation, ces processus interagissent. Il peut y avoir plusieurs types de processus dans un modèle, et plusieurs instances d'un même type de processus à un instant donné. Dans SIMOD, les processus sont implantés par des co-routines. La programmation avec vision par processus est reconnue comme étant une façon très naturelle de décrire un système complexe ([1, 3, 11, 29]). Les systèmes réels sont souvent constitués d'objets plus ou moins autonomes (tels que des personnes, des avions, des véhicules, des robots, etc.) qui interagissent pour accomplir leurs tâches, et on peut faire correspondre chaque type d'objet à un type de processus. Dans SIMOD, on peut aussi prévoir des événements et donc mélanger, si désiré, la vision par processus avec la vision par événements. Dans plusieurs cas, on peut même n'utiliser que des événements. En fait, il existe plusieurs types de systèmes qui se modélisent plus facilement et plus simplement de cette façon. De plus, l'utilisation des co-routines ralentit l'exécution de façon assez significative en pratique, et pour cette raison, on peut vouloir dans certain cas éviter leur utilisation. SIMOD permet aussi la simulation continue ou "mixte" (mélange de simulation continue et à événements discrets dans un même modèle). Dans ce cas, les valeurs de certaines variables évoluent selon des équations différentielles.

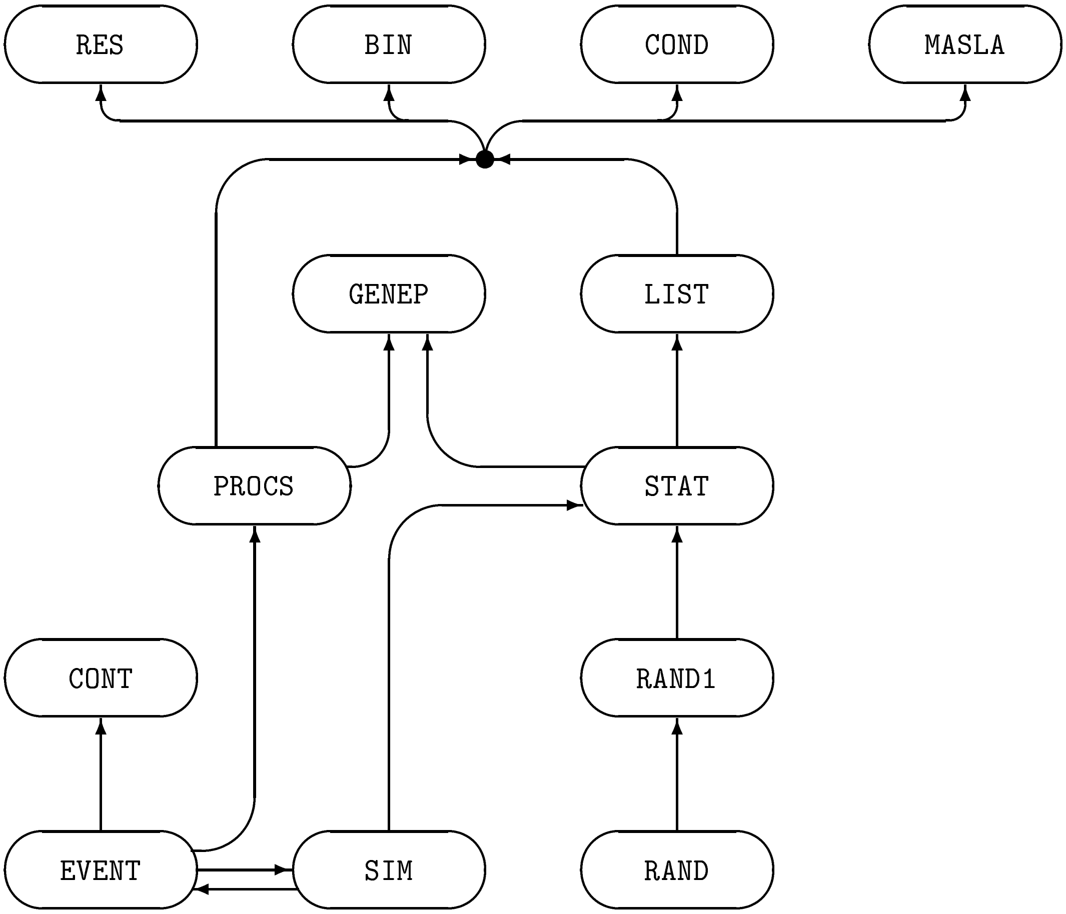
Un effort a été fait afin de bien séparer la définition fonctionnelle du progiciel (la partie visible à l'usager) de l'implantation proprement dite. L'usager peut utiliser les types de données et les routines fournies par le progiciel, mais n'a pas à accéder directement aux structures de données utilisées par le progiciel. Il n'a pas à connaître les détails d'implantation pour pouvoir utiliser le progiciel avec profit. Lorsqu'on a affaire à de gros modèles de simulation, SIMOD encourage l'idée de décomposer le modèle et d'en implanter les parties dans des modules compilés séparément. A la limite, on peut implanter indépendamment, dans différents modules, des programmes de simulation pour différents sous-systèmes, puis faire un petit module “chapeau” qui démarre une simulation intégrant les différents sous-systèmes (on peut même faire plusieurs “copies” d'un même sous-système).

Certaines des idées de SIMOD sont inspirées du progiciel DEMOS [2], basé sur le langage Simula [3], de SIMSCRIPT II.5 [24], et de SIMPascal [13]. D'autres progiciels de simulation à événements discrets, basés sur des langages qui supportent la quasi-concurrence, ont déjà été proposés. Sans prétendre être exhaustif, on peut mentionner des progiciels basés sur ADA [29], C [26], Simula [1, 2], SMALLTALK [10], SCHEME (un dialecte de Lisp) [27], PROLOG [31], etc. Quelques-uns sont aussi implantés en Modula-2 [18, 25], mais conçus différemment de SIMOD. SMALLTALK, SCHEME et PROLOG sont des langages très puissants et très intéressants pour développer des outils de prototypage exploratoire. Mais lorqu'il s'agit de faire de longues simulations afin d'obtenir des résultats statistiquement significatifs, ou encore d'appliquer certaines techniques d'approximation ou d'optimisation stochastique, ils deviennent pratiquement inutilisables. En effet, les implantations actuelles sont malheureusement très lentes, et il y a peu d'espoir que l'on arrive, dans un avenir rapproché, à des implantations qui produisent du code suffisamment efficace pour faire les expériences statistiques requises en simulation, sans sacrifier la puissance et la souplesse de ces langages. Simula est un langage qui semble en voie de disparition. C produit du code efficace, mais est de bas niveau et n'est pas très lisible. ADA est un langage très bien conçu, plus “propre” que Modula-2 à certains égards (voir section 4), mais complexe, de grande taille, et plus difficile d'accès (les compilateurs sont beaucoup plus coûteux que ceux de Modula-2). Il est aussi moins souple que Modula-2 sur certains points (pas de variables de type “PROCEDURE”, pas de type “ADDRESS”, etc.), et un peu moins bien adapté aux activités de prototypage. Nous ne prétendons pas que Modula-2 soit le langage parfait pour implanter un progiciel de simulation, mais il s'agit certainement d'un langage pratique, grâce auquel on peut dès maintenant implanter des outils à la fois attrayants et performants. SIMOD est présentement implanté sous VAX/VMS avec l'environnement Modula-2 de Logitech [6], sous MS-DOS dans l'envronnement *TopSpeed Modula-2* [28], et sous SUN/OS avec le compilateur Modula-2 de de SUN.

La prochaine section donne une vue d'ensemble de SIMOD. Dans la section 3, on présente une série d'exemples, d'abord simples puis plus complexes, dont l'objectif est d'illustrer l'utilisation de SIMOD. L'examen de ces exemples est probablement la meilleure façon de se familiariser avec le progiciel. On suggère au lecteur de référer à l'annexe A tout au long de la lecture de cette section. Dans la conclusion, on discute des aspects positifs et négatifs de notre expérience avec Modula-2 et avec SIMOD, et de quelques détails d'implantation. La définition fonctionnelle de chacun des modules est donnée à l'annexe A, et les modes d'utilisation sur *VAX/VMS* et MS-DOS sont donnés aux annexes B et C respectivement. Tout au long de ce guide, nous supposons que le lecteur est déjà familier avec le langage Modula-2. Pour une introduction à Modula-2, voir [7, 8, 32].

2 Une vue d'ensemble du progiciel

Nous allons maintenant décrire l'organisation de SIMOD et les principaux modules qui le constituent. La description détaillée de tous les outils disponibles se trouve en Annexe A. Le diagramme suivant illustre la structure hiérarchique qui existe entre les différents modules. Lorsqu'on peut se rendre du module `A` au module `B` en suivant une ou plusieurs flèches, cela indique que le module `B` peut utiliser des objets du module `A` dans sa définition ou dans son implantation. Par exemple, `RES` utilise `PROCS`, `EVENT`, `SIM`, `LIST`, `STAT`, `RAND1` et `RAND`. Ainsi, lorsqu'on utilise `RES`, on doit faire l'édition de liens avec tous ces modules. Le plupart des modules utilisent aussi les modules `NUM` et `MYINOUT`, qui ne font pas partie de SIMOD, mais qui servent à améliorer la portabilité.



Les modules accessibles à l'usager sont brièvement décrits ci-bas. Evidemment, outre les modules mentionnés ici, l'usager peut ajouter ses propres modules.

- `RAND` contient le générateur de valeurs pseudo-aléatoires. Son noyau est une adaptation du logiciel proposé par L'Ecuyer et Côté [14], basé sur le générateur 32-bits proposé par L'Ecuyer [16].
- `RAND1` fournit des fonctions pour générer des valeurs aléatoires suivant différentes lois de probabilité.
- `SIM` fournit une fonction qui retourne l'instant courant de la simulation, et des procédures pour initialiser, démarrer et stopper la simulation.
- `EVENT` offre des outils pour la prévision d'événements dans un délai donné, ou l'annulation d'événements déjà prévus.
- `CONT` fournit les outils de base pour la simulation continue, pour laquelle certaines variables évoluent selon des équations différentielles.
- `STAT` fournit des outils pour le recueil de statistiques.
- `PROCS` permet la programmation avec vision par processus. Chaque type de processus est associé à une procédure définie par l'usager, qui décrit le comportement de l'objet représenté par ce processus.
- `GENEP` permet de démarrer et stopper des mécanismes automatiques pour générer une suite d'événements ou de processus d'un certain type. Cette génération peut se faire selon un processus de renouvellement quelconque, et en particulier selon un processus de Poisson.
- `LIST` fournit un type qui correspond à une liste doublement chaînée, et un ensemble d'outils pour la gestion des listes. Les listes sont gérées sans connaître les types des objets qu'elles contiennent. En fait, ces objets sont vues par le logiciel comme s'ils étaient du type `ADDRESS`, qui est compatible avec tout type de pointeur. Ces listes peuvent donc contenir à peu près n'importe quoi. En pratique, on s'en sert surtout pour implanter les files d'attente. Le module offre aussi des outils pour le recueil automatique de statistiques sur les listes.
- `RES` permet la synchronisation des processus par le biais de ressources à capacité limitée. Une *ressource* correspond à un poste de service, avec une seule file d'attente, et dont la politique de service est fixée lors de la création. La capacité de la ressource peut correspondre, par exemple, au nombre de serveurs (identiques). Les processus doivent demander un nombre d'unités de la ressource avant de pouvoir l'utiliser, et libérer ce nombre d'unités lorsqu'ils en ont terminé. Un même processus peut détenir des unités de différentes ressources, ou des unités d'une même ressource demandées à des instants différents. Les requêtes peuvent aussi se faire avec des priorités. Enfin, le module peut recueillir (automatiquement) des statistiques sur les ressources.

- `BIN` permet les relations de type producteur/consommateur entre les processus. Un "bin" correspond essentiellement à une pile de jetons (un tampon) et une file d'attente de processus. Un *producteur* peut ajouter des jetons à la pile, et un (processus) *consommateur* peut demander à prendre des jetons de la pile. Dans ce dernier cas, s'il n'y a pas assez de jetons pour satisfaire la demande du consommateur, ce dernier est bloqué et placé dans la file d'attente pour cette pile. Il n'est réactivé que lorsque c'est à son tour de se faire servir et qu'un nombre suffisant de jetons est disponible.

- `COND` offre des outils permettant de gérer l'attente de processus sur des conditions booléennes. Lorsqu'un processus demande à attendre une *condition* et que cette dernière est fausse, celui-ci est bloqué jusqu'à ce que la condition devienne vraie.

- `MASLA` implante le paradigme de synchronisation de type maître/esclave entre les processus. Un point de synchronisation (ou point de rencontre) possède deux files d'attente: une pour les processus *maîtres* et une pour les processus *esclaves*. Un processus peut demander à devenir un maître (et obtenir un esclave) pour un point de rencontre donné. Si aucun esclave n'est disponible, ce processus est bloqué et placé dans une file d'attente jusqu'à ce qu'il puisse obtenir l'esclave demandé. Un processus peut aussi devenir un esclave pour un point de rencontre donné. Dans ce cas, si la file d'attente des maîtres est vide, ce processus est bloqué et placé dans la file d'attente des esclaves, sinon il réveille le premier maître de la file et devient son esclave.

3 Exemples de programmes SIMOD

Nous présentons dans cette section quelques exemples de programmes SIMOD. L'objectif est de familiariser le lecteur avec le progiciel et d'illustrer son utilisation. Le premier exemple est classique et très simple, il s'agit d'une file d'attente à plusieurs serveurs. Il illustre l'utilisation simultanée d'événements et de processus. Les deux exemples qui suivent sont inspirés de [12]. Il s'agit de programmes de simulation d'un atelier de production et d'un système à temps partagé. Le quatrième exemple est une simulation de visites guidées, et illustre une synchronisation des processus légèrement plus complexe. Le cinquième illustre l'utilisation de SIMOD pour la simulation continue, et le sixième illustre une simulation mixte et plus complexe. Le septième et dernier exemple montre comment SIMOD permet le développement modulaire des programmes de simulation. Il s'agit d'un robot qui, dans une cellule de production d'une usine, entretient une série de machines. La description du système, de la politique de commande, et de l'expérience statistique que l'on veut faire avec le simulateur, sont implantés dans trois modules séparés. Ainsi, on peut modifier par exemple la politique de commande sans changer quoi que ce soit aux deux modules qui décrivent le système et l'expérience statistique (et sans recompiler ces modules). Eventuellement, on pourrait remplacer le module qui décrit le système par un mécanisme de recueil de données et de commande en temps réel, branché sur le vrai système. De plus, le robot et la série de machines peuvent être vus comme un sous-système, dans un système plus vaste. Lorsqu'on veut simuler le système en entier, on peut vouloir définir plusieurs "copies" du sous-système, possiblement avec des paramètres différents, et qui évolueront simultanément durant la simulation. Ces sous-systèmes peuvent même interagir directement ou indirectement par le biais d'autres sous-systèmes. Dans cet exemple, nous illustrons comment SIMOD permet de développer des "simulateurs" pour différents types de sous-systèmes, puis d'intégrer ces "simulateurs", de façon modulaire, afin de former de plus gros sous-systèmes, etc.

En examinant ces exemples, le lecteur pourra se référer au besoin à la définition fonctionnelle des objets fournis par les modules de SIMOD, à l'annexe A.

3.1 Une file d'attente $M/U/s$

Considérons une file d'attente à s serveurs. Les clients arrivent selon un processus de Poisson et les durées de service sont des variables aléatoires de loi uniforme. Un programme SIMOD simulant un tel système est donné à la figure 1. Pour simplifier, cette version du programme ne lit aucune variable en entrée et les valeurs des paramètres du modèle apparaissent directement dans le programme. Evidemment, en pratique (surtout dans le cas des programmes réutilisés plusieurs fois), ces valeurs seront habituellement lues et placées dans des variables.

Les énoncés `IMPORT` permettent d'importer des objets (types et procédures) des modules de SIMOD. A remarquer qu'on peut importer explicitement des objets spécifiques, ou encore importer un module en entier pour utiliser l'un ou plusieurs des objets qui s'y trouvent. Dans ce dernier cas, les identificateurs des objets utilisés doivent être "qualifiés" (préfixés du nom de leur module) lors de leur utilisation. C'est au programmeur à décider du mode

```
MODULE Queue;

IMPORT SIM, EVENT, PROCS, RES, GENEP;
FROM RES    IMPORT Resource, Request, Release, Report, ServicePolicy;
FROM PROCS  IMPORT Delay, Terminate;
FROM RAND1  IMPORT Uniform;

VAR
   Serveur : Resource;
   Client  : PROCS.ProcessType;
   FinSim  : EVENT.EventType;
   Gener   : GENEP.ArrivalProcess;

PROCEDURE ProcClient;
   BEGIN
   Request (1, Serveur);
   Delay (Uniform (12.0, 16.0, 2));
   Release (1, Serveur);
   Terminate;
   END ProcClient;

PROCEDURE ProcFinSim;
   BEGIN
   Report (Serveur);
   SIM.Stop;
   END ProcFinSim;

BEGIN
EVENT.Create  (FinSim,     ProcFinSim,       'Fin de la simulation');
PROCS.Create  (Client,     ProcClient, 2000, 'Un client');
RES.Create    (Serveur,    Fifo, 3,          'Poste de service');
RES.CollectStat (Serveur);
GENEP.Create (Gener, "Processus d'arrivee");
SIM.Init;
EVENT.Schedule (FinSim, 1000.0, NIL);
GENEP.StartPoissonArrivalsP (Gener, Client, NIL, 10.0, 1, 0);
SIM.Start
END Queue.
```

Figure 1: Simulation d'une file d'attente $M/U/3$.

d'importation pour chacun des objets importés. Son choix devra viser à faciliter la lisibilité du programme. Une contrainte doit cependant être respectée: tous les identificateurs importés explicitement doivent être uniques les uns par rapport aux autres, et par rapport aux identificateurs déclarés localement. Par exemple, l'identificateur `Create`, disponible dans plusieurs modules de SIMOD, doit être qualifié lorsqu'il est importé de plus d'un module.

Dans cet exemple, chaque client est vu comme un processus, dont les activités sont décrites par la procédure `ProcClient`. Un client arrive, demande un serveur (`Request`), attend si aucun serveur n'est disponible, occupe le serveur pendant un certain temps (`Delay`), libère le serveur (`Release`), puis termine ses activités (`Terminate`). Lorsque le client appelle `Request` pour demander un serveur, il obtient un serveur et poursuit son exécution s'il y en a un de libre, sinon il est (automatiquement) placé dans une file d'attente et ne poursuit son exécution que lorsqu'il obtient le serveur demandé. En appelant `Delay`, le processus se bloque et doit attendre que l'horloge de la simulation avance d'un certain délai avant de pouvoir poursuivre son exécution. Ici, ce délai correspond à sa durée de service, et est généré aléatoirement (pour chaque client) selon une loi uniforme (continue) entre 12 et 16 unités de temps, et en utilisant le générateur de valeurs pseudo-aléatoires numéro 2.

La procédure `ProcFinSim` décrit un événement qui marque la fin de la simulation. Cette procédure fait imprimer un rapport statistique complet sur l'utilisation de la ressource (le serveur), puis arrête la simulation.

Dans le programme principal, on crée le type d'événement `FinSim`, associé à la procédure `ProcFinSim`, et le type de processus `Client`, associé à la procédure `ProcClient`. Chaque processus `Client` aura un espace-mémoire de 2000 octets pour son exécution. Pour chaque type de processus, on doit ainsi déclarer l'espace-mémoire requis, et cet espace doit être suffisant pour les variables locales de la procédure associée et de toutes les procédures qu'elle appelle directement ou indirectement. La variable `Serveur` a été déclarée comme représentant un type de ressource. La procédure `RES.Create` crée cette ressource, qui correspond ici au groupe de serveurs. La politique de service est `Fifo` (premier arrivé, premier servi), et le nombre de serveurs est de 3. La chaîne de caractères "Poste de service" sera utilisée par le progiciel pour identifier cette ressource dans les rapports statistiques. L'appel à `RES.CollectStat` sert à demander un recueil automatique de statistiques pour cette ressource. `SIM.Init` initialise le simulateur. Cela doit être fait avant toute prévision d'événement ou de processus. On prévoit ensuite l'événement `FinSim` dans 1000 unités de temps, puis on démarre le processus d'arrivée des clients. `Gener` est un processus d'arrivée, qui fera arriver les clients. La procédure `StartPoissonArrivalsP` démarre ce processus. Les clients arrivent selon un processus de Poisson de taux 0.1. Ainsi, les intervalles de temps entre les arrivées successives, de même que le temps jusqu'à la première arrivée, sont des variables aléatoires exponentielles de moyenne 10 (unités de temps). Leurs valeurs sont générées en utilisant le générateur numéro 1. Le dernier paramètre de `StartPoissonArrivalsP` indique habituellement le nombre maximum de processus à générer. Ici, sa valeur est 0, ce qui indique que le processus de Poisson générant les arrivées ne s'arrêtera que si on appelle `StopArrivals (Gener)`, ou que si la simulation se termine. L'appel à `SIM.Start` démarre la simulation, qui s'arrêtera après 1000 unités de temps pour imprimer le rapport apparaissant à la figure 2.

Dans ce rapport, on voit que la capacité de la ressource est toujours demeurée à 3, le nombre moyen de serveurs utilisés fut de 1.705, et la longueur moyenne de la file d'attente fut de 0.144. Lorsque la simulation s'est arrêtée, 123 clients avaient débuté leur service, et 121 étaient sortis du système (2 étaient en train de se faire servir, et la file d'attente était donc vide). Les temps moyens d'attente dans la file, de service, et de séjour dans le système (attente + service) furent 1.171, 13.953 et 15.133 respectivement. A noter que la première valeur est une moyenne sur 123 observations, tandis que les deux autres sont des moyennes sur 121 observations, et c'est ce qui explique que la somme des deux premières valeurs n'est pas exactement égale à la troisième valeur. Le rapport nous fournit également les écarts-types échantillonnaux de ces observations. Cependant, il ne faut pas oublier qu'il s'agit d'observations *dépendantes*, de sorte que ces écarts-types ne peuvent pas être utilisés pour calculer des intervalles de confiance en utilisant les formules habituelles.

```
REPORT ON RESOURCE : Poste de service
   From time  0.00   to time   1000.00
                 min        max      average  standard dev.  nb. Obs.
   Capacity       3          3         3.000
   Utilization    0          3         1.704
   Queue          0          3         0.144
   Wait          0.000     13.298      1.171      2.851          123
   Service      12.055     15.982     13.952      1.141          121
   Sojourn      12.055     28.607     15.132      3.115          121
```

Figure 2: Résultats de la simulation de la file $M/U/3$.

3.2 Un atelier de production

Cet exemple est inspiré de Law et Kelton [12], section 2.6, et de [13]. Un atelier de production contient M groupes de machines. Pour $m = 1, \ldots, M$, le groupe m est constitué de s_m machines identiques. L'atelier est modélisé par un réseau de files d'attente. A chaque groupe de machines, est associée une file d'attente FIFO à plusieurs serveurs identiques (s_m serveurs pour le groupe m). Il y a N types de tâches qui arrivent à l'atelier, et pour $n = 1, \ldots, N$, les tâches de type n arrivent selon un processus de Poisson de taux λ_n (la durée entre deux arrivées successives de tâches de type n est une variable aléatoire de loi exponentielle de moyenne $1/\lambda_n$).

Chaque type de tâche requiert une séquence d'opérations bien déterminée, chaque opération devant être effectuée sur un type de machine bien précis. Les durées des opérations sont déterministes. Une tâche de type n requiert p_n opérations, qui doivent être effectuées dans l'ordre sur les machines $m_{n,1}$, $m_{n,2}$, $\ldots m_{n,p_n}$, et dont les durées respectives sont $d_{n,1}$, $d_{n,2}$, $\ldots d_{n,p_n}$, en heures. A noter que rien n'empêche une tâche de repasser plusieurs fois sur un même type de machine, et il n'y a donc pas de limite supérieure sur la valeur de p_n.

On veut simuler le fonctionnement de l'atelier pendant T heures. On suppose que l'atelier est initialement vide, et on commencera à recueillir les statistiques après une période de réchauffement de T_R heures. On veut estimer : (a) le temps moyen passé dans le système pour chaque type de tâche; (b) le taux moyen d'utilisation, le temps moyen d'attente, la longueur moyenne de la file d'attente, etc. pour chaque type de machine.

Le programme SIMOD de la figure 3 permet d'effectuer cette simulation. Chaque groupe de machines est vu comme une ressource, dont la capacité est égale au nombre de machines (identiques) dans le groupe. Toutes les files d'attentes sont de type FIFO. Le programme utilise un dossier (un "RECORD") pour chaque type de tâche, afin de conserver l'information associée: le taux d'arrivée, le nombre d'opérations à effectuer, les durées de ces opérations et les groupes de machines sur lesquelles on doit les effectuer, de même qu'un bloc statistique servant à recueillir des statistiques sur les temps passés dans le système pour les tâches de ce type. Le chiffre 5 qui apparaît dans la déclaration des types correspond à une borne supérieure sur les nombres de groupes de machines, de types de tâches, et d'opérations par tâche. La procédure `LireDonnees` lit les données dans un fichier dont le nom est fourni interactivement par l'usager, et crée les types de machines et les types de tâches.

Chaque tâche dans le système est un processus dont la vie est décrite par la procédure `ProcTache`. Les tâches sont générées par des processus de Poisson, un par type de tâche, lesquels sont démarrés à partir du programme principal. Le second paramètre de `StartPoissonArrivalsP` est le type de tâche à générer, et correspondra à la valeur prise par l'attribut du processus tâche. La valeur de cet attribut est récupérée à l'intérieur de la procédure `ProcTache` en appelant la fonction `Attrib`. Pour chaque opération à effectuer, la tâche demande une machine du type approprié, se l'approprie pour la durée de l'opération, puis la libère. Lorsque la tâche se termine, le temps qu'elle a passé dans l'atelier constitue

```
MODULE Atelier;
IMPORT SIM, EVENT, PROCS, GENEP, STAT, RES;
FROM Storage IMPORT ALLOCATE;
FROM RES     IMPORT Resource, Request, Release, ServicePolicy;
FROM MYINOUT IMPORT OpenInput, CloseInput, ReadCard, ReadString, ReadLn;
FROM NUM     IMPORT LONGREAL, ReadLR;

TYPE
   NumTypMachine = CARDINAL [1..5]; (* Un numéro de groupe de machines.    *)
   NumTypTache   = CARDINAL [1..5]; (* Un numéro de type de tâche.         *)
   NumOper       = CARDINAL [1..5]; (* Un numéro d'opération.              *)
   InfoTypTache = RECORD            (* Information sur un type de tâche.   *)
      TauxArrivee  : LONGREAL;      (* Taux d'arrivée.                     *)
      DureesSejour : STAT.Block;    (* Stats. sur les durées de séjour.    *)
      NOper        : CARDINAL;      (* Nb. d'opérations à effectuer.       *)
      DureeOper    : ARRAY NumOper OF LONGREAL;
                                    (* Durées des opérations.              *)
      MachOper     : ARRAY NumOper OF Resource;
      END;                          (* Machines requises pour ces opér.    *)
VAR
   M,                               (* Nombre de groupes de machines.      *)
   N,                               (* Nombre de types de tâches.          *)
   m,                               (* Indice d'un groupe de machines.     *)
   j : CARDINAL;                    (* Indice d'un type de tâche.          *)
   TypMachine  : ARRAY NumTypMachine OF Resource;
                                    (* Tableau des groupes de machines.    *)
   TypTache    : ARRAY NumTypTache OF POINTER TO InfoTypTache;
                                    (* Tableau des types de tâches.        *)
   ArrivTaches : ARRAY NumTypTache OF GENEP.ArrivalProcess;
   Tache            : PROCS.ProcessType;
   FinRechauf, FinSim : EVENT.EventType;
   DureeRechauf, DureeSim : LONGREAL;

PROCEDURE LireDonnees;
   VAR
      Nom      : ARRAY [0..31] OF CHAR;
      Capacite : CARDINAL;          (* Nb. de machines dans un groupe.     *)
      Oper     : NumOper;
   BEGIN
   OpenInput (".DAT");              (* Fichier d'entrée.                   *)
   ReadLR (DureeRechauf);  ReadLR (DureeSim);
   ReadCard (M);  ReadCard (N);  ReadLn;
   FOR m:=1 TO M DO                 (* Données sur les machines.           *)
      ReadString (Nom);  ReadCard (Capacite);  ReadLn;
      RES.Create (TypMachine [m], Fifo, Capacite, Nom);
      RES.CollectStat (TypMachine [m])
      END;
   FOR j := 1 TO N DO               (* Données sur les tâches.             *)
      NEW (TypTache [j]);
      WITH TypTache [j]^ DO
         ReadString (Nom);  ReadLR (TauxArrivee);  ReadCard (NOper);
         FOR Oper := 1 TO NOper DO
            ReadCard (m);  ReadLR (DureeOper [Oper]);
            MachOper [Oper] := TypMachine [m]
            END;
         STAT.Create (DureesSejour, STAT.Tally, Nom);  ReadLn
         END
      END;
   CloseInput
   END LireDonnees;
```

Figure 3: Simulation d'un atelier de production.

```
PROCEDURE ProcTache;
   VAR
      TempsArrivee : LONGREAL;      (* Instant d'arrivée de cette tâche.   *)
      Typ  : POINTER TO InfoTypTache; (* Type de cette tâche.              *)
      Oper : NumOper;               (* Numéro de l'opération courante.     *)
   BEGIN
   TempsArrivee := SIM.Time ();
   Typ          := PROCS.Attrib ();
   WITH Typ^ DO
      FOR Oper := 1 TO NOper DO
         Request (1, MachOper [Oper]);
         PROCS.Delay (DureeOper [Oper]);
         Release (1, MachOper [Oper])
         END;
      STAT.Update (DureesSejour, SIM.Time () - TempsArrivee)
      END;
   PROCS.Terminate
   END ProcTache;

PROCEDURE ProcFinRechauf;
   BEGIN
   FOR m:=1 TO M  DO  RES.InitStat (TypMachine [m])  END;
   FOR j:=1 TO N  DO  STAT.Init (TypTache [j]^.DureesSejour)  END
   END ProcFinRechauf;

PROCEDURE ProcFinSim;
   BEGIN
   FOR m:=1 TO M  DO  RES.Report (TypMachine [m])  END;
   FOR j:=1 TO N  DO  STAT.Report (TypTache [j]^.DureesSejour)  END;
   SIM.Stop
   END ProcFinSim;

BEGIN
EVENT.Create (FinSim, ProcFinSim, 'Fin de la simulation');
EVENT.Create (FinRechauf, ProcFinRechauf, 'Fin du rechauffement');
PROCS.Create (Tache, ProcTache, 2000, 'Une tache');
LireDonnees;
EVENT.Schedule (FinSim, DureeSim, NIL);
EVENT.Schedule (FinRechauf, DureeRechauf, NIL);
FOR j := 1 TO N DO
   GENEP.Create (ArrivTaches [j], " ");
   GENEP.StartPoissonArrivalsP (ArrivTaches [j], Tache, TypTache [j],
                             1.0 / (TypTache [j]^.TauxArrivee), 1, 0)
   END;
SIM.Start
END Atelier.
```

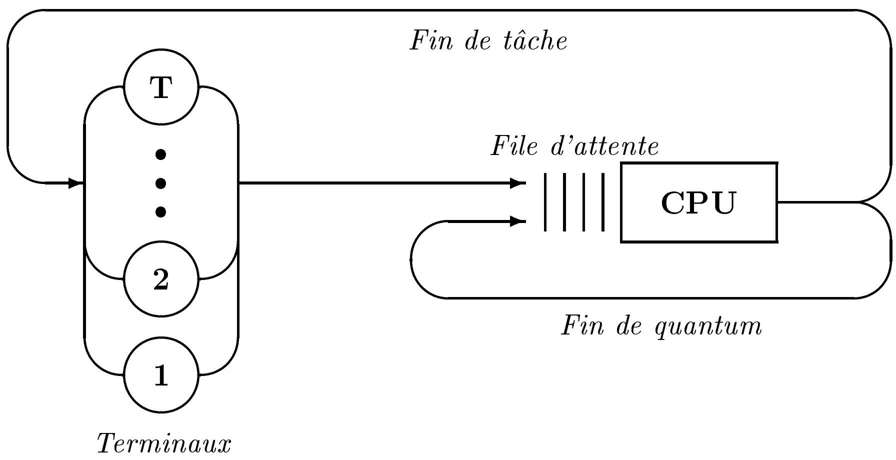
Figure 3: Simulation d'un atelier de production (suite).

une nouvelle observation pour le bloc statistique `DureesSejour`, qui recueille des statistiques sur les durées de séjour pour les tâches de ce type.

Avant de démarrer la simulation, dans le programme principal, deux événements sont prévus, l'un marquant la fin de la période de réchauffement et l'autre marquant la fin de la simulation. Les procédures `ProcFinRechauf` et `ProcFinSim`, respectivement, s'exécutent lors de l'occurence de ces événements. La première réinitialise tous les blocs statistiques, tandis que la seconde fournit les rapports statistiques demandés et arrête la simulation.

3.3 Un système à temps partagé

Cet exemple est inspiré de [12], section 2.4. Considérons un système informatique à temps partagé, simplifié. Le système est constitué de T terminaux indépendants et identiques, tous occupés, qui utilisent la même unité centrale (CPU). L'occupant d'un terminal soumet une tâche au CPU, attend la réponse, puis réfléchit pendant une durée aléatoire avant de soumettre une nouvelle tâche, etc. Le temps de réflexion est une variable aléatoire de loi exponentielle de moyenne μ, tandis que le temps de CPU requis par 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 pour le CPU, et sont servies selon une politique de "round-robin" avec quantum. Lorsqu'une tâche s'empare du CPU, s'il lui reste moins de q secondes à exécuter (la valeur du quantum), elle conserve le CPU jusqu'à la fin de son exécution. Sinon, elle s'exécute pendant q secondes et retourne à la fin de la file. Dans les deux cas, il faut aussi h secondes supplémentaires ("overhead") pour l'enlever du CPU 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 CPU lorsqu'elle a terminé son exécution. On s'intéresse au temps de réponse moyen, à l'état stationnaire. On peut simuler 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 se seront terminées. De plus, on effectue R répétitions indépendantes de cette simulation.

Supposons que l'on veut comparer deux configurations du système, du point de vue du temps de réponse moyen. On définit ici une configuration du système comme étant caractérisée par $(T, q, h, \mu, \alpha, \lambda)$. Lorsque l'on veut comparer ainsi deux systèmes ou deux configurations, on y gagne en général à utiliser des techniques de réduction de variance, dont l'une des plus simples et des plus efficaces consiste à utiliser des valeurs 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 i-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 i-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

```
MODULE TempsPartage;

IMPORT SIM, STAT, PROCS, RES;
FROM RAND    IMPORT InitGenerator, SeedType;
FROM RAND1   IMPORT Weibull, Expon;
FROM RES     IMPORT Resource, Request, Release, ServicePolicy;
FROM MYINOUT IMPORT OpenInput, OpenOutput, CloseInput, CloseOutput, ReadCard;
FROM NUM     IMPORT LONGREAL, ReadLR;
VAR
   TotTaches     : CARDINAL;        (* Nombre de fins de tâches avant     *)
                                    (* la fin de la simulation.           *)
   PrelimTaches  : CARDINAL;        (* Nombre de fins de tâches avant     *)
                                    (* la fin du réchauffement.           *)
   NTaches       : CARDINAL;        (* Nombre de tâches déjà terminées.   *)
   NTer          : CARDINAL;        (* Nombre de terminaux.               *)
   Quantum       : LONGREAL;        (* Durée d'un quantum.                *)
   Overhead      : LONGREAL;        (* Durée de l'overhead.               *)
   TReflexMoy    : LONGREAL;        (* Temps moyen de réflexion.          *)
   Alpha, Lambda : LONGREAL;        (* Par. de la loi Weibull du temps    *)
                                    (* de CPU par tâche.                  *)
   CPU       : Resource;            (* Le CPU.                            *)
   Terminal  : PROCS.ProcessType;   (* Un terminal.                       *)
   NRep, Rep : CARDINAL;            (* Nb. de répét. et numéro de la      *)
                                    (* répét. courante.                   *)
   MoyConf1  : ARRAY [1..100] OF LONGREAL; (* Temps de réponse moy. des   *)
                                    (* répétitions pour la config. 1.     *)
   TRepMoy   : STAT.Block;          (* Stat. sur les temps de réponse     *)
                                    (* pour la répétition courante.       *)
   Diff      : STAT.Block;          (* Stat. sur les différences entre    *)
                                    (* les temps de réponse.              *)
   Conf      : [1..2];              (* No. de la configation courante.    *)
   i         : CARDINAL;

PROCEDURE LireDonnees;
   BEGIN
   ReadCard (NTer);        ReadLR (Quantum);   ReadLR (Overhead);
   ReadLR (TReflexMoy);    ReadLR (Alpha);     ReadLR (Lambda);
   ReadCard (TotTaches);   ReadCard (PrelimTaches);  ReadCard (NRep)
   END LireDonnees;
```

Figure 4: Simulation d'un système à temps partagé.

```
PROCEDURE ProcTerminal;
   (* Décrit la suite des activités reliées à un terminal.                 *)
   VAR
      TArrivee    : LONGREAL;       (* Instant d'arrivée de la dernière    *)
                                    (* tâche de ce terminal.               *)
      TCPURestant : LONGREAL;       (* Temps cpu encore requis.            *)
   BEGIN
   WHILE (NTaches <= TotTaches) DO
      PROCS.Delay (Expon (TReflexMoy, 1));  (* Temps de réflexion.         *)
      TArrivee := SIM.Time();       (* Arrivée d'une tâche.                *)
      TCPURestant := Weibull (Alpha, Lambda, 2); (* Durée de la tâche.     *)
      WHILE (TCPURestant > Quantum) DO
         Request (1, CPU);          (* Il reste plus d'un quantum.         *)
         PROCS.Delay (Quantum + Overhead);
         TCPURestant := TCPURestant - Quantum;
         Release (1, CPU)
         END;
      Request (1, CPU);             (* Il reste une fraction de quantum.   *)
      PROCS.Delay (TCPURestant + Overhead);
      Release (1, CPU);             (* Fin de la tâche.                    *)
      STAT.Update (TRepMoy, SIM.Time () - TArrivee);
      IF (NTaches = PrelimTaches) THEN
         STAT.Init (TRepMoy)  END;  (* Fin du réchauffement.               *)
      INC (NTaches)
      END;
   SIM.Stop;  PROCS.Terminate       (* Ici, on a  NTaches = TotTaches.     *)
   END ProcTerminal;

PROCEDURE SimulerUneRepetition;
   BEGIN
   PROCS.KillAll;  SIM.Init;  STAT.Init (TRepMoy);  RES.Init (CPU);
   NTaches := 0;
   FOR i := 1 TO NTer DO            (* Activer les terminaux.              *)
      PROCS.Schedule (Terminal, 0.0, NIL)  END;
   SIM.Start
   END SimulerUneRepetition;

BEGIN
PROCS.Create (Terminal, ProcTerminal, 1000, 'Un terminal');
RES.Create   (CPU, Fifo, 1, 'Le CPU');
STAT.Create  (TRepMoy, STAT.Tally, 'Temps reponse moyen');
STAT.Create  (Diff, STAT.Tally, 'Differences');
OpenInput (".DAT");  OpenOutput (".RES");
FOR Conf := 1 TO 2 DO
   LireDonnees;
   IF Conf = 2 THEN
      InitGenerator (1, InitialSeed);  InitGenerator (2, InitialSeed)  END;
   FOR Rep := 1 TO NRep DO
      SimulerUneRepetition;
      IF Conf = 1 THEN
         MoyConf1 [Rep] := STAT.Average (TRepMoy)
      ELSE
         STAT.Update (Diff, MoyConf1 [Rep] - STAT.Average (TRepMoy))  END;
      InitGenerator (1, NewSeed);   InitGenerator (2, NewSeed)
      END
   END;
STAT.ReportConfidenceInterval (Diff, 0.9);
CloseInput;  CloseOutput
END TempsPartage.
```

Figure 4: Simulation d'un système à temps partagé (suite).

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 SIMOD de la figure 4 permet d'effectuer cette simulation. La variable `Conf` désigne le numéro de la configuration courante. Le tableau `MoyConf1` sert à mémoriser les valeurs des R_{1i}, et le bloc statistique `Diff` sert à recueillir les statistiques sur les D_i et à calculer un intervalle de confiance pour d.

Pour chaque répétition, le bloc statistique `TRepMoy` cumule les statistiques sur les temps de réponse des $N - N_0$ tâches qui se terminent durant la période de recueil des statistiques. On l'initialise au début de la répétition et y effectue une mise à jour chaque fois que l'une de ces tâches se termine. Entre une répétition et sa suivante, on doit initialiser à zéro le nombre de tâches terminées, réinitialiser la simulation et le bloc statistique `TRepMoy`, et fournir des nouveaux germes aux deux générateurs utilisés, afin de s'assurer que d'une configuration à l'autre, les répétitions correspondantes se fassent toutes à partir des mêmes germes et utilisent des valeurs aléatoires communes. Avant de passer à la seconde configuration, les générateurs sont réinitialisés à leurs germes initiaux.

Au début de chaque répétition, on active aussi un processus pour chaque terminal. Lorsque la N-ième tâche se termine, le processus correspondant à cette tâche appelle `SIM.Stop`, ce qui arrête la simulation et retourne le contrôle au programme principal, juste après l'appel à `SimulerUneRepetition`. Lorsqu'on réinitialise la simulation, tous les processus de la répétition précédente (s'il y a lieu) sont détruits.

```
REPORT ON STATISTIC (Tally) : Difference
                  min        max      average  standard dev.  nb. obs.
                  0.138      0.293     0.177      0.045         10
            90% confidence interval for mean : (     0.151,      0.203 )
```

Figure 5: Simulation d'un système à temps partagé.

Soient $T = 20$, $h = .001$, $\mu = 5$ sec., $\alpha = 1/2$ et $\lambda = 1$ pour les deux configurations. Pour ces valeurs, la moyenne de la loi Weibull est de 2. On choisit $q = 0.1$ pour la configuration 1 et $q = 0.2$ pour la configuration 2. On choisit aussi $N_0 = 100$, $N = 1100$ et $R = 10$ répétitions. On obtient les résultats de la figure 5. L'intervalle de confiance ne contenant pas zéro, on peut en conclure que les temps de réponse moyens avec $q = 0.2$ sont significativement plus courts qu'avec $q = 0.1$.

A noter qu'on pourrait rendre le modèle plus réaliste en considérant par exemple différents "types" de terminaux avec des paramètres différents, un nombre de terminaux variable en fonction du temps, différentes classes de tâches avec des priorités, des demandes d'entrées/sorties, etc. Le programme deviendrait plus élaboré, mais la structure de base serait similaire. SIMOD permet très bien de simuler ce genre de modèle complexe.

3.4 Des visites guidées

On offre des visites guidées d'un site touristique. Trois hôtesses travaillent à cet endroit pour guider les visiteurs. Le site ouvre à 10 heures et ferme à 16 heures. Les groupes de clients arrivent selon un processus de Poisson à partir de 9:45 heures jusqu'à 16 heures, au taux d'un groupe par 3 minutes. Le nombre de visiteurs dans un groupe est une variable aléatoire X qui prend la valeur i avec une probabilité p_i donnée au tableau suivant:

i	1	2	3	4
p_i	.2	.6	.1	.1

Les visiteurs qui arrivent avant 10 heures doivent attendre l'ouverture. Après 16 heures, on termine les visites déjà amorcées, mais aucune nouvelle visite n'est entreprise. Ainsi, les visiteurs qui sont encore dans la file d'attente après 16 heures doivent s'en retourner.

```
MODULE Visites;

IMPORT SIM, STAT, PROCS, LIST, BIN, EVENT;
FROM NUM     IMPORT LONGREAL;
FROM RAND    IMPORT Uniform01;
FROM RAND1   IMPORT DiscreteDist, CreateDiscreteDist, Discrete, Expon;
FROM BIN     IMPORT Bin, Take, Give, Avail, WaitList, ServicePolicy;
FROM PROCS   IMPORT ProcessType, Delay;
FROM EVENT   IMPORT EventType;

VAR
   TailleQ  : CARDINAL;            (* Taille de la file d'attente.       *)
   NbPerdus : CARDINAL;            (* Nb. de visiteurs perdus pour la    *)
                                   (* Répétition courante.               *)
   VisiteOK : Bin;                 (* Contient un jeton ssi il y a assez *)
                                   (* de monde pour démarrer une visite. *)
   TailleArriv : DiscreteDist;     (* Fct. de répartition de la taille   *)
                                   (* groupes qui arrivent.              *)
   Hotesse    : ProcessType;       (* Une hôtesse.                       *)
   GenArrivee : ProcessType;       (* Générateur d'arrivées.             *)
   Fermeture  : EventType;         (* La fermeture du site.              *)
   StatPertes : STAT.Block;        (* Stat. sur les pertes pour les      *)
                                   (* différentes répétitions.           *)
   Rep  : CARDINAL;                (* No. de la répét. courante.         *)
   i    : CARDINAL;
   V, F : ARRAY [0..3] OF LONGREAL; (* Valeurs possibles de la taille X    *)
                                   (* d'un groupe et fct de répart. de X. *)

PROCEDURE ProcFermeture;
   BEGIN
   IF LIST.Size (WaitList (VisiteOK)) = 0 THEN (* Aucune hôtesse de libre. *)
      NbPerdus := NbPerdus + TailleQ
      END;
   SIM.Stop
   END ProcFermeture;
```

Figure 6: Simulation de visites guidées.

```
PROCEDURE ProcHotesse;
   VAR
      Lunch : BOOLEAN;
   BEGIN
   Lunch := FALSE;                  (* N'a pas encore mangé.            *)
   WHILE TRUE DO
      IF (NOT Lunch) AND (SIM.Time () > 12.0) THEN  (* Aller manger.      *)
         Delay (0.5);  Lunch := TRUE  END;
      Take (1, VisiteOK);
      IF TailleQ > 15 THEN
         TailleQ := TailleQ - 15
      ELSE
         TailleQ := 0  END;
      IF (TailleQ >= 8) THEN        (* Il y a encore assez de monde.    *)
         Give (1, VisiteOK)  END;
      Delay (0.75)                  (* Durée de la visite.              *)
      END
   END ProcHotesse;

PROCEDURE ProcGenArrivee;
   BEGIN
   WHILE TRUE DO
      Delay (Expon (0.05, 1));
      IF ((TailleQ > 10)
            AND (Uniform01 (2) < (VAL (LONGREAL, TailleQ - 10)) / 30.0)
            OR (TailleQ >= 40)) THEN (* Ces visiteurs renoncent.         *)
         NbPerdus := NbPerdus + TRUNC (Discrete (TailleArriv, 3))
      ELSE
         TailleQ := TailleQ + TRUNC (Discrete (TailleArriv, 3));
         IF (TailleQ >= 8) AND (Avail (VisiteOK) = 0) THEN
            Give (1, VisiteOK)  END  (* Il y a maintenant assez de monde. *)
         END
      END
   END ProcGenArrivee;

PROCEDURE SimulerUneRepetition;
   BEGIN
   BIN.Init (VisiteOK);  PROCS.KillAll;  SIM.Init;
   TailleQ := 0;  NbPerdus := 0;
   EVENT.Schedule (Fermeture, 16.0, NIL);
   PROCS.Schedule (GenArrivee, 9.75, NIL);
   FOR i := 1 TO 3 DO  PROCS.Schedule (Hotesse, 10.0, NIL)  END;
   SIM.Start;
   STAT.Update (StatPertes, VAL (LONGREAL, NbPerdus))
   END SimulerUneRepetition;

BEGIN
EVENT.Create (Fermeture, ProcFermeture, 'Fermeture du site');
PROCS.Create (GenArrivee, ProcGenArrivee, 1000, "Generateur d'arrivees");
PROCS.Create (Hotesse, ProcHotesse, 2000, 'Une hotesse');
BIN.Create   (VisiteOK, Fifo, 'Groupe');
STAT.Create  (StatPertes, STAT.Tally, 'Les clients perdus');
FOR i := 0 TO 3 DO  V[i] := VAL (LONGREAL, i + 1)  END;
F[0] := 0.2;  F[1] := 0.8;  F[2] := 0.9;  F[3] := 1.0;
CreateDiscreteDist (TailleArriv, 4, V, F);

FOR Rep := 1 TO 10 DO  SimulerUneRepetition  END;
STAT.ReportConfidenceInterval (StatPertes, 0.9)
END Visites.
```

Figure 6: Simulation de visites guidées (suite).

Une visite dure 45 minutes et nécessite une hôtesse. Une hôtesse ne peut pas faire visiter plus de 15 personnes à la fois, et n'amorcera une visite que s'il y a au moins 8 personnes en attente (sauf à 16 heures, où s'il y a une hôtesse de libre, celle-ci amorcera une visite avec le dernier groupe de personnes même s'ils sont moins que 8). A midi, chaque hôtesse libre prend 30 minutes pour manger. Les hôtesses qui sont occupées à ce moment prendront 30 minutes pour manger dès qu'elles termineront leur visite en cours. Lorsqu'une hôtesse se libère (termine une visite sans devoir aller manger, revient de manger, ou débute sa journée), s'il y a moins de 8 personnes en attente, elle attend, sinon elle entreprend une visite. S'il y a plus de 15 personnes en attente, 15 d'entre elles iront avec l'hôtesse.

Les personnes qui veulent visiter forment une seule file d'attente pour les hôtesses. Parfois, lorsqu'un groupe arrive et qu'il y a trop de monde dans la file d'attente, celui-ci renonce à attendre et s'en va. La probabilité d'une telle renonciation dépend du nombre n de personnes déjà dans la file, et est donnée par:

$$R(n) = \begin{cases} 0 & \text{pour } n \leq 10; \\ (n-10)/30 & \text{pour } 10 < n < 40; \\ 1 & \text{pour } n \geq 40. \end{cases}$$

On veut estimer le nombre de visiteurs perdus par jour, en moyenne. Les visiteurs perdus au cours d'une journée sont ceux qui renoncent à attendre, et ceux qui sont encore dans la file après 16 heures.

Le programme SIMOD apparaît à la figure 6. Pour simplifier les entrées/sorties, les paramètres du modèle ont été placés directement dans le programme. En pratique, ces valeurs seront habituellement lues dans un fichier et placées dans des variables. Le temps est mesuré en heures et l'instant 0.0 correspond à minuit. Ainsi, par exemple, 9:45 heures correspond à l'instant 9.75, et 30 minutes correspondent à 0.5 unités de temps. Le processus `Hotesse` correspond à une hôtesse. Le processus `GenArrivee` génère les arrivées des groupes de visiteurs, et l'événement `Fermeture` correspond à la fermeture du site, à 16 heures. Le générateur d'arrivées utilise le "`Bin`" `VisiteOK` (qui contient toujours 0 ou 1 jeton), afin de pouvoir "réveiller" une hôtesse lorsque le nombre de visiteurs en attente devient suffisamment élevé pour démarrer une visite. Les résultats de la simulation apparaissent à la figure 7.

A remarquer que nous aurions pu générer les arrivées des groupes de façon différente, en considérant chaque arrivée comme un événement, et en utilisant la procédure `StartPoissonArrivalsE` du module `GENEP`. On pourrait aussi penser utiliser une `Condition` du module `COND` à la place du `Bin`. La `Condition` deviendrait vraie lorsqu'il y aurait au moins 8 personnes en attente. Malheureusement, cela poserait une difficulté importante, car en mettant la `Condition` à vrai, on réveillerait *toutes* les hôtesses en attente, et non pas seulement la première.

```
REPORT ON STATISTIC (Tally) : Les clients perdus
                  min        max      average  standard dev.  nb. obs.
                  8.000     47.000     22.800     12.743          10
            90% confidence interval for mean : (    15.412,     30.187 )
```

Figure 7: Résultats de la simulation des visites guidées.

3.5 Simulation continue: un système proies–prédateurs

Considérons 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:

$$\begin{aligned} x'(t) &= rx(t) - cx(t)z(t) \\ z'(t) &= -sz(t) + dx(t)z(t) \end{aligned}$$

avec pour valeurs initiales $x(0) = x_0 > 0$ et $z(0) = z_0 > 0$. A remarquer ici que nous approximons 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 du module `CONT` de SIMOD, nous allons le résoudre numériquement. Cela est fait dans le programme de la figure 8.

Ce programme lit les paramètres du modèle dans un fichier d'extension "`.DAT`", et imprime la trajectoire de $(t, x(t), z(t))$ dans un fichier d'extension "`.RES`". Les noms de ces fichiers sont fournis au terminal par l'usager. Le second fichier pourra être traité par la suite, par exemple pour tracer le graphique de cette trajectoire. Les procédures `Dx` et `Dz` retournent la dérivée en fonction du temps pour les valeurs de `x` et `z` respectivement. L'événement `UnPas` s'exécute à chaque pas d'intégration, et permet d'imprimer la trajectoire. Cet événement n'est associé qu'à la variable `x`, et non pas à `x` et `z`, car on veut qu'il ne soit exécuté qu'une seule fois à chaque pas d'intégration. On utilise ici la méthode de Runge Kutta d'ordre 4.

```
MODULE ProiePred;

IMPORT SIM, EVENT, CONT;
FROM NUM    IMPORT ReadLR, WriteLR, LONGREAL;
FROM InOut  IMPORT WriteLn, OpenInput, OpenOutput, CloseInput, CloseOutput;
FROM SYSTEM IMPORT ADDRESS;
FROM EVENT  IMPORT EventType;
FROM CONT   IMPORT VarCont, Value, SelectMethod, Method, StartInteg;

VAR
   r, c, s, d, x0, z0 : LONGREAL;   (* Paramètres du modèle.              *)
   x, z   : VarCont;                (* Nb. de proies et de prédateurs.    *)
   Duree  : LONGREAL;               (* Durée de la simulation.            *)
   FinSim : EventType;              (* Fin de la simulation.              *)
   UnPas  : EventType;              (* Evénement exécuté à chaque pas.    *)

PROCEDURE ProcFinSim;
   BEGIN  SIM.Stop  END ProcFinSim;

PROCEDURE ProcUnPas;
   BEGIN
   WriteLR (SIM.Time (), 12);
   WriteLR (Value (x), 12);  WriteLR (Value (z), 12);  WriteLn;
   END ProcUnPas;

PROCEDURE Dx (t : LONGREAL) : LONGREAL;
   BEGIN
   RETURN  r * Value (x) - c * Value (x) * Value (z)
   END Dx;

PROCEDURE Dz (t : LONGREAL) : LONGREAL;
   BEGIN
   RETURN  - s * Value (z) + d * Value (x) * Value (z)
   END Dz;

BEGIN
OpenInput (".DAT");  OpenOutput (".RES");
ReadLR (r);   ReadLR (c);   ReadLR (s);   ReadLR (d);
ReadLR (x0);  ReadLR (z0);  ReadLR (Duree);
EVENT.Create (UnPas, ProcUnPas, "Un pas d'integration");
EVENT.Create (FinSim, ProcFinSim, "Fin de la simulation");
CONT.Create (x, x0, Dx, NIL);   CONT.Create (z, z0, Dz, NIL);
SelectMethod (RungeKutta4, 5.0);
StartInteg (x, UnPas, NIL);   StartInteg (z, EventType (NIL), NIL);
EVENT.Schedule (FinSim, Duree, NIL);
SIM.Start;
CloseInput;   CloseOutput
END ProiePred.
```

Figure 8: Simulation d'un système proies-prédateurs.

3.6 Simulation discrète et continue: chauffage de lingots

Cet exemple est inspiré de [22], page 210. Dans une aciérie, des lingots d'acier arrivent à une fournaise de trempage pour être chauffés avant de poursuivre leur route vers l'étape suivante du processus. Les lingots arrivent selon un processus de Poisson, et le temps moyen entre les arrivées est de a heures. Il y a de la place pour N lingots dans la fournaise. Lorsqu'un lingot arrive à la fournaise, on l'y insère s'il reste de la place. Sinon, le lingot est placé dans une zone d'attente pour les lingots froids, et attend qu'il y ait de la place dans la fournaise. La température initiale d'un lingot qui arrive est une variable aléatoire qui suit à peu près une loi normale de moyenne μ_T et d'écart-type σ_T (en degrés celsius). On suppose cependant que tous les lingots placés dans la zone d'attente ont une température égale à τ_T degrés celsius lorsqu'ils sont insérés dans la fournaise.

La variation de température des lingots lorsqu'ils sont chauffés par la fournaise est décrite par:

$$\frac{dT_j}{dt} = (T - T_j)h_j$$

où T_j est la température du lingot j et h_j est le coefficient de chauffage du lingot j. Pour chaque lingot, le coefficient h_j est la valeur d'une variable aléatoire normale de moyenne μ_C et d'écart-type σ_C. Les lingots sont chauffés dans la fournaise jusqu'à ce que leur température atteigne S degrés celsius. A ce moment, ils sont retirés du four. La fournaise est chauffée de sorte que sa température T évolue dans le temps selon l'équation différentielle:

$$\frac{dT}{dt} = (K - T)c - \sum_{j \in \Gamma} (T - T_j) b h_j,$$

où b et c sont des constantes, et Γ est l'ensemble des lingots qui sont dans le four.

Le programme SIMOD donné à la figure 3.6 permet de simuler un tel système. Il combine à la fois la simulation à événements discrets, avec vision par événements et par processus, et la simulation continue. Encore une fois, pour simplifier, les valeurs des paramètres du modèle ont été placées directement dans le programme. En pratique, il est évidemment recommandé de généraliser le programme en plaçant ces valeurs dans des variables. Ici, nous avons utilisé les valeurs suivantes: $N = 5$, $a = 15.0$, $\mu_T = 230$, $\sigma_T = 12$, $\tau_T = 200$, $K = 1425$, $\mu_C = 0.15$, $\sigma_C = 0.01$, $b = 0.5$, $c = 0.2$, $S = 1200$. La température initiale de la fournaise est de 600 degrés celsius et cette dernière ne contient aucun lingot. Le système est simulé pour 1000 heures d'opération, et le programme produit le rapport de la figure 3.6 sur l'utilisation de la fournaise.

Le premier lingot n'arrive qu'après une durée exponentielle à partir du début de la simulation, mais on commence à chauffer la fournaise dès le début de la simulation. L'événement `FinSim` marque la fin de la simulation. Les places dans le four sont vues comme une ressource, de capacité N, et pour laquelle on demande un recueil automatique des statistiques. Pour la partie continue, les équations différentielles sont intégrées numériquement en utilisant une méthode de Runge-Kutta d'ordre 2. Il y a une variable continue qui correspond à la température du four, et à la température de chaque lingot. Chaque lingot est représenté par un

```
MODULE Lingots;

IMPORT SIM, EVENT, STAT, PROCS, LIST, RES, CONT;
FROM NUM     IMPORT LONGREAL;
FROM SYSTEM  IMPORT ADDRESS;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM RAND1   IMPORT Normal, Expon;
FROM EVENT   IMPORT EventType, EventNotice, NoticeAttrib;
FROM CONT    IMPORT VarCont, Value, VarAttrib, CurrentVar, SelectMethod,
                    Method, StartInteg, Delete;
FROM PROCS   IMPORT ProcessType, ProcessInstance, Delay, Suspend, Resume,
                    InstanceAttrib, CurrentProcess, Terminate;
FROM LIST    IMPORT List, View, OutPosition;
FROM RES     IMPORT Resource, Request, Release, ServList, User, UserDossier,
                    Avail, ServicePolicy, Report;
TYPE
   Lingot = POINTER TO RECORD       (* Un lingot.                          *)
      Proc : ProcessInstance;       (* Le processus associé à ce lingot.   *)
      Tj   : VarCont;               (* La température du lingot.           *)
      hj   : LONGREAL               (* Coefficient de réchauffement.       *)
      END;
VAR
   PlaceDansFour : Resource;        (* Les places dans le four.            *)
   UnLingot      : ProcessType;     (* Décrit le comportement d'un lingot. *)
   GenArrivees   : ProcessType;     (* Génère les arrivées des lingots.    *)
   FinSim        : EventType;       (* Fin de la simulation.               *)
   TestS         : EventType;       (* Vérifie la température d'un lingot. *)
   TemperFour    : VarCont;         (* La température du four.             *)

PROCEDURE ProcLingot;
   VAR
      L : Lingot;
   BEGIN
   L := PROCS.Attrib ();
   WITH L^ DO
      IF Avail (PlaceDansFour) = 0 THEN  CONT.Init (Tj, 200.0)  END;
      Request (1, PlaceDansFour);
      StartInteg (Tj, TestS, L);   Suspend;
      (* Le lingot vient maintenant d'atteindre la temperature S.          *)
      Release (1, PlaceDansFour);   Delete (Tj);   DISPOSE (L);   Terminate
      END
   END ProcLingot;

PROCEDURE ProcTestS;
   VAR
      L : Lingot;
   BEGIN
   L := EVENT.Attrib ();
   IF Value (L^.Tj) >= 1200.0 THEN  Resume (L^.Proc)  END
   END ProcTestS;

PROCEDURE DerivLingot (t : LONGREAL) : LONGREAL;
   VAR
      L : Lingot;
   BEGIN
   L := VarAttrib (CurrentVar ());
   RETURN  L^.hj * (Value (TemperFour) - Value (CurrentVar ()))
   END DerivLingot;
```

Figure 9: Chauffage de lingots dans un four.

```
PROCEDURE DerivFour (t : LONGREAL) : LONGREAL;
   VAR
      L    : Lingot;
      Sum  : LONGREAL;
      U    : ADDRESS;
      Serv : List;
   BEGIN
   Serv := ServList (PlaceDansFour);
   Sum := 0.2 * (1425.0 - Value (TemperFour));
   View (U, First, Serv);
   WHILE U # NIL DO
      L := InstanceAttrib (User (UserDossier (U)));
      Sum := Sum - 0.5 * L^.hj * (Value (TemperFour) - Value (L^.Tj));
      View (U, Next, Serv)
      END;
   RETURN Sum
   END DerivFour;

PROCEDURE ProcFinSim;
   BEGIN   Report (PlaceDansFour);   SIM.Stop   END ProcFinSim;

PROCEDURE ProcGenArrivees;
   VAR
      L : Lingot;
   BEGIN
   WHILE TRUE DO
      Delay (Expon (15.0, 1));  NEW (L);
      WITH L^ DO
         hj := Normal (0.15, 0.01, 3);
         CONT.Create (Tj, Normal (230.0, 12.0, 2), DerivLingot, L);
         PROCS.ScheduleNamedNext (UnLingot, L, Proc)
         END
      END
   END ProcGenArrivees;

BEGIN
RES.Create (PlaceDansFour, Fifo, 5, "Places dans le four");
RES.CollectStat (PlaceDansFour);
PROCS.Create (GenArrivees, ProcGenArrivees, 2000, "Generateur d'arrivees");
PROCS.Create (UnLingot, ProcLingot, 1000, "Un lingot");
EVENT.Create (FinSim, ProcFinSim, "Fin de la simulation");
EVENT.Create (TestS, ProcTestS, "Test si temper. >= S");
CONT.Create  (TemperFour, 600.0, DerivFour, NIL);

EVENT.Schedule (FinSim, 1000.0, NIL);
PROCS.Schedule (GenArrivees, 0.0, NIL);
SelectMethod (RungeKutta2, 0.01);
StartInteg (TemperFour, EventType (NIL), NIL);
SIM.Start
END Lingots.
```

Figure 9: Chauffage de lingots dans un four (suite).

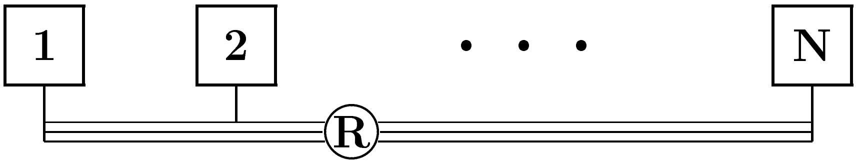
pointeur sur un enregistrement contenant son coefficient h_j, la variable continue qui décrit sa température T_j, et le processus associé. Le comportement de ce processus est décrit par la procédure `ProcLingot`, et les lingots sont générés par le processus `GenArrivees`, dont le comportement est décrit dans `ProcGenArrivees`. Le paramètre (attribut) associé à chaque "processus" lingot lors de sa création est en fait le lingot lui-même. Il en est de même pour la variable continue qui décrit sa température. A noter que les variables `hj` et `Tj` ne peuvent pas être simplement des variables locales de la procédure `ProcLingot`, car si tel était le cas, on ne pourrait pas leur accéder de l'extérieur. Au début de l'exécution de sa procédure, le lingot récupère le pointeur sur son enregistrement, et le place dans `L`. Il regarde ensuite si le four est plein, auquel cas il réinitialise sa température à τ_T. Puis, il demande une place dans le four. Dès qu'il obtient cette place, il démarre le processus d'intégration qui fera évoluer sa température, et se place dans l'état `Suspended`. A chaque pas d'intégration, un événement de type `TestS` sera exécuté, avec pour paramètre ce lingot. En exécutant cet événement, on vérifie si la température du lingot a atteint le seuil S, et si oui, on réactive le lingot. Lors de cette réactivation, le lingot libère sa place dans le four, détruit la variable continue qui lui est associée (afin d'en arrêter l'intégration et d'en récupérer l'espace), puis disparaît.

```
REPORT ON RESOURCE : Places dans le four
   From time  0.00 to time    1000.00
                min        max       average  standard dev.  nb. Obs.
   Capacity       5          5         5.000
   Utilization    0          5         2.326
   Queue Size     0          3         0.131
   Wait           0.000     19.714     1.694      4.724          77
   Service       14.609     54.224    30.201     10.567          77
   Sojourn       14.609     69.214    31.895     12.924          77
```

Figure 10: Statistiques sur l'utilisation du four.

3.7 Entretien d'un groupe de machines par un robot

Description du modèle. On retrouve le genre de système décrit ici dans certaines usines de produits textiles [17]. Soit un groupe de N machines identiques placées en ligne droite, équidistantes, numérotées $1, 2, \ldots, N$, de gauche à droite. Chaque machine fonctionne pour une durée aléatoire, puis se bloque, attend que l'on vienne la réparer, se fait réparer, fonctionne pour une autre période de durée aléatoire, etc. Un robot patrouille la ligne pour réparer les machines qui se bloquent. La distance entre deux machines voisines est d'une unité, et le robot voyage à vitesse constante, de sorte qu'il lui faut $|j - i|$ unités de temps pour voyager de la machine i à la machine j sans s'arrêter. Le robot peut changer de direction n'importe où, même entre les machines, et il peut aussi s'arrêter en face d'une machine. Lorsqu'il se trouve en face d'une machine bloquée, il peut décider de la réparer et, dans ce cas, il ne peut pas se déplacer tant que la réparation n'est pas terminée. Il arrive parfois que le robot ne réussisse pas à compléter lui-même une réparation, et doive faire appel à un réparateur pour la compléter manuellement. Dans ce cas, le robot demeure en face de la machine brisée jusqu'à ce que le réparateur ait terminé la réparation. La durée d'une réparation par le robot, celle d'une réparation manuelle par le réparateur, et le temps entre la fin d'une réparation et le prochain bloquage, sont des variables aléatoires indépendantes dont les lois de probabilité sont connues.



Les décisions concernant les mouvements du robot sont prises en suivant une *politique de commande*, c'est-à-dire une règle qui, dépendant de l'état actuel du système, détermine ce que le robot doit faire. Le robot peut prendre une décision lorsqu'il arrive à destination, ou lors d'une fin de réparation, ou lorsqu'une machine se bloque pendant qu'il n'est pas en train d'effectuer une réparation.

L'état du système, tel qu'observé par le robot, est un couple (p, E), où $p \in [1, N]$ indique la position du robot sur la ligne, et $E \subseteq \{1, \ldots, N\}$ est l'ensemble des machines qui sont en panne. Chaque décision d du robot est un élément de l'ensemble $D = \{0, 1, \ldots, N\}$. Le choix de $d \in \{1, \ldots, N\}$ indique que le robot se déplace en direction de la position d. Si une machine se bloque avant qu'il atteigne cette position, il reconsidère alors sa décision, sinon il doit prendre une nouvelle décision lorsqu'il atteint la position d (c'est-à-dire dans $|p - d|$ unités de temps). Si $d = p$, le robot s'arrête et attend la prochaine panne. Si $d = 0$, il décide de réparer la machine qui se trouve devant lui. Cette dernière décision n'est admissible que lorsque le robot se trouve devant une machine bloquée.

On veut estimer le taux de production moyen du système, c'est-à-dire le nombre moyen de machines qui fonctionnent, en fonction du temps, sur horizon infini. La politique de commande du robot n'est pas fixée d'avance. On pourrait par exemple utiliser la simulation pour comparer plusieurs dizaines de politiques différentes sur un même système.

Structure du programme. Dans ce cas-ci, nous avons choisi d'implanter le programme de simulation sous forme de plusieurs modules compilés séparément. Les modules `GroupeMach`, `Commande` et `Experience` implantent respectivement la description des lois de comportement d'un groupe de machines avec un robot, différentes politiques de commande du robot, et une expérience statistique que l'on peut faire avec ce système. Les parties "définitions" des deux premiers modules apparaissent aux figures 11 et 12. Ces définitions servent d'interfaces entre l'intérieur et l'extérieur de ces modules. Les objets qui apparaissent dans la définition du module `GroupMach` sont utilisés par le module `Experience` pour initialiser le système et la position du robot, et par le module `Commande` pour connaître l'état du système lors de la prise des décisions. L'implantation du module `GroupMach` utilise aussi les procédures du module `Commande` pour déterminer les actions du robot. Un exemple de module `Experience` apparaît à la figure 13. L'implantation de `GroupMach` est donnée à la figure 16. A noter qu'il n'est pas nécessaire de connaître ou d'avoir accès à cette implantation pour écrire les modules `Experience` et `Commande`, de la même façon qu'on n'a pas besoin d'examiner l'implantation des modules de SIMOD pour les utiliser. De même, il n'est pas nécessaire de connaître l'implantation du module `Commande` pour pouvoir l'utiliser. Cette implantation est donnée à la figure 17.

Dans la définition de `GroupMach`, un groupe de machines avec un robot est défini comme un pointeur sur un bloc d'information contenant toutes les caractéristiques de ce groupe. Cela peut permettre éventuellement de traiter plusieurs groupes de machines simultanément, chaque groupe possédant ses propres caractéristiques. (Nous y reviendrons plus loin.) La procédure `Create` permet d'initialiser (créer) un groupe, tandis que `Delete` permet de le détruire. Nous avons opté pour un maximum de flexibilité: le nombre de machines, la position initiale du robot, la règle de décision, le réparateur utilisé et toutes les lois de probabilité sont passés en paramètres et peuvent varier d'un groupe à l'autre. Tel que le module est implanté, l'usager a directement accès à tous les champs du "RECORD" associé à un groupe de machines. Cependant, il ne devrait pas les modifier directement (sauf en appelant la procédure `Create`), mais seulement *observer* leurs valeurs. Pour plus de sécurité, nous aurions pu adopter une plus grande "abstraction des donnés", selon les principes modernes du génie logiciel, en définissant le type `Groupe` comme un type opaque. La définition de `InfoGroupe` n'aurait été donnée que dans l'implantation du module, et on aurait fourni quelques procédures permettant d'observer quelques-uns de ses champs. On peut noter, d'ailleurs, que les modules de SIMOD sont implantés selon une telle approche. Par contre, malgré ses dangers, l'approche de l'accès direct aux champs du "RECORD" permet de réduire la longueur du code et (un peu) les temps d'exécution. Le programmeur utilisant le module doit alors être suffisamment discipliné pour éviter les manoeuvres dangereuses.

Le module `Commande` implante quelques politiques de commande du robot. La politique `BalayageComplet` consiste à patrouiller la ligne de gauche à droite jusqu'à la machine N,

```
DEFINITION MODULE GroupMach;
IMPORT EVENT, PROCS, STAT, RES;
FROM NUM IMPORT LONGREAL;

CONST
   Nmax    = 30;                (* Nombre maximum de machines permis.     *)
TYPE
   Num     = [1..Nmax];         (* Numéro de machine.                     *)
   ExtNum  = [0..Nmax];
   EnsMach = SET OF ExtNum;     (* Un ensemble de machines.               *)
   Direction = (Gauche, Droite);
   Groupe  = POINTER TO InfoGroupe; (* Un groupe de machines avec un robot.*)
   Politique = PROCEDURE (Groupe) : ExtNum;
                                (* Une politique de commande du robot.    *)
                                (* Retourne 0 : repare mach. devant lui;  *)
                                (* Retourne 0 < i <= N : va vers i.       *)
   DureeFonc = PROCEDURE () : LONGREAL; (* Type de proc. pour retourner   *)
                                (* une durée de fonctionnement.           *)
   DureeRepar = PROCEDURE (VAR BOOLEAN) : LONGREAL;
                                (* Retourne une durée de répar. et met    *)
                                (* Succes a TRUE ssi la répar. a réussi.  *)
   InfoGroupe = RECORD
      N       : Num;            (* Nb. de machines.                       *)
      NbOK    : ExtNum;         (* Nb. de machines qui fonctionnent.      *)
      Pos     : LONGREAL;       (* Position courante du robot.            *)
      TDepart : LONGREAL;       (* Instant de la dernière mise-a-jour     *)
                                (* de Pos, si le robot voyage.            *)
      Dir     : Direction;      (* Dernière direction de déplacement.     *)
      Devant  : ExtNum;         (* Si > 0, le robot est en face de la     *)
                                (* mach. i, sinon il est entre 2 mach.    *)
      Dest    : ExtNum;         (* Si > 0, le robot va vers cette machine *)
                                (* (ou s'y trouve), si = 0, il répare     *)
                                (* la machine en face de lui.             *)
      EnPanne     : EnsMach;    (* Ensemble des machines en panne.        *)
      Decision    : Politique;  (* Règle de décision utilisée.            *)
      DFonc       : DureeFonc;  (* Génère une durée de fonctionnement.    *)
      DRepar      : DureeRepar; (* Une durée de réparation par le robot.  *)
      DReparMan   : DureeRepar; (* Une durée de rép. par le réparateur.   *)
      Reparateur  : RES.Resource; (* Le réparateur affecté a ce groupe.   *)
      Prod        : STAT.Block; (* Bloc stat. sur la productivité.        *)
      Robot       : PROCS.ProcessInstance;
      Machine     : ARRAY Num OF PROCS.ProcessInstance;
      AvisArrivee : EVENT.EventNotice (* Arriv. prévue du robot à une mach.*)
      END;

PROCEDURE Create
   ( VAR G : Groupe;
     N, Devant  : Num;
     Decision   : Politique;
     DFonc      : DureeFonc;
     DRepar     : DureeRepar;
     DReparMan  : DureeRepar;
     Reparateur : RES.Resource
   );
   (* Crée le groupe de machines G, et initialise les champs du RECORD    *)
   (* aux valeurs passées en paramètres.                                  *)

PROCEDURE Delete (VAR G : Groupe);
   (* Détruit le groupe de machines G et récupère l'espace.               *)

END GroupMach.
```

Figure 11: Definition du module `GroupMach`.

```
DEFINITION MODULE Commande;

FROM GroupMach IMPORT ExtNum, Groupe;

PROCEDURE BalayageComplet (G : Groupe) : ExtNum;
   (* Patrouille la ligne de 1 a N, puis de N a 1, etc., en reparant      *)
   (* les machines brisees que l'on rencontre.                            *)

PROCEDURE PlusPres (G : Groupe) : ExtNum;
   (* Si toutes les machines fonctionnent, on va vers le centre puis on   *)
   (* attend, sinon on va vers la machine brisee la plus pres.            *)
   (* En cas d'egalite, on s'eloigne d'abord du centre.                   *)

END Commande.
```

Figure 12: Definition du module `Commande`.

puis de droite à gauche jusqu'à la machine 1, et ainsi de suite. Chaque fois qu'il rencontre une machine bloquée, le robot s'arrête pour la réparer. Sous la politique `PlusPres`, le robot entreprend aussi la réparation chaque fois qu'il se trouve en face d'une machine bloquée. Si toutes les machines fonctionnent, il se dirige vers la machine centrale (ou la plus proche des deux machines centrales s'il y a un nombre pair de machines). Sinon, s'il n'est pas déjà en face d'une machine bloquée, il se dirige vers la machine bloquée qui se trouve le plus près de lui. Ainsi, si une machine se bloque pendant que le robot se déplace, il est possible que ce dernier change de direction. En cas d'égalité (deux machines bloquées à la même distance), le robot choisit la direction de l'extrémité la plus proche (il s'éloigne du centre de la ligne), et si l'égalité persiste, il choisit au hasard. On pourrait évidemment ajouter d'autres politiques à ce module, et en particulier celles suggérées dans [17]. On doit fournir une telle politique chaque fois que l'on initialise un groupe de machines. Evidemment, il n'est pas nécessaire que la politique fournie provienne du module `Commande`, elle peut provenir d'ailleurs. En pratique, on pourrait éventuellement remplacer l'implantation du module `GroupMach` par de vrais groupes de machines, commandés en utilisant directement les procédures du module `Commande`.

Exemple d'une expérience. Dans l'expérience décrite dans le module de la figure 13, on ne considère qu'un seul groupe de machines. On examine différentes politiques pour différentes configurations, et dans chaque cas, on calcule un intervalle de confiance sur la productivité. Pour calculer cet intervalle de confiance, on utilise ici la méthode des lots ("batch means") [4, 12]. On simule pour une durée donnée, que l'on subdivise en `NLots` intervalles de temps égaux (de durée `DureeLot`). Les résultats des `Rechauf` premiers intervalles sont éliminés, et on traite les moyennes associées aux intervalles restants comme `(NLots-Rechauf)` valeurs de variables aléatoires indépendantes, distribuées selon la loi normale. Le bloc statistique `ProdMoy` sert à recueillir ces valeurs et à calculer l'intervalle de confiance. Il est mis-à-jour à chaque fin de lot (sauf pour ceux du réchauffement) par l'événement `FinLot`, qui exécute la procédure `ProcFinLot`. Cette procédure réinitialise aussi le bloc statistique `Prod` associé au groupe de machines, et qui sert à calculer la productivité moyenne pour chaque

```
MODULE Experience;

IMPORT SIM, STAT, EVENT, RAND1, RES;
FROM MYINOUT   IMPORT WriteLn, ReadLn, WriteString, ReadString, ReadCard,
               WriteCard, OpenInput, CloseInput, OpenOutput, CloseOutput;
FROM NUM       IMPORT LONGREAL, WriteLR, ReadLR;
FROM CHRONO    IMPORT Chrono, TimeUnit, InitChrono, WriteChrono;
FROM EVENT     IMPORT EventType, Schedule, Attrib;
FROM RAND      IMPORT Uniform01;
FROM STAT      IMPORT Update, Block, Tally, Average;
FROM GroupMach IMPORT Groupe, Create, Delete, Politique, DureeRepar;
FROM Commande  IMPORT BalayageComplet, PlusPres;

VAR
   N        : CARDINAL;         (* Nb. de machines.                         *)
   Lot      : CARDINAL;         (* No. du lot courant.                      *)
   NLots    : CARDINAL;         (* Nb. de lots a effectuer au total.        *)
   Rechauf  : CARDINAL;         (* Nb. de lots pour le réchauffement.       *)
   DureeLot : LONGREAL;         (* Durée de simulation par lot.             *)
   ProdMoy  : Block;            (* Stats. pour les différents lots.         *)
   FinLot   : EventType;        (* La fin d'un lot.                         *)
   Alpha, Lambda, ProbSucces, DureeR, DureeRM : LONGREAL;
                                (* Paramètres des lois ci-bas.              *)
   Ident    : ARRAY [0..63] OF CHAR;
   Reparateur : RES.Resource;
   G : Groupe;   C : Chrono;   i : CARDINAL;

PROCEDURE DFonc () : LONGREAL;
   (* Duree de fonctionnement entre deux pannes.                            *)
   BEGIN  RETURN RAND1.Gamma (Alpha, Lambda, 1)  END DFonc;

PROCEDURE DRepMan (VAR S : BOOLEAN) : LONGREAL;
   (* Duree de reparation manuelle.                                         *)
   BEGIN  S := TRUE;  RETURN DureeRM  END DRepMan;

PROCEDURE Const (VAR S : BOOLEAN) : LONGREAL;
   BEGIN  S := TRUE;  RETURN DureeR END Const;

PROCEDURE Geo1 (VAR S : BOOLEAN) : LONGREAL;
   BEGIN
   IF Uniform01 (2) <= ProbSucces THEN S := TRUE ELSE S := FALSE END;
   RETURN DureeR
   END Geo1;

PROCEDURE Geo2 (VAR S : BOOLEAN) : LONGREAL;
   BEGIN
   IF Uniform01 (2) <= ProbSucces THEN S := TRUE;  RETURN DureeR  END;
   IF Uniform01 (2) <= ProbSucces THEN S := TRUE  ELSE S := FALSE END;
   RETURN (DureeR + DureeR)
   END Geo2;

PROCEDURE ProcFinLot;
   BEGIN
   IF Lot > Rechauf THEN Update (ProdMoy, Average (G^.Prod)) END;
   STAT.Init (G^.Prod);
   IF Lot >= NLots THEN  Rapport
   ELSE  INC (Lot);  Schedule (FinLot, DureeLot, NIL)  END
   END ProcFinLot;
```

Figure 13: Implantation du module `Experience`.

```
PROCEDURE Rapport;
   (* Produit un rapport a la fin de la simulation d'une configuration.    *)
   VAR
      MeanOp, MeanW, RadiusW, MeanProd, RadiusP, B1, B2 : LONGREAL;
   BEGIN
   WriteString ("Temps d'execution (CPU)   : ");
   WriteChrono (C, sec);  WriteLn;
   WriteString ("Temps de simulation (sec) : ");
   WriteLR (SIM.Time (), 9, 1, 1);
   STAT.ReportConfidenceInterval (ProdMoy, 0.95);  WriteLn;
   Delete (G);  SIM.Stop
   END Rapport;

PROCEDURE EssayerConfig
   ( Pol : Politique;  DRep : DureeRepar;  Nam : ARRAY OF CHAR
   );
   (* Simule pour la configuration courante avec la politique Pol.         *)
   (* La loi de prob. de la duree de repar. par le robot est DRep.         *)
   BEGIN
   WriteString ("CONFIGURATION: ");  WriteString (Ident);
   WriteString (",  ");    WriteString (Nam);   WriteLn;
   SIM.Init;  STAT.Init (ProdMoy);  RES.Init (Reparateur);  Lot := 1;
   Create (G, N, (N + 1) DIV 2, Pol, DFonc, DRep, DRepMan, Reparateur);
   Schedule (FinLot, DureeLot, NIL);
   InitChrono (C);
   SIM.Start
   END EssayerConfig;

BEGIN
EVENT.Create (FinLot, ProcFinLot, "Fin d'un lot");
STAT.Create (ProdMoy, Tally, "Prod. moyenne");
RES.Create (Reparateur, RES.Fifo, 1, "Le reparateur");
OpenInput (".DAT");  OpenOutput (".RES");
ReadCard (NLots);  ReadCard (Rechauf);  ReadLongReal (DureeLot);
WriteString ("Simulation d'une serie de machines");
WriteString (" entretenues par un robot.");
WriteLn;  WriteLn;
WriteString ("Nombre de lots                          : ");
WriteCard (NLots, 8);  WriteLn;
WriteString ("Nombre de lots pour rechauffement      : ");
WriteCard (Rechauf, 8);  WriteLn;
WriteString ("Duree de la simulation pour chaque lot : ");
WriteLR (DureeLot, 10, 1, 1);  WriteLn;  WriteLn;
ReadString (Ident);  ReadLn;
WHILE Ident [0] # "0" DO
   ReadCard (N);
   ReadLR (Alpha);   ReadLR (Lambda);
   ReadLR (ProbSucces);  ReadLR (DureeR);  ReadLR (DureeRM);
   EssayerConfig (BalayageComplet, Const, "Balayage complet, rep. const.");
   EssayerConfig (PlusPres, Const, "Plus pres, rep. const");
   EssayerConfig (PlusPres, Geo1, "Plus pres, rep. Geo1");
   EssayerConfig (PlusPres, Geo2, "Plus pres, rep. Geo2");
   ReadString (Ident);   ReadLn
   END;
CloseInput;  CloseOutput
END Experience.
```

Figure 13: Implantation du module `Experience` (suite).

lot. Lorsqu'on a terminé le dernier lot, on appelle la procédure `Rapport`, qui imprime les résultats, détruit le groupe `G`, et arrête la simulation.

Dans le programme principal, on crée le type d'événement `FinLot`, le bloc statistique et le réparateur. On lit ensuite le nombre de lots, le nombre de lots à rejeter, la durée de la simulation pour chaque lot (la durée totale sera `DureeLot * NLots`), puis on imprime une entête. Ensuite, dans le fichier de données, il y a deux lignes de données pour chaque configuration que l'on doit traiter. La première ligne contient une chaîne de caractères décrivant cette configuration, et la seconde contient le nombre de machines et les valeurs des paramètres à utiliser dans les lois de probabilité définies au début du module. Après la dernière configuration, le fichier contient une ligne dont le premier caractère est un zéro. La boucle `WHILE` du programme principal lit les configurations et les traite. Un appel à `EssayerConfig` effectue une simulation en utilisant les paramètres que l'on vient de lire. La politique à utiliser et la loi de probabilité de la durée de réparation par le robot sont passées en paramètres. Les autres lois sont fixées, mais on a lu les valeurs de leurs paramètres. A noter que cette façon de faire est tout à fait arbitraire et sert ici à illustrer la flexibilité dont on dispose. On suppose ici que l'on ne veut varier que la politique et la loi `DRep`, mais pas les autres lois. Dans `EssayerConfig`, on réinitialise la simulation, le bloc statistique, le réparateur et le chronomètre. On crée et initialise ensuite un groupe de machines, et on prévoit un événement qui marquera la fin du premier lot. A noter que comme chaque appel à `EssayerConfig` crée un nouveau groupe de machines, il est important de détruire `G` à la fin de la simulation. Sinon, tous les groupes de machines que l'on a crée vont continuer d'exister et d'être simulés indéfiniment. Ici, toutes les configurations sont simulées avec des valeurs aléatoires différentes. Si on voulait calculer un intervalle de confiance pour la différence entre deux configurations, il vaudrait probablement mieux utiliser des valeurs aléatoires communes, comme dans l'exemple de la section 3.3.

Les cinq premières procédures du module implantent les lois de probabilité utilisées. La durée de fonctionnement d'une machine suit la loi Gamma de paramètres `Alpha` et `Lambda`. Le temps requis pour compléter manuellement une réparation est constant et égal à `DureeRM`. Les trois autres procédures servent à générer des durées de réparation par le robot. Dans le cas de `Const`, le réparation réussit toujours et sa durée est constante et égale à `DureeR`. Dans le cas de `Geo1`, elle a la même durée, mais ne réussit qu'avec une probabilité égale à `ProbSucces`. `Geo2` est semblable à `Geo1`, sauf que si la réparation échoue au premier essai, le robot tente un second essai (avec la même probabilité de réussite) avant de faire venir le réparateur. On pourrait évidemment ajouter d'autres procédures, implantant d'autres lois de probabilité. Selon leur nombre ou leur taille, on pourrait aussi décider de les placer dans un module à part, tout comme on l'a fait pour les politiques de commande.

```
10  1  1000.0
Config1
10  1.5  0.015  0.8  5.0  5.0
0
```

Figure 14: Un fichier de données pour le module `Experience`.

```
Simulation d'une serie de machines entretenues par un robot.

Nombre de lots                         :       10
Nombre de lots pour rechauffement      :        1
Duree de la simulation pour chaque lot :     1000.0

CONFIGURATION: Config1,  Balayage complet, rep. const.
Temps d'execution (CPU)   :       2.80 seconds
Temps de simulation (sec) :   10000.0
REPORT ON STATISTIC (Tally) : Prod. moyenne
                  min        max      average  standard dev.  nb. Obs.
                  8.345      8.827      8.596      0.191          9
            95% confidence interval for mean : (     8.449,      8.742 )

CONFIGURATION: Config1,  Plus pres, rep. const
Temps d'execution (CPU)   :       2.80 seconds
Temps de simulation (sec) :   10000.0
REPORT ON STATISTIC (Tally) : Prod. moyenne
                  min        max      average  standard dev.  nb. Obs.
                  8.609      9.117      8.847      0.163          9
            95% confidence interval for mean : (     8.722,      8.972 )

CONFIGURATION: Config1,  Plus pres, rep. Geo1
Temps d'execution (CPU)   :       2.80 seconds
Temps de simulation (sec) :   10000.0
REPORT ON STATISTIC (Tally) : Prod. moyenne
                  min        max      average  standard dev.  nb. Obs.
                  8.397      9.195      8.706      0.241          9
            95% confidence interval for mean : (     8.522,      8.891 )

CONFIGURATION: Config1,  Plus pres, rep. Geo2
Temps d'execution (CPU)   :       2.85 seconds
Temps de simulation (sec) :   10000.0
REPORT ON STATISTIC (Tally) : Prod. moyenne
                  min        max      average  standard dev.  nb. Obs.
                  7.578      8.858      8.338      0.391          9
            95% confidence interval for mean : (     8.037,      8.638 )
```

Figure 15: Les résultats produits à partir du fichier de la figure 14.

La figure 14 montre un exemple de fichier de données, et la figure 15 donne les résultats imprimés par le programme en utilisant ce fichier de données. Dans cet exemple, il n'y a qu'une seule configuration. Habituellement, en pratique, on imprimera un rapport plus complet, indiquant par exemple, pour chaque configuration, le nombre de machines, la politique utilisée, les paramètres des lois, etc.

Implantation de `GroupMach`. Dans l'implantation du module `GroupMach`, on voit que pour chaque groupe de machines, le robot et chacune des machines correspondent à des processus. Ces derniers sont démarrés par la procédure `Create`, lors de la création du groupe, et exécutent des boucles infinies. Ils ne se terminent donc que lorsqu'on détruit le groupe. Ces processus sont prévus avec des paramètres dont on récupère la valeur au tout début de leur exécution. Le paramètre d'un robot est le nom de son groupe, tandis que le paramètre d'une machine est un pointeur sur un enregistrement contenant le nom de son groupe et son numéro dans le groupe.

Chaque machine exécute une boucle infinie. Elle fonctionne jusqu'à ce qu'elle tombe en panne puis, si le robot n'est pas occupé à réparer une machine (il se trouve alors dans l'état `Suspended`), elle le réactive, afin qu'il prenne une nouvelle décision compte tenu du nouvel état du système. Ensuite, la machine se "suspend" et attend d'être réactivée par le robot, à la fin de la réparation, avant de recommencer la boucle.

Le robot exécute lui aussi une boucle infinie. Au début de la boucle, il prend une décision, selon l'état du groupe de machines et de la politique adoptée. S'il décide de réparer (il ne peut le faire que s'il se trouve exactement en face d'une machine brisée), il génère une durée de réparation et attend pendant cette durée. Si la réparation n'a pas réussi, il fait ensuite appel au réparateur, génère une durée de réparation manuelle, et utilise le réparateur pour cette durée. Il réactive ensuite la machine qu'il vient de réparer. Le lecteur aura sans doute remarqué que l'ensemble des machines en panne, leur nombre, et le bloc statistique sur la

```
IMPLEMENTATION MODULE GroupMach;

IMPORT SIM, STAT, EVENT, PROCS;
FROM SYSTEM  IMPORT ADDRESS;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM MYINOUT IMPORT WriteString, WriteLn;
FROM STAT    IMPORT Update;
FROM RES     IMPORT Resource, Request, Release;
FROM EVENT   IMPORT EventType, EventNotice, ScheduleNotice, CancelNotice;
FROM PROCS   IMPORT ProcessType, ScheduleNamed, Delay, Resume, Suspend,
                    CurrentProcess, InstanceState, ProcessState, Terminate;
TYPE
   ParamMach = POINTER TO RECORD (* Paramètres d'une machine.           *)
      Gr : Groupe;              (* Son Groupe.                          *)
      No : Num                  (* Son numéro.                          *)
      END;
```

Figure 16: Implantation du module `GroupMach`.

```
VAR
   UnRobot, UneMachine : ProcessType;
   ArriveeRobot        : EventType;

PROCEDURE Erreur (i : CARDINAL);
   BEGIN
   WriteString ("DECISION ILLEGALE DU ROBOT:");
   CASE i OF
      1 : WriteString (" Reparation sans etre en face d'une mach. fautive.")
    | 2 : WriteString (" Destination < 1.")
    | 3 : WriteString (" Destination > Nb. de mach.")
      END;
   WriteLn;  SIM.Stop;  Terminate
   END Erreur;

PROCEDURE Create
   ( VAR G : Groupe;  n, dev : Num;  d : Politique;
     df : DureeFonc;  dr : DureeRepar;  drm : DureeRepar;  r : Resource
   );
   VAR
      i : Num;   Par : ParamMach;
   BEGIN
   NEW (G);
   WITH G^ DO
      EnPanne := EnsMach {};
      FOR i := 1 TO n DO
         NEW (Par);   Par^.Gr := G;   Par^.No := i;
         ScheduleNamed (UneMachine, 0.0, Par, Machine [i]);
         END;
      N := n;   NbOK := n;   Devant := dev;   Pos := VAL (LONGREAL, Devant);
      Dir := Droite;   Decision  := d;  DFonc := df;   DRepar := dr;
      DReparMan := drm;  Reparateur := r;
      ScheduleNamed (UnRobot, 0.0, G, Robot);
      STAT.Create  (Prod, STAT.Accumulate, "Taux de production");
      STAT.Init (Prod);   Update (Prod, VAL (LONGREAL, N))
      END
   END Create;

PROCEDURE Delete (VAR G : Groupe);
   (* Récupère tout l'espace occupé par G.                                  *)
   VAR
      i : Num;   Par : ParamMach;
   BEGIN
   WITH G^ DO
      FOR i := 1 TO N DO
         Par := ParamMach (PROCS.InstanceAttrib (Machine [i]));
         DISPOSE (Par);  PROCS.Kill (Machine [i])
         END;
      STAT.Delete (Prod);   PROCS.Kill (Robot)
      END;
   DISPOSE (G)
   END Delete;

PROCEDURE ProcArrivee;
   (* Evénement : Le robot atteint sa destination.                          *)
   VAR
      G : Groupe;
   BEGIN
   G := Groupe (EVENT.Attrib ());
   WITH G^ DO  Devant := Dest;  Resume (Robot)  END;
   END ProcArrivee;
```

Figure 16: Implantation du module `GroupMach` (suite).

```
PROCEDURE ProcMachine;
   (* Décrit le comportement d'une machine.                            *)
   VAR
      No : Num;                (* Son numéro.                          *)
      Par : ParamMach;         (* Pointeur sur ses paramètres.         *)
   BEGIN
   Par := ParamMach (PROCS.Attrib ());   No := Par^.No;
   WITH Par^.Gr^ DO            (* Groupe auquel elle appartient.       *)
      WHILE TRUE DO
         Delay (DFonc ());     (* Fin du délai: La mach. tombe en panne. *)
         DEC (NbOK);  INCL (EnPanne, No);  Update (Prod, VAL (LONGREAL, NbOK));
         IF InstanceState (Robot) = Suspended THEN  Resume (Robot) END;
         Suspend
         END                   (* Réactivation: fin de la réparation.  *)
      END
   END ProcMachine;

PROCEDURE ProcRobot;
   VAR
      S : BOOLEAN;  G : Groupe;
   BEGIN
   G := Groupe (PROCS.Attrib ()); (* Groupe auquel il appartient.      *)
   WITH G^ DO
      WHILE TRUE DO
         Dest := Decision (G);
         IF Dest = 0 THEN      (* Répare la machine devant lui.        *)
            IF (Devant = 0) OR (NOT (Devant IN EnPanne)) THEN Erreur (1) END;
            Delay (DRepar (S));
            IF (NOT S) THEN    (* La réparation a échoué.              *)
               Request (1, Reparateur);
               Delay (DReparMan (S));
               Release (1, Reparateur)
               END;
            Resume (Machine [Devant]);
            (* Doit s'exécuter avant la prochaine prise de décision:   *)
            INC (NbOK);  EXCL (EnPanne, Devant);
            Update (Prod, VAL (LONGREAL, NbOK))
         ELSIF Dest = Devant THEN  Suspend (* Le robot attend.         *)
         ELSE                  (* Le robot se déplace.                 *)
            IF Dest > N THEN  Erreur (3) END;
            IF VAL (LONGREAL, Dest) > Pos THEN Dir := Droite ELSE Dir := Gauche END;
            Devant := 0;  TDepart := SIM.Time ();
            ScheduleNotice (ArriveeRobot, ABS (Pos - VAL (LONGREAL, Dest)),
                            G, AvisArrivee);
            Suspend;
            IF Devant = Dest THEN (* Le robot arrive à destination.    *)
               Pos := VAL (LONGREAL, Dest)
            ELSE               (* Une machine tombe en panne.          *)
               IF Dir = Droite THEN  Pos := Pos + SIM.Time () - TDepart
               ELSE  Pos := Pos - SIM.Time () + TDepart  END;
               CancelNotice (AvisArrivee)
               END
            END
         END
      END
   END ProcRobot;

BEGIN
EVENT.Create (ArriveeRobot, ProcArrivee, "Arrivee du robot");
PROCS.Create (UneMachine, ProcMachine, 2000, "Une machine");
PROCS.Create (UnRobot, ProcRobot, 2000, "Un robot")
END GroupMach.
```

Figure 16: Implantation du module GroupMach (suite).

```
IMPLEMENTATION MODULE Commande;
FROM NUM       IMPORT LONGREAL;
FROM RAND      IMPORT Uniform01;
FROM GroupMach IMPORT Num, ExtNum, EnsMach, Direction, Groupe;

PROCEDURE BalayageComplet (G : Groupe) : ExtNum;
   VAR
      m : ExtNum;
   BEGIN
   WITH G^ DO
      IF (Devant IN EnPanne) THEN  RETURN 0  END;  (* Répare.                 *)
      IF N = 1 THEN RETURN 1 END;
      IF Devant = 1 THEN  Dir := Droite  END;
      IF Devant = N THEN  Dir := Gauche  END;
      IF Devant > 0 THEN         (* En face d'une machine.                    *)
         m := Devant
      ELSE                       (* Entre deux machines.                      *)
         m := TRUNC (Pos);
         IF ((Dir = Droite) AND (m < N)) OR (m < 1) THEN  INC (m)  END
         END;
      IF Dir = Droite THEN
         WHILE ((m < N) AND (NOT (m IN EnPanne))) DO  INC (m)  END
      ELSE
         WHILE ((m > 1) AND (NOT (m IN EnPanne))) DO  DEC (m)  END
         END;
      RETURN m
      END
   END BalayageComplet;

PROCEDURE PlusPres (G : Groupe) : ExtNum;
   VAR
      i, j : ExtNum;   D, Dmin : LONGREAL;
   BEGIN
   WITH G^ DO
      IF (Devant IN EnPanne) THEN  RETURN 0  END;  (* Répare.                 *)
      IF N = 1 THEN RETURN 1 END;
      IF (EnPanne = EnsMach{}) THEN (* Pas de panne, va vers le centre.     *)
         IF (Pos < VAL (LONGREAL, N + 1) / 2.0) THEN  RETURN (N + 1) DIV 2
         ELSE  RETURN (N + 2) DIV 2  END
         END;
      (* Trouve la mach. en panne la plus proche: la mach. j.                 *)
      Dmin := VAL (LONGREAL, N);
      FOR i := 1 TO N DO
         IF i IN EnPanne THEN
            D := ABS (VAL (LONGREAL, i) - Pos);
            IF D < Dmin THEN  j := i;  Dmin := D  END
            END
         END;
      (* Vérifie les égalités (si devant une machine).                        *)
      IF Devant > 0 THEN
         i := 2 * Devant - j;     (* La mach. i est à la même dist. que j.  *)
         IF ((1 <= i) AND (i <= N) AND (i IN EnPanne)) THEN
            IF (((i < j) AND (i - 1 < N - j)) OR ((i > j) AND (N - i < j - 1))
                              (* i est plus près d'une extrémité.       *)
                  OR ((i - 1 = N - j) AND (Uniform01 (4) < 0.5))) THEN
               RETURN i
               END
            END
         END
      END;
   RETURN j
   END PlusPres;

END Commande.
```

Figure 17: Implantation du module `Commande`.

productivité, sont mis-à-jour par le processus `Machine` lors d'une panne, mais par le processus `Robot` à la fin de la réparation. La raison est qu'à la fin d'une réparation, cette mise-à-jour doit se faire *avant* que le robot ne prenne sa prochaine décision. Comme la machine ne prend pas le contrôle immédiatement dès que le robot appelle `Resume`, mais seulement après qu'il ait appelé `Suspend`, le robot prendra sa prochaine décision avant de passer le contrôle à la machine. Une autre approche consisterait à forcer le robot à passer le contrôle à la machine avant la prochaine prise de décision, par exemple en plaçant l'instruction `Delay (0.0)` juste après l'appel à `Resume`.

Dans le cas où le robot est exactement en face d'une machine et décide d'y rester, il se suspend et attend d'être réactivé par la prochaine machine qui tombera en panne. S'il décide de se déplacer, il prévoit un événement qui correspond à son arrivée à destination, et se suspend. Si aucune panne ne survient d'ici là, cet événement réactive le robot. Sinon, ce dernier est réactivé par une machine qui tombe en panne, et annule l'événement qui prévoyait sa réactivation. Dans les deux cas, il met à jour sa position et reprend une nouvelle décision.

Discussion. Souvent, comme le laisse entrevoir l'implantation du module `Commande`, la politique de commande d'un système peut s'exprimer assez facilement sous forme de règles. Dans certains cas, il pourrait être intéressant d'implanter cette politique sous forme d'une base de règles, par exemple sous forme d'un programme PROLOG. Le système *Modula-Prolog* décrit dans [21] permet d'utiliser Modula-2 et PROLOG simultanément, dans un même programme, et pourrait donc permettre de le faire.

A remarquer que nous ne présentons pas ici le programme de simulation le plus efficace pour estimer la productivité d'un groupe de machines. Pour diminuer les temps d'exécution, on pourrait d'abord facilement précalculer certaines expressions. Ensuite, l'adoption de techniques de réduction de la variance, telles que celles utilisées dans [17], permettrait de réduire par un facteur de 5 à 10 le temps de simulation requis pour un intervalle de confiance de rayon donné. Enfin, si on utilisait une approche uniquement par événements au lieu d'utiliser des processus, on réduirait encore les temps d'exécution de façon substantielle. Les événements correspondraient alors aux instants où le robot arrive à destination, où une machine tombe en panne, et où une réparation se termine. A noter que dans ce cas-ci, le code ne serait pas beaucoup plus long ni plus compliqué.

Un système composé de plusieurs groupes de machines. Pour effectuer l'expérience décrite à la figure 13, il n'était pas vraiment nécessaire de définir un module `GroupMach` aussi élaboré. Mais l'intérêt de la définition adoptée est qu'elle permet de créer autant de groupes de machines que l'on veut, et de les faire évoluer en même temps. De plus, ces groupes de machines ne sont pas nécessairement indépendants: ils peuvent par exemple partager un ou plusieurs réparateur(s). Ainsi, notre programme de simulation d'un groupe de machines peut servir de "composante" d'un programme de simulation d'un plus gros système, qui peut lui-même servir de composante d'un programme encore plus gros, etc. Le programme de la figure 18 illustre cette idée. On crée quatre groupes de machines, dont trois groupes identiques qui partagent un même réparateur, et un quatrième groupe qui possède son propre réparateur. On simule durant 2000 unités de temps, puis on fait imprimer un rapport sur chacun des groupes de machines.

```
MODULE Systeme;

IMPORT SIM, STAT, EVENT, RAND, RAND1, RES;
FROM NUM       IMPORT LONGREAL;
FROM EVENT     IMPORT EventType, Schedule;
FROM GroupMach IMPORT Groupe, Create;
FROM Commande  IMPORT BalayageComplet, PlusPres;

VAR
   FinSim   : EventType;        (* La fin de la simulation.               *)
   Guy, Bob : RES.Resource;
   G        : ARRAY [1..4] OF Groupe;
   i        : CARDINAL;

PROCEDURE DFonc () : LONGREAL;
   BEGIN  RETURN RAND1.Expon (100.0, 1)  END DFonc;

PROCEDURE DRepMan (VAR S : BOOLEAN) : LONGREAL;
   BEGIN  S := TRUE;  RETURN 5.0  END DRepMan;

PROCEDURE Geo2 (VAR S : BOOLEAN) : LONGREAL;
   BEGIN
   IF RAND.Uniform01 (2) <= 0.8 THEN S := TRUE;  RETURN 5.0  END;
   IF RAND.Uniform01 (2) <= 0.8 THEN S := TRUE  ELSE S := FALSE END;
   RETURN 10.0
   END Geo2;

PROCEDURE ProcFinSim;
   BEGIN
   FOR i := 1 TO 4 DO STAT.Report (G [i]^.Prod) END;
   SIM.Stop
   END ProcFinSim;

BEGIN
EVENT.Create (FinSim, ProcFinSim, "Fin de la simulation");
RES.Create (Guy, RES.Fifo, 1, "Le reparateur Guy");
RES.Create (Bob, RES.Fifo, 1, "Le reparateur Bob");
FOR i := 1 TO 3 DO
   Create (G [i], 10, 5, PlusPres, DFonc, Geo2, DRepMan, Guy)
   END;
Create (G [4], 15, 1, BalayageComplet, DFonc, Geo2, DRepMan, Bob);
Schedule (FinSim, 2000.0, NIL);
SIM.Start
END Systeme.
```

Figure 18: Un système comprenant plusieurs groupes de machines.

4 Conclusion

Modula-2 est un langage qui gagne de plus en plus en popularité. Il est relativement simple, permet la vérification des types et la conception modulaire, sépare la définition des objets (abstractions) de leur implantation, offre des variables de type `PROCEDURE`, et supporte les co-routines. Mais comme tout langage, Modula-2 a aussi ses faiblesses. A ce sujet, on peut lire la critique de Moffat [20] et la section 5 de [15].

Parmi les aspects agaçants ou restrictifs, mentionnons d'abord le fait que Modula-2 ne produit pas vraiment des programmes tout-à-fait “portables”, car en général, on doit faire appel à des librairies de modules, et ces derniers varient d'une implantation à l'autre. Cependant, les programmes sont en général faciles à modifier pour passer d'une implantation à l'autre; il suffit souvent de changer quelques identificateurs. Modula-2 n'est pas non plus un langage tout à fait “orienté-objet”. En particulier, il ne supporte pas la notion d'extension de type (définition d'un sous-type qui hérite des propriétés du type parent, et qui peut posséder des propriétés additionnelles). On peut mentionner aussi le traitement des entrées/sorties (qui ne fait pas vraiment partie du langage), l'impossibilité d'avoir des procédures qui ont des paramètres optionnels ou qui varient en nombre, l'impossibilité de surcharger les identificateurs (plusieurs procédures qui ont le même nom et qui se distinguent par les types de leurs paramètres), l'impossibilité d'avoir des co-routines dont les procédures associées ont des paramètres, l'obligation de calculer la taille de l'espace mémoire alloué à chaque co-routine, le fait que les pointeurs ne sont pas initialisés à `NIL` (ce qui nous empêche de faire vérifier par SIMOD si un objet a déjà été créé ou pas), le traitement des exceptions, le traitement des chaînes de caractères, etc. Il est clair, cependant, qu'ajouter au langage toutes ces facilités augmenterait la difficulté d'implantation et la complexité des compilateurs.

Modula-2 demeure malgré tout très intéressant, efficace et pratique pour implanter des logiciels de simulation. Les principaux points forts de SIMOD sont (a) sa grande flexibilité (accès à un langage de base moderne et connu, possibilité d'appeler des routines dans d'autres langages), (b) l'efficacité du code produit, (c) l'utilisation d'une syntaxe répandue et qui donne lieu à des programmes lisibles, et (d) la puissance des outils disponibles. Comme nous l'avons vu dans les exemples de la section 3, SIMOD est au moins aussi puissant, pour la simulation à événements discrets, que les langages spécialisés couramment utilisés (tels que GPSS, SLAM, SIMSCRIPT II.5, SIMAN). Il est aussi relativement facile d'y ajouter d'autres modules au besoin. On pourrait par exemple y ajouter des outils de haut niveau pour l'animation graphique, etc. Déjà, on peut utiliser SIMOD conjointement avec des progiciels existants pour la gestion de bases de données, les statistiques, le graphisme par ordinateur, etc. Du point de vue de la vitesse d'exécution, des tests empiriques nous ont permis de constater que des versions SIMSCRIPT II.5 des exemples de la section 3 consommaient environ entre 20% et 100% de plus de temps de CPU, à l'exécution, que les versions SIMOD. Comme nous n'avons pas encore vraiment pris soin d'optimiser l'implantation de SIMOD, il y a certainement moyen d'améliorer davantage sa performance, et ceci est très encourageant.

Nous avons vu, à la section 3.7, comment SIMOD permet et encourage l'implantation modulaire d'un programme de simulation, et comment cette modularité facilite la modification ultérieure d'une partie du modèle, ou des règles de commande, ou encore du plan d'expérience, etc. L'avantage de cette approche sera d'autant plus marqué que le modèle sera complexe et/ou de grande taille. Ainsi, SIMOD n'est pas seulement un outil pédagogique servant à traiter de petits exemples académiques, mais est aussi tout à fait approprié pour la simulation des grands systèmes rencontrés en pratique.

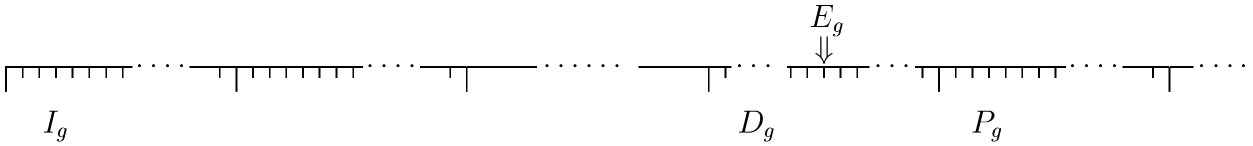
ANNEXES

A. Définition fonctionnelle des modules de SIMOD

RAND

Ce module fournit les générateurs aléatoires de base. On peut l'utiliser directement, mais il sert aussi de base au module `RAND1`, qui permet de générer des valeurs aléatoires suivant des lois de probabilité plus élaborées. Le générateur de base utilisé est celui proposé par L'Ecuyer [16], dont la période est d'environ 2.3×10^{18}, et ce module implante les outils proposés dans [14]. A partir de la suite des états visités dans une période par le générateur de base, on extrait K sous-suites disjointes (et contiguës) de 2^{50} états chacune. On obtient ainsi K générateurs "virtuels", numérotés de 1 à K. Dans la présente version du module, la constante K est fixée à 32. On peut la modifier si désiré, à condition de recompiler le module et de choisir $0 < K < 2048$. A tout moment durant l'exécution du programme, l'état de chaque générateur est une paire d'entiers positifs et correspond à l'une des 2^{50} valeurs de la sous-suite associée. Cet état est modifié à chaque appel.

La sous-suite associée à chaque générateur est aussi partitionnée en 2^{20} sous-sous-suites de 2^{30} états chacune. Les premiers états de ces sous-sous-suites correspondent à ce qu'on appelle des "germes". Supposons que le générateur g est dans l'état E_g, disons dans sa sous-sous-suite numéro k. On définit son *dernier germe* D_g comme étant le germe initial de cette sous-sous-suite, et son *prochain germe* P_g comme étant le germe initial de la sous-sous-suite suivante. Ainsi, si I_g désigne le *germe initial* du générateur g (celui associé à sa première sous-sous-suite), alors D_g et P_g sont respectivement les $(k-1)2^{30}$-ième et $k2^{30}$-ième états suivant I_g dans la séquence de base.



Initialement, on a $E_g = D_g = I_g$ pour chaque générateur g. L'état initial du premier générateur est fixé par défaut à $I_1 = (1234567890, 123456789)$. L'usager peut cependant choisir un état initial différent, s'il le désire, en utilisant la procédure `SetInitialSeed`. Cette procédure recalcule I_g et réinitialise E_g et D_g à I_g pour chaque générateur.

La procédure `InitGenerator` permet de réinitialiser l'état E_g du générateur g soit à son germe initial I_g, soit à son dernier germe D_g, ou soit à un nouveau germe qui correspond à son prochain germe P_g. La valeur de D_g est mise à jour en conséquence. On peut ainsi, avec chaque générateur, générer plusieurs séquences disjointes (pourvu que leur longueur ne dépasse pas 2^{30}), et chaque séquence ne dépend pas de la longueur des autres séquences produites. En pratique, lorsqu'on veut comparer deux ou plusieurs configurations d'un système en utilisant la simulation avec des valeurs aléatoires communes, et que l'on veut effectuer plusieurs répétitions, on fournit pour chaque répétition un nouveau germe à chaque générateur utilisé. Cela permet de s'assurer que pour un même numéro de répétition, les germes utilisés sont les mêmes quelque soit la configuration.

La procédure `GetState` permet de lire l'état E_g du générateur numéro g. Il est aussi possible de modifier le germe initial d'un générateur particulier sans modifier celui des autres générateurs. Il suffit d'appeler `SetSeed`. On peut aussi faire avancer l'état d'un générateur de 2^k valeurs en appelant `AdvanceState`. A noter cependant qu'après un appel à ces deux dernières fonctions, les germes initiaux ne sont plus espacés de 2^{50} valeurs. Pour cette raison, il est recommandé d'éviter d'utiliser ces procédures inutilement.

Deux fonctions permettent de générer des valeurs selon la loi uniforme : `Uniform01` génère des valeurs réelles comprises (strictement) entre 0 et 1, tandis que `Random` génère des entiers dans l'ensemble $\{1, \ldots, 2147483562\}$. Ces valeurs peuvent être utilisées et tranformées, si désiré, pour générer des valeurs suivant d'autres lois de probabilité. La fonction `Random` est fournie parce qu'il est parfois préférable d'utiliser directement l'entier produit par le générateur plutôt que sa transformation en un nombre réel point flottant.

Chaque générateur peut produire des valeurs *antithétiques* par rapport aux valeurs normalement produites. Le système maintient à cet effet un interrupteur booléen pour chaque générateur. Sa valeur initiale de défaut est à `FALSE`, mais la procédure `SetAntithetic` permet de la modifier. Si x est la valeur normalement retournée par `Uniform01` (lorsque son interrupteur est à `FALSE`), alors $1 - x$ est la valeur antithétique, retournée lorsque son interrupteur est à `TRUE`. La même chose tient pour `Random`, avec n et $2147483561 - n$ à la place de x et $1 - x$.

```
DEFINITION MODULE RAND;

TYPE
   Gen = [1..32];
```

Un numéro de générateur.

```
   SeedType = (InitialSeed, LastSeed, NewSeed);

PROCEDURE InitGenerator
   ( g : Gen;
     Where : SeedType
   );
```

Réinitialise l'état du générateur `g`. Selon la valeur du paramètre `Where`, ce générateur sera remis à son germe initial I_g (`InitialSeed`), ou à son dernier germe D_g (`LastSeed`), ou à un nouveau germe (`NewSeed`), égal à P_g, et espacé de D_g de 2^{30} valeurs.

```
PROCEDURE SetAntithetic
   ( g : Gen;
     B : BOOLEAN
   );
```

A partir du moment où cette procédure est appelée avec `B = TRUE`, le générateur `g` commence à générer des valeurs antithétiques, et le fait jusqu'à ce que cette procédure soit de nouveau appelée pour `g` avec `B = FALSE`. Initialement (par défaut) tous les générateurs produisent des valeurs non-antithétiques.

```
PROCEDURE SetInitialSeed
   ( S1, S2 : LONGINT
   );
```

Initialise le germe initial I_1 du générateur numéro 1 au couple (`S1`, `S2`). Ces valeurs doivent satisfaire : $1 \le$ `S1` ≤ 2147483562 et $1 \le$ `S2` ≤ 2147483398. Les germes initiaux de tous les autres générateurs sont recalculés en conséquence, de façon à ce qu'ils soient espacés de 2^{50} valeurs, et tous les générateurs sont réinitialisés à leurs germes initiaux. Cette procédure est appelée automatiquement par *SIMOD* au tout début de l'exécution, avec les valeurs de défaut `S1` = 1234567890 et `S2` = 123456789.

```
PROCEDURE SetSeed
   ( g : Gen;
     S1, S2 : LONGINT
   );
```

Initialise le germe initial I_g du générateur numéro `g` au couple (`S1`, `S2`). Ces valeurs doivent satisfaire : $1 \leq$ `S1` ≤ 2147483562 et $1 \leq$ `S2` ≤ 2147483398. Le générateur `g` est ensuite remis à ce germe initial. Les états et les germes initiaux des autres générateurs ne sont pas modifiés. A noter qu'après un appel à cette procédure, les germes initiaux des générateurs ne sont *plus* espacés de 2^{50} valeurs. Il est donc recommandé de limiter son utilisation.

```
PROCEDURE GetState
   ( g : Gen;
     VAR S1, S2 : LONGINT
   );
```

Permet d'examiner l'état E_g du générateur numéro `g`. Cet état est retourné dans `S1` et `S2`. Peut servir si on veut mémoriser l'état d'un générateur afin, par exemple, de le réinitialiser ultérieurement à cet état.

```
PROCEDURE AdvanceState
   ( g : Gen;
     k : LONGCARD
   );
```

Avance le générateur numéro `g` de 2^k valeurs, sans modifier les états des autre générateurs. Les remarques faites pour la procédure `SetSeed`, s'appliquent ici également.

```
PROCEDURE Random
   ( g : Gen
   ) : LONGCARD ;
```

Fournit une valeur aléatoire suivant la loi uniforme discrète sur les entiers $\{1, 2, \ldots, 2147483562\}$, en utilisant le générateur numéro `g`.

```
PROCEDURE Uniform01
   ( g : Gen
   ) : LONGREAL;
```

Fournit une valeur aléatoire suivant la loi uniforme (0,1), en utilisant le générateur numéro `g`.

```
END RAND.
```

RAND1

Ce module sert à générer des valeurs aléatoires selon différentes lois de probabilité. Des fonctions sont disponibles pour générer des valeurs selon les lois uniforme discrète, uniforme continue, exponentielle, Weibull, gamma, bêta, normale et Student. Elles utilisent toutes la méthode d'inversion [4, 12], sauf pour les lois gamma et bêta. L'usager peut aussi définir sa propre loi discrète, en donnant l'ensemble des valeurs qui peuvent être prises, et la fonction de répartition. Pour faire cela, on déclare une variable de type `DiscreteDist`, on définit la distribution en question en appelant `CreateDiscreteDist`, puis on utilise la fonction `Discrete` pour générer des valeurs.

```
DEFINITION MODULE RAND1;

FROM RAND IMPORT Gen;

TYPE
   DiscreteDist;
```

Une loi de probabilité discrète, que l'usager définira lui-même.

```
PROCEDURE DiscreteUniform
   ( N1, N2 : LONGINT;
     g : Gen
   ): LONGINT;
```

Fournit une valeur aléatoire suivant la loi uniforme discrète sur l'ensemble des entiers $\{\texttt{N1}, \ldots, \texttt{N2}\}$, en utilisant le générateur numéro `g`. La différence `N1 - N2` ne doit pas dépasser 2147483561.

```
PROCEDURE CreateDiscreteDist
   ( VAR D    : DiscreteDist;
     n        : CARDINAL;
     VAR V, F : ARRAY OF LONGREAL
   );
```

Permet de définir la loi de probabilité discrète `D`. Le nombre de valeurs qui peuvent être prises est `n`, ces valeurs (distinctes) doivent être placées dans le tableau `V` en ordre croissant (dans `V[0..n-1]`), et le tableau `F` doit contenir la valeur de la fonction de répartition à chacune de ces valeurs. Ainsi, `F[i]` est la probabilité que la valeur de la variable aléatoire soit inférieure ou égale à `V[i]`. Les valeurs placées dans `F[0..n-1]` doivent être entre 0.0 et 1.0, en ordre croissant, et on doit avoir `F[n-1]`= 1.0.

```
PROCEDURE Discrete
   ( D : DiscreteDist;
     g : Gen
   ): LONGREAL;
```

Fournit une valeur aléatoire suivant la loi discrète `D`, définie précédemment par l'usager, en utilisant le générateur numéro `g`.

```
PROCEDURE Uniform
   ( A, B : LONGREAL;
     g    : Gen
   ) : LONGREAL;
```

Fournit une valeur aléatoire suivant la loi uniforme continue de paramètres `A` et `B`, en utilisant le générateur numéro `g`. La valeur produite est comprise strictement entre `A` et `B`.

```
PROCEDURE Normal
   ( Mean : LONGREAL;         (* Moyenne.                        *)
     StdDeviation : LONGREAL; (* Ecart-type                      *)
     g : Gen              (* Numéro du générateur utilisé. *)
   ): LONGREAL;
```

Fournit une valeur aléatoire suivant la loi normale de moyenne $\mu =$ `Mean` et d'écart-type $\sigma =$ `StdDeviation`. Utilise l'inversion via la fonction `InvNormalDist`.

```
PROCEDURE InvNormalDist
   ( U : LONGREAL
   ) : LONGREAL;
```

Retourne $y = F^{-1}(u)$, où F est la fonction de répartition de la loi de normale de moyenne 0 et de variance 1. Utilise une approximation rationnelle donnant au moins 7 décimales de précision pour 10^{-20} <`U`< $1 - 10^{-20}$.

```
PROCEDURE Student
   ( DegFreedom : LONGINT; (* Nombre de degrés de liberté   *)
     g : Gen               (* Numéro du générateur utilisé. *)
   ): LONGREAL;
```

Fournit une valeur aléatoire suivant la loi de Student avec `DegFreedom` degrés de liberté. Utilise l'inversion et une approximation rationnelle.

```
PROCEDURE InvStudentDist
   ( n : LONGINT;
     u : LONGREAL
   ) : LONGREAL;
```

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

```
PROCEDURE Expon
   ( Mean : LONGREAL;         (* Moyenne.                        *)
     g    : Gen           (* Numéro du générateur utilisé. *)
   ) : LONGREAL;
```

Fournit une valeur aléatoire suivant la loi exponentielle de moyenne $\mu =$ `Mean`, c'est-à-dire, dont la fonction de répartition est

$$F(x) = 1 - e^{-x/\mu}, \qquad x > 0.$$

Appelle la fonction `Uniform01` et utilise l'inversion.

```
PROCEDURE Weibull
   ( Alpha  : LONGREAL;        (* Paramètre de forme.           *)
     Lambda : LONGREAL;        (* Paramètre d'échelle.          *)
     g : Gen               (* Numéro du générateur utilisé. *)
   ): LONGREAL;
```

Fournit une valeur aléatoire suivant la loi Weibull de paramètres $\alpha =$ `Alpha` et $\lambda =$ `Lambda`, c'est-à-dire dont la fonction de répartition est

$$F(x) = 1 - e^{-\lambda x^{\alpha}}, \qquad x > 0.$$

On doit avoir $\alpha > 0$ et $\lambda > 0$. La moyenne est égale à

$$\frac{\Gamma(1 + 1/\alpha)}{\lambda^{1/\alpha}}, \qquad \alpha > 0.$$

Appelle la fonction `Uniform01` et utilise l'inversion.

```
PROCEDURE Gamma
   ( Alpha, Lambda : LONGREAL;
     g : Gen               (* Numéro du générateur utilisé. *)
   ): LONGREAL;
```

Fournit une valeur aléatoire suivant la loi Gamma de paramètres $\alpha =$ `Alpha` et $\lambda =$ `Lambda`, c'est-à-dire dont la fonction de densité est

$$f(x) = \frac{\lambda^{\alpha} x^{\alpha-1} e^{-\lambda x}}{\Gamma(\alpha)}, \qquad x > 0.$$

On doit avoir $\alpha > 0$ et $\lambda > 0$. La moyenne et la variance sont α/λ et α/λ^2 respectivement. Utilise les fonctions RGS et RGKM3 des figures L.12 et L.13 de [4]. Il s'agit d'une méthode d'acceptation-rejet, de sorte que le nombre d'appels à la fonction `Uniform01` est variable.

```
PROCEDURE Beta
   ( a, b : LONGREAL;
     g : Gen               (* Numéro du générateur utilisé. *)
   ): LONGREAL;
```

Fournit une valeur aléatoire suivant la loi Beta de paramètres $\alpha =$ `a` et $\beta =$ `b`, c'est-à-dire dont la fonction de densité est

$$f(x) = \frac{\Gamma(\alpha + \beta) x^{\alpha-1} (1 - x)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)}, \qquad 0 < x < 1.$$

On doit avoir $\alpha > 0$ et $\beta > 0$. Utilise l'algorithme de Cheng donné à la figure L.2 de [4]. Il s'agit d'une méthode d'acceptation-rejet, de sorte que le nombre d'appels à la fonction `Uniform01` est variable.

```
END RAND1.
```

SIM

Ce module fait l'interface avec l'exécutif du progiciel de simulation. Il fournit des procédures permettant d'initialiser, de démarrer ou d'arrêter la simulation, de même qu'une fonction qui retourne la valeur de l'horloge.

`DEFINITION MODULE SIM;`

`PROCEDURE Init;`

Réinitialise la simulation en vidant la liste des événements, si elle ne l'est pas déjà, et en remettant l'horloge à zéro. S'il y a des processus existants (voir module `PROCS`), on doit tous les détruire avant d'appeler cette procédure.

`PROCEDURE Start;`

Démarre l'exécutif de la simulation. Lors de son appel, il doit déjà y avoir au moins un événement ou un processus de prévu.

`PROCEDURE Stop;`

Arrête l'exécutif de la simulation et retourne le contrôle au programme principal (ou à l'une de ses procédures), juste après l'instruction `SIM.Start` qui a démarré cette simulation. A noter cependant que l'exécutif ne s'arrêtera que lorsqu'on lui aura tranféré le contrôle. Si on est en train d'exécuter un événement, ce sera dès la fin de la procédure d'événement, et si on est dans un processus, ce sera dès que ce processus appelle `Terminate` pour terminer son exécution ou passe le contrôle à l'exécutif en appelant `Delay` ou `Suspend` ou `Request` ou `Take`, etc.

`PROCEDURE Time () : LONGREAL;`

Retourne la valeur de l'instant courant de la simulation.

`END SIM.`

EVENT

Ce module offre les outils nécessaires pour effectuer une simulation avec vision par événements. Le type `EventType`, prédéfini dans ce module, correspond à un type d'événe- ment qui est associé à une procédure sans paramètre. Pour chaque type d'événement, l'usager doit déclarer une variable de type `EventType`, puis appeler `Create` pour lui associer sa procédure correspondante, qui sera exécutée à chaque occurence d'un événement de ce type. On peut prévoir un événement dans un délai donné, par la procédure `Schedule`. On donne alors le type de l'événement, le délai, et un pointeur sur les attributs (paramètres) associés à cet événement. On peut récupérer la valeur de ce pointeur, à l'intérieur de la procédure qui exécute l'événement en question, en appelant la fonction `Attrib`. La procédure `Cancel` permet d'annuler un événement déjà prévu, selon un attribut particulier. Pour chaque événement prévu, le système crée un avis d'événement (`EventNotice`) qui contient le type de l'événement (le `EventType`), son instant d'occurence (supérieur ou égal à l'instant courant) et un pointeur sur ses attributs, s'il en a.

Lorsqu'on veut prévoir un événement pour l'instant courant, et que l'on veut qu'il soit placé au tout début de la liste des événements, avant les autres déjà prévus pour le même instant s'il y a lieu, on appelle plutôt `ScheduleNext` au lieu de `Schedule`. Les procédures `ScheduleNotice` et `ScheduleNoticeNext` permettent de prévoir un événement tout en récupérant le pointeur sur l'avis d'événement. Cela peut permettre par exemple d'annuler ultérieurement cet avis d'événement en appelant `CancelNotice`. On peut aussi placer un avis d'événement juste avant ou juste après un autre avis qui se trouve déjà dans la liste (voir `ScheduleNoticeBefore` et `ScheduleNoticeAfter`). D'autres procédures permettent d'obtenir le type, le pointeur sur les attributs, et l'instant d'occurence d'un événement prévu, de retrouver un avis d'événement satisfaisant une condition particulière, ou de faire imprimer la liste des événements prévus.

```
DEFINITION MODULE EVENT;

FROM SYSTEM    IMPORT ADDRESS;

TYPE
   EventType;
```

Un type d'événement. Pour chaque type d'événement qu'il veut utiliser, l'usager doit déclarer une variable de ce type, puis appeler la procédure `Create`.

```
   EventNotice;
```

Un avis d'événement.

```
   CondProc = PROCEDURE (ADDRESS) : BOOLEAN;
```

Type de procédure utilisé comme paramètre dans la procédure `FindNoticeSuchThat`.

```
PROCEDURE Create
   ( VAR E : EventType;
     P     : PROC;
     Name  : ARRAY OF CHAR
   );
```

Associe la procédure `P` et l'identificateur `Name` au type d'événement `E`. Cette procédure doit être obligatoirement appelée pour chaque type d'événement. `Name` servira à identifier ce type d'événement si on fait imprimer la liste des événements. On recommande un maximum de 32 caractères.

```
PROCEDURE Schedule
   ( E : EventType;
     D : LONGREAL;
     Par : ADDRESS
   );
```

Prévoit un événement de type `E`, en insérant un avis dans la liste des événements. Son occurence est prévue pour l'instant "`SIM.Time() + D`", c'est-à-dire dans `D` unités de temps. L'usager peut associer à cet événement un bloc de paramètres (un attribut), sous forme d'un pointeur dont la valeur est donnée au paramètre formel `Par`. Il pourra récupérer la valeur de ce pointeur lors de l'exécution de l'événement, en utilisant la fonction `Attrib()`. `D` doit être non négatif, sinon une erreur est signalée et rien d'autre n'est effectué. Lorsque deux ou plusieurs événements sont prévus pour le même instant, ils sont placés dans la liste (et exécutés) dans l'ordre selon lequel ils ont été prévus.

```
PROCEDURE ScheduleNext
   ( E : EventType;
     Par : ADDRESS
   );
```

Tout comme `Schedule`, sauf que l'événement prévu est placé au tout début de la liste d'événements, et 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 avant eux.

```
PROCEDURE Cancel
   ( E : EventType;
     Par : ADDRESS
   );
```

Annule (enlève de la liste d'événements) le premier avis d'événement de type `E` et qui a été prévu avec la valeur `Par` pour ses attributs. La recherche s'effectue à partir du début de la liste des événements (par ordre chronologique). Si la liste d'événements est vide, ou que l'événement recherché est inexistant, rien n'est effectué.

```
PROCEDURE Attrib () : ADDRESS;
```

Retourne le pointeur sur les attributs de l'événement en cours d'exécution. Ce pointeur est la valeur du paramètre `Par` fourni lors de sa prévision. C'est à l'usager de s'assurer que ce résultat est affecté à une variable du même type (c'est-à-dire un pointeur sur le même type de structure) que la variable passée en paramètre lors de la prévision de cet événement. Attention : si l'usager appelle cette procédure ailleurs que dans une procédure exécutant un événement, les résultats sont imprévisibles (il est possible que l'on soit en train d'exécuter un événement défini par SIMOD...).

```
PROCEDURE ScheduleNotice
   ( E : EventType;
     D : LONGREAL;
     Par : ADDRESS;
     VAR N : EventNotice
   );
```

Tout comme `Schedule`, mais retourne l'avis d'événement associé.

```
PROCEDURE ScheduleNoticeNext
   ( E : EventType;
     Par : ADDRESS;
     VAR N : EventNotice
   );
```

Tout comme `ScheduleNext`, mais retourne l'avis d'événement associé.

```
PROCEDURE ScheduleNoticeBefore
   ( E : EventType;
     Par : ADDRESS;
     VAR N : EventNotice;
     N1    : EventNotice
   );
```

Tout comme `ScheduleNotice`, sauf que l'avis d'événement `N` est placé immédiatement avant l'avis `N1` dans la liste. `N1` doit être un avis d'événement déjà prévu. Cette procédure peut être utile par exemple pour placer un événement "espion" qui s'exécutera juste avant un autre événement donné.

```
PROCEDURE ScheduleNoticeAfter
   ( E : EventType;
     Par : ADDRESS;
     VAR N : EventNotice;
     N1    : EventNotice
   );
```

Tout comme `ScheduleNoticeBefore`, sauf que l'avis d'événement `N` est placé immédiatement après l'avis `N1` dans la liste.

```
PROCEDURE FindNoticeSuchThat
   ( E : EventType;
     Cond : CondProc
   ) : EventNotice;
```

Retourne l'avis d'événement associé au premier événement de type `E` dans la liste tel que la procédure `Cond`, appliquée au pointeur sur les attributs de cet événement, retourne `TRUE`. S'il n'y en a pas, cette procédure retourne `NIL`.

```
PROCEDURE CancelNotice
   ( VAR N : EventNotice
   );
```

Annule l'événement associé à l'avis d'événement `N`.

```
PROCEDURE NoticeType
   ( N : EventNotice
   ) : EventType;
```

Retourne le type de l'événement associé à l'avis d'événement `N`.

```
PROCEDURE NoticeAttrib
   ( N : EventNotice
   ) : ADDRESS;
```

Retourne le pointeur sur les attributs de l'événement associé à `N`. Ce pointeur est la valeur du paramètre `Par` fourni lors de sa prévision.

```
PROCEDURE NoticeTime
   ( N : EventNotice
   ) : LONGREAL;
```

Retourne l'instant (prévu) d'occurence de l'événement associé à `N`.

```
PROCEDURE PrintEventList;
```

Imprime la liste des événements dans le fichier de sortie courant.

```
PROCEDURE EmptyEventList;
```

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

```
PROCEDURE DeleteType
   ( VAR E : EventType
   );
```

Détruit le type d'événement `E` et récupère l'espace mémoire utilisé pour sa description.

```
END EVENT.
```

CONT

Ce module permet la simulation continue, pour laquelle l'évolution de certaines variables en fonction du temps est régie par des équations différentielles. Dans un même modèle, on peut mélanger des événements discrets, des processus, et des variables "continues" qui obéissent à des équations différentielles. Il doit cependant y avoir un nombre fini de telles variables, et pour chacune d'entre elles, on doit donner l'équation différentielle qui la régit, sous forme d'une fonction. Ce module ne permet pas de traiter les équations aux dérivées partielles, que l'on simule habituellement par des techniques d'éléments finis.

Pour chaque variable continue que l'on veut utiliser, on doit déclarer une variable de type `VarCont`, puis la créer en appelant `Create`. Lors de la création de la variable, on doit donner sa valeur initiale, et une fonction qui retourne sa dérivée en fonction du temps. L'évolution de la valeur de la variable se fait en intégrant cette fonction. Pour chaque variable, on peut démarrer l'intégration en appelant `StartInteg`, et la stopper en appelant `StopInteg`. Lors du démarrage, on peut aussi choisir un type d'événement qui sera exécuté à chaque pas d'intégration pour cette variable (juste après avoir effectué ce pas). Un tel événement pourrait par exemple vérifier si une variable continue a atteint un certain seuil, ou encore mettre à jour un graphique illustrant l'évolution du système. Avant de démarrer toute intégration, on doit appeler `SelectMethod` pour choisir la méthode et la longueur h du pas d'intégration. Le même h et la même méthode seront utilisés pour toutes les variables. Le pas d'intégration qui fait passer la variable de la valeur qu'elle a à l'instant $t - h$, à celle qu'elle doit avoir à l'instant t, se fait lorsque l'horloge de la simulation atteint l'instant t.

Lors de la création d'une variable, on peut aussi lui associer un paramètre (en général un pointeur), que l'on pourra ensuite récupérer en appelant `VarAttrib`. La fonction `CurrentVar`, lorsqu'appelée à l'intérieur de la fonction de type `Deriv` associée à une variable continue `V`, durant l'intégration, retourne `V`. En particulier, une combinaison de ces deux fonctions permet d'examiner le paramètre de la variable que l'on est en train d'intégrer. On peut réinitialiser la valeur d'une variable continue en appelant `Init`, et examiner sa valeur courante en appelant `Value`. Enfin, pour la détruire et récupérer l'espace qu'elle occupait, il suffit d'appeler `Delete`.

```
DEFINITION MODULE CONT;

FROM EVENT  IMPORT EventType;
FROM SYSTEM IMPORT ADDRESS;

TYPE

   VarCont;
```

Pour chaque variable continue que l'on veut utiliser, on doit déclarer une variable de ce type, puis appeler `Create` pour la créer.

```
   Deriv = PROCEDURE (LONGREAL) : LONGREAL;
```

Type de procédure à passer en paramètre dans `Create`.

```
Method = (Euler, RungeKutta2, RungeKutta4);
```

Une méthode d'intégration. `Euler`: la méthode d'Euler; `RungeKutta2`: la méthode d'Euler modifiée (Runge Kutta d'ordre 2); `RungeKutta4`: Runge Kutta d'ordre 4.

```
PROCEDURE Create
   ( VAR V : VarCont;
     Val   : LONGREAL;
     D     : Deriv;
     Par   : ADDRESS
   );
```

Crée la variable continue `V`, l'initialise à la valeur `Val`, et lui associe la procédure `D`. Cette procédure, appelée avec le paramètre `t`, doit retourner la dérivée de la valeur de la variable en fonction du temps, au temps `t`, en fonction des valeurs courantes des variables (continues ou autres) du modèle. Le paramètre `Par` est en général un pointeur sur un bloc d'information quelconque associé à cette variable (ou toute autre variable compatible avec `ADDRESS`). On pourra récupérer sa valeur en appelant `VarAttrib`.

```
PROCEDURE Init
   ( V   : VarCont;
     Val : LONGREAL
   );
```

Réinitialise la valeur de la variable `V` à `Val`.

```
PROCEDURE Value
   ( V : VarCont
   ) : LONGREAL;
```

Retourne la valeur courante de la variable `V`.

```
PROCEDURE VarAttrib
   ( V : VarCont
   ) : ADDRESS;
```

Retourne la valeur du paramètre associé à `V`, c'est-à-dire la valeur de `Par` fournie lors de sa création.

```
PROCEDURE CurrentVar
   () : VarCont;
```

Lorsqu'appelée à l'intérieur d'une procédure de type `Deriv` associée à une variable `V`, durant l'intégration, cette fonction retourne `V`. Sinon, retourne `NIL`.

```
PROCEDURE SelectMethod
   ( M : Method;
     h : LONGREAL
   );
```

Choisit la méthode d'intégration qui fait évoluer l'état des variables continues, et la longueur du pas d'intégration. Pour toutes les variables pour lesquelles l'intégration a été démarrée, la méthode d'intégration sera `M`, avec un pas d'intégration de `h` unités de temps.

```
PROCEDURE StartInteg
   ( V : VarCont;
     E : EventType;
     Par : ADDRESS
   );
```

Démarre le processus d'intégration qui fait évoluer l'état de la variable `V`. Juste après chaque pas d'intégration, un événement de type `E`, avec `Par` comme pointeur sur ses paramètres, sera exécuté. Si on ne veut pas d'un tel événement, on passe `EventType (NIL)` pour valeur de `E`.

```
PROCEDURE StopInteg
   ( V : VarCont
   );
```

Stoppe l'intégration qui avait été démarrée par `StartInteg` pour la variable `V`. La valeur de `V` demeurera celle calculée lors du dernier pas d'intégration effectué avant l'appel à `StopInteg`.

```
PROCEDURE Delete
   ( VAR V : VarCont
   );
```

Détruit la variable continue `V`, et récupère l'espace qu'elle occupait.

```
END CONT.
```

STAT

Le module `STAT` permet de recueillir des statistiques, d'obtenir des rapports et de calculer des intervalles de confiance. Il fournit un type prédéfini appelé `Block`, qui correspond à un bloc d'information pour le recueil de statistiques. Pour chaque quantité pour laquelle on désire recueillir des statistiques, on déclare une variable de type `Block`, puis on crée le bloc d'information correspondant en appelant `Create`. On doit distinguer deux types de blocs statistiques. On a un bloc de type `Tally` lorsqu'on s'intéresse à une séquence d'observations (de valeurs numériques) $X_1, X_2, X_3, \ldots$ d'une variable. Par ailleurs, lorsqu'on s'intéresse à l'évolution dans le temps de la valeur d'une variable, $X(t), t \geq 0$, on a un bloc de type `Accumulate`. Par exemple, pour estimer la longueur moyenne d'une file d'attente en fonction du temps, on utilisera un bloc de type `Accumulate`, tandis que pour estimer la durée moyenne d'attente par client, on utilisera un bloc de type `Tally`. Le type d'un bloc doit être déclaré lors de sa création.

Dans *SIMOD*, les blocs statistiques déclarés par l'usager ne sont pas mis à jour automatiquement par le système. L'usager doit appeler la procédure `Update` pour fournir la nouvelle valeur de X_i chaque fois qu'une nouvelle valeur est observée, dans le cas d'un bloc de type `Tally`, et pour donner la nouvelle valeur de $X(t)$ chaque fois que la valeur de la variable est modifiée, dans le cas d'un bloc de type `Accumulate`.

La procédure `Init` permet de réinitialiser, à n'importe quel moment, tous les compteurs associés à un bloc statistique. On peut donc réinitialiser sélectivement ces derniers au besoin. Lorsqu'on effectue plusieurs répétitions, par exemple, on peut utiliser un bloc statistique pour recueillir la distribution des moyennes des différentes répétitions, et réinitialiser seulement les autres blocs au début de chaque répétition. `Delete` permet d'éliminer un bloc statistique et d'en récupérer l'espace.

La procédure `Report` permet d'obtenir un rapport complet sur un bloc statistique. Lorsqu'on veut connaître seulement la valeur d'une statistique particulière, on peut faire appel aux fonctions `NumberObs`, `Minimum`, `Maximum`, `Sum`, `Average`, `Variance` et `StandardDev`. A noter cependant que le nombre d'observations, la variance et l'écart-type ne sont disponibles que pour les blocs de type `Tally`. La variance et l'écart-type n'ont d'utilité, en général, que lorsque les valeurs observées sont indépendantes (par exemple les moyennes de plusieurs répétitions). On peut aussi, sous ces mêmes restrictions, calculer un intervalle de confiance, de niveau spécifié, pour la moyenne d'une variable. Il suffit d'appeler la procédure `ConfidenceInterval` ou `ReportConfidenceInterval`. Ces procédures supposent cependant que les valeurs observées sont indépendantes et identiquement distribuées selon une loi normale. Les intervalles de confiance obtenus ne seront valides que si cette hypothèse est satisfaite, ou n'est pas trop loin de l'être. Pour un bloc de type `Accumulate`, tout appel à l'une des procédures mentionnées dans ce paragraphe provoque une mise-à-jour automatique (par un appel à `Update`).

```
DEFINITION MODULE STAT;

TYPE
   Block;
```

Bloc statistique renfermant toutes les informations nécessaires pour le recueil des statistiques concernant une variable donnée. Chaque bloc statistique que l'on veut utiliser doit être identifié par une variable de type `Block`, dont l'identificateur servira de nom au bloc statistique en question, puis créé à l'aide de la procédure `Create`.

```
   BlockType = (Tally, Accumulate);
```

Sorte de bloc statistique.

```
PROCEDURE Create
   ( VAR X : Block;
     Kind  : BlockType;
     Name  : ARRAY OF CHAR
   );
```

Crée le bloc statistique `X`, dont le type est spécifié par `Kind`. Alloue l'espace et initialise tous les compteurs associés à ce bloc. Le paramètre `Name` permet de donner un nom à `X` sous forme d'une chaîne de caractères. Ce nom sera utilisé lorsqu'on fera imprimer un rapport pour `X`. On recommande un maximum de 32 caractères.

```
PROCEDURE Init
   ( X : Block
   );
```

Initialise ou réinitialise tous les compteurs associés au bloc statistique `X`. Le bloc `X` doit avoir été créé auparavant. A noter que dans le cas où `X` est de type `Accumulate`, `Init` ne modifie pas la valeur courante de la variable. Cette valeur courante est toujours celle du dernier appel à `Update`. Pour l'initialiser, la modifier, ou la remettre à 0, il faut appeler `Update`. Habituellement, un appel à `Init` pour un bloc de type `Accumulate` devrait être immédiatement suivi d'un appel à `Update`.

```
PROCEDURE Update
   ( X : Block;
     V : LONGREAL
   );
```

Met à jour tous les compteurs associés au bloc statistique `X`. Si `X` est de type `Tally`, une nouvelle observation de la variable, de valeur `V`, est ajoutée. Si `X` est de type `Accumulate`, cette mise a jour indique qu'à partir de l'instant courant, la variable associée au bloc `X` prend la valeur `V`.

```
PROCEDURE Report
   ( X : Block
   );
```

Imprime, dans le fichier de sortie (habituellement à l'écran), un rapport complet sur la variable associée au bloc statistique `X`, depuis la dernière initialisation de ce bloc.

```
PROCEDURE NumberObs
   ( X : Block                    (* Bloc statistique de type Tally. *)
   ): LONGCARD ;
```

Pour une statistique de type `Tally`, fournit le nombre de valeurs observées pour la variable associée au bloc statitique `X` depuis la dernière initialisation de ce bloc. Pour une statistique de type `Accumulate`, imprime un message d'erreur.

```
PROCEDURE Minimum
   ( X : Block
   ): LONGREAL;
```

Fournit la plus petite valeur qu'a prise la variable associée au bloc statistique `X` depuis la dernière initialisation de ce bloc.

```
PROCEDURE Maximum
   ( X : Block
   ): LONGREAL;
```

Fournit la plus grande valeur qu'a prise la variable associée au bloc statistique `X` depuis la dernière initialisation de ce bloc.

```
PROCEDURE Sum
   ( X : Block
   ): LONGREAL;
```

Pour une statistique de type `Tally`, fournit la somme des valeurs prises par la variable associée au bloc `X` depuis sa dernière initialisation. Pour une statistique de type `Accumulate`, fournit l'intégrale de la valeur de la variable, par rapport au temps, depuis la dernière initialisation du bloc `X` jusqu'à l'instant courant.

```
PROCEDURE Average
   ( X : Block
   ): LONGREAL;
```

Pour une statistique de type `Tally`, fournit la moyenne des observations de la variable associée au bloc `X` (somme des valeurs divisée par le nombre d'observations) depuis sa dernière initialisation. Pour une statistique de type `Accumulate`, fournit la valeur moyenne de la variable, par unité de temps, depuis la dernière initialisation du bloc.

```
PROCEDURE Variance
   ( X : Block                    (* Bloc statistique de type Tally. *)
   ): LONGREAL;
```

Pour une statistique de type `Tally`, fournit la variance des observations de la variable associée au bloc `X` (somme des carrés des écarts à la moyenne, divisée par le nombre d'observations moins 1) depuis sa dernière initialisation. Si le nombre d'observations est inférieur à 2, ou si `X` est de type `Accumulate`, un message d'erreur sera imprimé. *Attention*: L'implantation utilise la double précision, mais il peut y avoir une erreur numérique significative lorsque la variance est *très* petite par rapport à la moyenne.

```
PROCEDURE StandardDev
   ( X : Block                 (* Bloc statistique de type Tally. *)
   ): LONGREAL;
```

Pour une statistique de type `Tally`, fournit l'écart-type des observations. Appelle la fonction `Variance` et extrait la racine carrée.

```
PROCEDURE ConfidenceInterval
   ( X      : Block;           (* Bloc statistique de type Tally.   *)
     Level  : LONGREAL;            (* Niveau de confiance.              *)
     VAR Center, Radius : LONGREAL  (* Milieu et rayon de l'intervalle.  *)
   );
```

Pour une statistique de type `Tally`, fournit le point milieu et le rayon d'un intervalle de confiance, de niveau spécifié par l'usager, pour la vraie moyenne μ de la variable associée à `X`. Cet intervalle est calculé en utilisant la statistique :

$$T = \frac{\overline{X} - \mu}{S_x/\sqrt{n}}$$

où n est le nombre d'observations de la variable, $\overline{X}$ = `Average(X)` est la moyenne échantillonnale, et S_x = `StandardDev(X)` est l'écart-type échantillonnal. Si on suppose que les observations de la variable sont indépendantes et identiquement distribuées suivant une loi normale, alors la statistique T suit la loi de Student à $n-1$ degrés de liberté. Lorsque cette hypothèse n'est pas vérifiée, les valeurs fournies peuvent avoir une valeur statistique doûteuse. Si $n < 2$, ou si la statistique `X` est de type `Accumulate`, un message d'erreur sera imprimé.

```
PROCEDURE ReportConfidenceInterval
   ( X      : Block;           (* Bloc statistique de type Tally.   *)
     Level  : LONGREAL             (* Niveau de confiance.              *)
   );
```

Pour une statistique de type `Tally`, imprime un intervalle de confiance, de niveau spécifié par l'usager, pour la vraie moyenne μ de la variable associée à `X`. Cet intervalle est calculé en utilisant la procédure `ConfidenceInterval`.

```
PROCEDURE Delete
   ( VAR X : Block
   );
```

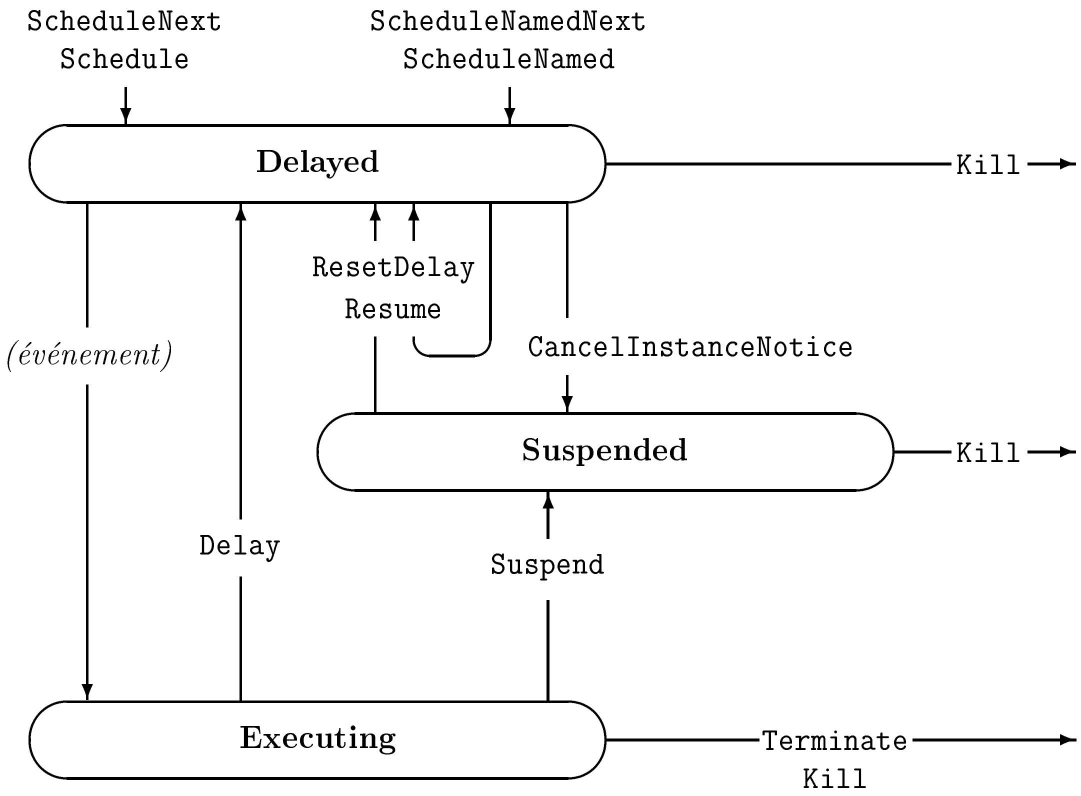
Détruit le bloc statistique `X` et récupère l'espace mémoire qu'il utilisait.

```
END STAT.
```

PROCS

Ce module fournit les outils de base pour la simulation avec vision par processus. Le type `ProcessType` correspond à un type de processus. Pour chaque type de processus, l'usager doit déclarer une variable de type `ProcessType`, puis appeler `Create`, pour lui associer une procédure (sans paramètres) et déclarer la taille de l'espace mémoire requis à l'exécution de sa coroutine. Un type de processus correspond habituellement à un type d'objet dans le modèle, et la procédure (coroutine) associée décrit le comportement de cet objet.

La procédure `Schedule` permet de prévoir l'activation d'un processus d'un type donné, dans un délai donné. Cette procédure place un avis d'événemment (`EventNotice`) dans la liste d'événements, et c'est cet événement qui déclenchera l'exécution du processus. Lors de la prévision, on donne le type du processus, le délai, et un pointeur sur les attributs (les paramètres) que l'on veut associer à ce processus. La valeur de ce pointeur peut être récupérée, durant l'exécution du processus en question, en appelant la fonction `Attrib`. Habituellement, les processus sont anonymes, c'est-à-dire que les processus d'un même type (il peut y en avoir plusieurs qui évoluent simultanément) n'ont pas de noms qui les distinguent. Lorsqu'on veut donner un nom spécifique (un identificateur) à un processus, on doit d'abord déclarer un identificateur de type `ProcessInstance`, puis prévoir l'activation de ce processus en appelant `ScheduleNamed` au lieu de `Schedule`. Ce processus devient alors *baptisé*, et on pourra y référer par l'identificateur en question. Lorsqu'on veut qu'un processus débute son exécution à l'instant courant, et que l'événement déclenchant cette exécution soit placé au tout début de la liste d'événements, avant les autres déjà prévus pour le même instant s'il y a lieu, on appelle plutôt `ScheduleNext` ou `ScheduleNamedNext` pour le prévoir (l'effet équivalent peut être obtenu en appelant `ScheduleNamed` suivi de `Resume` pour le processus en question). Chaque processus reçoit aussi un numéro de série, qui lui est donné lors de sa prévision: le n-ième processus prévu reçoit le numéro n. On peut connaître le numéro de série d'un processus en appelant `InstanceSerialNumber`. L'intérêt est que ce numéro est unique: deux processus ne peuvent pas avoir le même numéro (à moins que l'on génère un plus grand nombre de processus que le plus grand entier représentable sur l'ordinateur). Par contre, l'identificateur de type `ProcessInstance` est en fait un pointeur sur le bloc d'information associé à un processus. Deux processus qui existent simultanément ne peuvent pas avoir des identificateurs de même valeur, mais un processus prévu après qu'un autre processus se soit terminé peut avoir un identificateur de même valeur que ce dernier.



Un processus existant peut se trouver dans l'un des trois états suivants: `Executing`, `Delayed` ou `Suspended` (voir le diagramme ci-haut). Il n'y a jamais plus d'un seul processus à la fois qui se trouve dans l'état `Executing`, et lorsqu'il y en a un, c'est celui-ci qui est en exécution (qui est en train d'exécuter ses instructions). On dit aussi qu'il a le contrôle. Lorsque ce n'est pas un processus de l'usager qui a le contrôle (par exemple lorsqu'on exécute un événement), on dit que c'est le "processus principal" (ou encore l'exécutif de la simulation) qui l'a. Un processus est dans l'état `Delayed` s'il y a un événement de prévu pour lui redonner le contrôle. 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.

Lorsqu'on prévoit son activation, un processus passe à l'état `Delayed`. Il passe à l'état `Executing` lorsqu'il débute ou reprend son exécution. Le processus en exécution peut passer à l'état `Suspended` en appelant `Suspend` (ou en appelant une procédure qui appelle elle-même `Suspend`, comme c'est le cas dans les modules `RES`, `BIN`, etc.). Dans ce cas, on exécutera ensuite le prochain événement de la liste (souvent, cet événement redonnera le contrôle à un autre processus). Le processus en exécution peut aussi appeler `Delay` afin de passer à l'état `Delayed`, et attendre que l'horloge de la simulation ait avancé d'un certain nombre d'unités de temps avant de reprendre son exécution. Cette procédure place dans la liste d'événements un avis d'événement qui redonnera le contrôle à ce processus. On peut modifier par la suite l'instant d'occurence de cet événement en appelant `ResetDelay`. On peut aussi appeler `ResetDelay` pour faire passer un processus de l'état `Suspended` à l'état `Delayed`, et prévoir un événement qui lui redonnera éventuellement le contrôle. Pour faire passer un processus de l'état `Delayed` à l'état `Suspended`, il suffit d'appeler `CancelInstanceNotice`. Par ailleurs, si on veut que l'événement redonnant le contrôle à un processus soit placé au tout début de la liste, on peut appeler `Resume`.

D'autres procédures permettent de connaître le type d'un processus, son état, le pointeur sur ses attributs, l'avis d'événement prévu qui lui donnera ou redonnera le contrôle (s'il y a lieu), et le temps qui reste avant l'exécution de cet l'événement. La procédure `CurrentProcess` permet aussi de connaître le processus présentement en cours d'exécution.

La vie d'un processus se termine lorsqu'il appelle la procédure `Terminate`, ou encore lorsqu'on appelle `Kill` pour le détruire de l'extérieur. On peut aussi détruire d'un seul coup tous les processus. A noter qu'un processus doit toujours se terminer avant d'atteindre le `END` final de sa procédure associée, sinon le programme s'arrêtera lorsque ce `END` sera atteint.

```
DEFINITION MODULE PROCS;

FROM EVENT  IMPORT EventNotice;
FROM SYSTEM IMPORT ADDRESS;

TYPE
   ProcessType;
```

Un type de processus. Pour chaque type de processus qu'il veut utiliser, l'usager doit déclarer une variable de ce type, puis appeler la procédure `Create`.

```
   ProcessInstance;
```

Lorsqu'on veut donner un nom spécifique (identificateur) à un processus, c'est-à-dire à un "exemplaire" d'un type de processus, on utilise une variable de ce type comme identificateur. On associe cet identificateur au processus lorsqu'on prévoit son activation par `ScheduleNamed` ou `ScheduleNamedNext`. A noter que le type `ProcessInstance` est différent du type `PROCESS` prédéfini dans Modula-2.

```
   ProcessState = (Executing, Delayed, Suspended);
```

Liste des différents états dans lesquels un processus peut se trouver : en exécution (`Executing`), retardé pour une durée connue, avec un événement prévu pour lui redonner le contrôle (`Delayed`), en attente qu'on lui redonne le contrôle, sans qu'il y ait d'événement de prévu à cet effet (`Suspended`).

```
PROCEDURE Create
   ( VAR T : ProcessType;
     P     : PROC;
     Size  : CARDINAL;
     Name  : ARRAY OF CHAR
   );
```

Associe la procédure `P` au type de processus `T`, et spécifie l'espace mémoire (`Size`) requis pour chaque processus de ce type. Cette procédure est obligatoire pour chaque type de processus. Elle ne crée pas de processus, mais seulement un type de processus. La taille de l'espace mémoire est spécifiée en octets, et doit être suffisante pour l'exécution de la procédure `P`, pour ses variables locales, de même que pour tout l'espace mémoire (pile d'exécution) requis par les procédures qu'elle appelle directement ou indirectement. `Name` est une chaîne de caractères identifiant ce type de processus. Seuls les 32 premiers caractères sont retenus.

```
PROCEDURE Schedule
   ( T : ProcessType;
     D : LONGREAL;
     Par : ADDRESS
   );
```

Crée un nouveau processus (un nouvel exemplaire) de type `T`, et prévoit (par un appel à `EVENT.Schedule`) un événement qui donnera le contrôle à ce processus, dans `D` unités de temps. Ce processus débutera donc son exécution à l'instant "`SIM.Time() + D`". En attendant, il est dans l'état `Delayed`. L'usager peut associer à ce processus un bloc de paramètres (un "attribut"), sous forme d'un pointeur dont la valeur est donnée au paramètre `Par`. Il pourra récupérer la valeur de ce pointeur lorsque ce processus sera en exécution, en utilisant la fonction `Attrib()`. Lorsqu'on n'a pas besoin de passer une telle information, on passe `NIL` pour la valeur de `Par`. La valeur de `D` doit être non négative, sinon une erreur est signalée et rien d'autre n'est effectué.

```
PROCEDURE ScheduleNamed
   ( T     : ProcessType;
     D     : LONGREAL;
     Par   : ADDRESS;
     VAR P : ProcessInstance
   );
```

Identique à `Schedule`, à la seule différence près que l'usager donne ici le nom spécifique `P` au processus prévu. Ce processus devient donc baptisé et pourra être référencé par son nom (c'est-à-dire par la valeur du paramètre `P`) ailleurs dans le programme.

```
PROCEDURE ScheduleNext
   ( T : ProcessType;
     Par : ADDRESS
   );
```

Tout comme `Schedule`, sauf que l'avis d'événement pour démarrer le processus est placé au tout début de la liste d'événements. Ce processus doit démarrer à l'instant courant. S'il y a d'autres événements prévus pour l'instant courant, l'avis de démarrage du processus sera placé avant eux.

```
PROCEDURE ScheduleNamedNext
   ( T     : ProcessType;
     Par   : ADDRESS;
     VAR P : ProcessInstance
   );
```

Identique à `ScheduleNext`, à la seule différence près que l'usager donne ici le nom spécifique `P` au processus prévu, tout comme dans `ScheduleNamed`.

```
PROCEDURE Attrib () : ADDRESS;
```

Retourne le pointeur sur les attributs du processus en cours d'exécution. Ce pointeur est la valeur du paramètre `Par` fourni lors de sa prévision. C'est à l'usager de s'assurer que ce résultat est affecté à une variable du même type (c'est-à-dire un pointeur sur le même type de structure) que la variable passée en paramètre lors de la prévision de ce processus.

```
PROCEDURE Delay
   ( D : LONGREAL
   );
```

Indique que l'horloge de la simulation (la valeur de `SIM.Time`) doit avancer de `D` unités de temps avant que la prochaine instruction du processus courant s'exécute. Cette procédure arrête l'exécution du processus courant, le place dans l'état `Delayed`, et prévoit (par un avis d'événement) son redémarrage dans D unités de temps.

```
PROCEDURE ResetDelay
   ( P : ProcessInstance;
     D : LONGREAL
   );
```

L'événement prévu qui doit redonner le contrôle au processus `P` (voir la procédure `InstanceNotice`, plus bas) est déplacé dans le temps de façon à ce que son exécution soit maintenant prévue dans `D` unités de temps à partir de l'instant courant. Si un tel événement n'existe pas, on en crée un et le processus passe dans l'état `Delayed`, sauf s'il s'agit du processus en exécution, auquel cas on imprime un message d'erreur. Par exemple, si le processus `P` a appelé la procédure `Delay` à l'instant $t = 10.0$ avec `D = 5.0`, puis à $t = 13.5$, un autre processus appelle `ResetDelay (P, 6.2)`, le processus `P` ne reprendra son exécution qu'à l'instant 19.7.

```
PROCEDURE CancelInstanceNotice
   ( P : ProcessInstance
   );
```

Annule l'avis d'événement qui doit redonner le contrôle au processus `P`, et place ce dernier dans l'état `Suspended`. Si un tel avis n'existe pas, c'est-à-dire si le processus n'est pas dans l'état `Delayed`, imprime un message d'erreur.

```
PROCEDURE Suspend;
```

Le processus appelant cette procédure laisse le contrôle au processus principal, et passe dans l'état `Suspended`.

```
PROCEDURE Resume
   ( P : ProcessInstance
   );
```

Place un événement, au tout début de la liste d'événements (voir `ScheduleNext`), qui fera passer le contôle au processus `P`. Ce dernier doit être dans l'état `Delayed` ou `Suspended`.

```
PROCEDURE Terminate;
```

Indique la fin normale de la vie d'un processus. Cette procédure doit être obligatoirement appelée à la fin de la procédure associée à un type de processus, à moins que les processus de ce type soient détruits de l'extérieur avant la fin de l'exécution de leur procédure.

```
PROCEDURE CurrentProcess
   () : ProcessInstance;
```

Retourne le processus qui est en train de s'exécuter.

```
PROCEDURE InstanceType
   ( P : ProcessInstance
   ) : ProcessType;
```

Retourne le type du processus `P`.

```
PROCEDURE InstanceState
   ( P : ProcessInstance
   ) : ProcessState;
```

Retourne l'état dans lequel se trouve actuellement le processus `P`.

```
PROCEDURE InstanceAttrib
   ( P : ProcessInstance
   ) : ADDRESS;
```

Retourne le pointeur sur les attributs du processus `P`.

```
PROCEDURE InstanceNotice
   ( P : ProcessInstance
   ) : EventNotice;
```

Retourne l'avis d'événement prévu qui devrait redonner le contrôle au processus `P` (c'est-à-dire qui marquera le début ou la poursuite de son exécution). Si un tel événement n'existe pas pour ce processus, retourne `NIL`.

```
PROCEDURE InstanceDelay
   ( P : ProcessInstance
   ) : LONGREAL;
```

Retourne le temps qu'il reste avant l'exécution de l'avis d'événement retourné par la procédure `InstanceNotice` (voir la procédure précédente). Si cet événement n'existe pas, imprime un message d'erreur.

```
PROCEDURE InstanceSerialNumber
   ( P : ProcessInstance
   ) : LONGCARD;
```

Retourne le numéro de série du processus `P`.

```
PROCEDURE Kill
   ( VAR P : ProcessInstance
   );
```

Détruit le processus `P`. Si `P` est le processus en cours d'exécution, cela équivaut à appeler `Terminate`. A noter que dans le cas ou le processus possède des paramètres (valeur de `Par` lors de la prevision), et que `Par` est un pointeur sur une structure de données quelconque, l'espace occupé par cette structure n'est pas recupéré par le système, car celui-ci ne la connaît pas. De plus, si `P` se trouve dans une ou plusieurs listes lorsqu'on le détruit (à part la liste d'événements et certaines listes gérées par le système), le contenu des listes en question n'est pas modifié. En fait le système ne mémorise par les noms des listes dans lesquelles se trouve un processus, et c'est à l'usager de s'en occuper.

```
PROCEDURE KillAll;
```

Détruit tous les processus existants (par des appels à `Kill`. A noter qu'il faut éviter d'appeler cette procédure à l'intérieur d'un module qui risque de faire partie d'un système contenant plusieurs modules, car si les autres modules contiennent des processus, ceux-ci seront aussi détruits.

```
PROCEDURE DeleteType
   ( VAR T : ProcessType
   );
```

Détruit le type de processus `T` et récupère l'espace mémoire utilisé pour sa description.

```
END PROCS.
```

GENEP

Ce module fournit des outils permettant de créer, démarrer ou stopper des mécanismes automatiques pour générer une suite d'événements ou de processus d'un certain type. Cette génération peut se faire selon un processus de Poisson, ou plus généralement selon un processus de renouvellement quelconque.

Pour définir un tel mécanisme d'activation, l'usager doit déclarer une variable de type `ArrivalProcess`, puis appeler `Create`. Les procédures `StartArrivalsE`, `StartPoissonArrivalsE`, `StartArrivalsP`, `StartPoissonArrivalsP` et `StopArrivals` permettent de démarrer ou de stopper ce mécanisme d'arrivées. `Delete` permet de le détruire et d'en récupérer l'espace.

La fonction `NbArriv` permet de connaître le nombre d'arrivées qui se sont produites depuis le dernier démarrage du mécanisme. Si on veut que des statistiques soient recueillies automatiquement sur les durées entre les arrivées successives, il suffit d'appeler `CollectStat` avant d'activer le mécanisme. La fonction `StatDelay` retournera alors le bloc statistique sur ces durées. La fonction `Report` permet d'obtenir un rapport statistique.

```
DEFINITION MODULE GENEP;

FROM EVENT  IMPORT EventType;
FROM PROCS  IMPORT ProcessType;
FROM RAND   IMPORT Gen;
FROM STAT   IMPORT Block;
FROM SYSTEM IMPORT ADDRESS;
TYPE
   ArrivalProcess;
```

Un mécanisme de génération d'arrivées ou de processus. Pour chaque mécanisme qu'il veut utiliser, l'usager doit déclarer une variable de ce type, puis appeler `Create`.

```
   DelayProc = PROCEDURE () : LONGREAL;
```

Type utilisé en paramètre dans `StartArrivals`.

```
PROCEDURE Create
   ( VAR A : ArrivalProcess;
     Name  : ARRAY OF CHAR
   );
```

Crée le mécanisme `A` de génération d'événements ou de processus, et lui associe le descripteur `Name`.

```
PROCEDURE StartArrivalsE
   ( A    : ArrivalProcess;
     E    : EventType;
     Par  : ADDRESS;
     D    : DelayProc;
     Nmax : LONGCARD
   );
```

Démarre le mécanisme d'arrivées `A`. Le temps jusqu'à la première arrivée, de même que les laps de temps entre les arrivées successives, sont des variables aléatoires indépendantes, identiquement distribuées, dont les valeurs sont générées en utilisant la fonction `D`, fournie par l'usager. Chaque "arrivée" correspond à l'exécution d'un événement de type `E`. Tous ces événements auront la valeur `Par` comme pointeur sur leur bloc de paramètres (c'est-à-dire leurs attributs). Si `Nmax` > 0, le générateur s'arrêtera après avoir généré `Nmax` arrivées, à moins que l'on appelle `StopArrivals` avant que ces `Nmax` arrivées ne se soient produites. Si `Nmax` $= 0$, c'est comme si on avait `Nmax` égal à l'infini.

```
PROCEDURE StartPoissonArrivalsE
   ( A    : ArrivalProcess;
     E    : EventType;
     Par  : ADDRESS;
     Mean : LONGREAL;
     g    : Gen;
     Nmax : LONGCARD
   );
```

Identique à `StartArrivalsE`, sauf que les durées entre les arrivées sont générées selon la loi exponentielle de moyenne `Mean`, en utilisant le générateur numéro `g`. Ainsi, les arrivées se font selon un processus de Poisson de taux `1/Mean`.

```
PROCEDURE StartArrivalsP
   ( A    : ArrivalProcess;
     T    : ProcessType;
     Par  : ADDRESS;
     D    : DelayProc;
     Nmax : LONGCARD
   );
```

Identique à `StartArrivalsE`, sauf que chaque "arrivée" active une nouvelle instance d'un processus de type `T`.

```
PROCEDURE StartPoissonArrivalsP
   ( A    : ArrivalProcess;
     T    : ProcessType;
     Par  : ADDRESS;
     Mean : LONGREAL;
     g    : Gen;
     Nmax : LONGCARD
   );
```

Identique à `StartPoissonArrivalsE`, sauf que chaque "arrivée" active une nouvelle instance d'un processus de type `T`.

```
PROCEDURE StopArrivals
   ( A : ArrivalProcess
   );
```

Arrête le mécanisme de génération `A`. Ce mécanisme devait avoir été démarré auparavant en utilisant l'une des quatre procédures précédentes. On pourra le redémarrer en utilisant aussi l'une de ces procédures.

```
PROCEDURE CollectStat
   ( A : ArrivalProcess
   );
```

Déclenche un mécanisme automatique de recueil de statistiques pour le processus d'arrivées `A`.

Un bloc statistique de type `Tally`, qui échantillonne les durées entre les arrivées successives, est créé et initialisé. C'est ce bloc qui est retourné par la fonction `StatDelay`. Il est réinitialisé automatiquement lorsqu'on redémarre le processus d'arrivées. La durée entre deux arrivées est recueillie au moment où la seconde arrivée se produit, et seulement les durées qui se terminent après l'appel à `CollectStat` seront recueillies.

```
PROCEDURE StatDelay
   ( A : ArrivalProcess
   ) : Block;
```

Retourne le bloc statistique qui échantillonne les durées entre les arrivées successives pour le processus d'arrivées `A`. Il s'agit d'un bloc de type `Tally`, qui n'existe que si on a déjà appelé `CollectStat` pour `A`.

```
PROCEDURE NbArriv
   ( A : ArrivalProcess
   ) : LONGCARD;
```

Retourne le nombre d'arrivées qui se sont produites pour `A` depuis son dernier démarrage.

```
PROCEDURE Report
   ( A : ArrivalProcess
   );
```

Fournit un rapport statistique sur `A`. Il faut d'abord avoir appelé `CollectStat` pour pouvoir utiliser cette fonction.

```
PROCEDURE Delete
   ( VAR A : ArrivalProcess
   );
```

Détruit le processus d'arrivées `A` et récupère l'espace qu'il utilisait.

```
END GENEP.
```

LIST

`LIST` est un module de gestion de listes doublement chaînées. Le type `List` est prédéfini dans le module. Un objet de ce type est constitué d'une tête de liste, contenant l'information générale sur cette liste, et peut contenir zéro ou plusieurs objets.

Ce module gère chaque liste sans connaître le type des objets qui s'y trouvent. Les "objets" insérés ou retirés des listes sont de type `ADDRESS`, et `LIST` ne manipule que ces pointeurs. On peut construire des listes contenant à peu près n'importe quoi, comme par exemple des processus, des blocs statistiques, des ressources, ou même d'autres listes. Il peut y avoir plusieurs types d'objets dans une même liste, et un objet peut se trouver dans plusieurs listes à la fois. En pratique, les types des objets d'une liste sont déclarés dans le programme de l'usager et sont en général des pointeurs sur des structures de données de types biens spécifiques. L'usager doit donc être prudent, et s'assurer que chaque fois qu'un objet est retiré d'une liste, il est placé dans une variable du même type (c'est-à-dire le même type de pointeur) que celle utilisée lors de son insertion dans la liste.

Pour chaque liste qu'il veut utiliser, l'usager doit déclarer une variable de type `List`, puis créer cette liste. Une liste peut être ordonnée ou bien selon la façon dont les objets y sont insérés, ou encore selon une procédure d'ordonnancement des objets fournie par l'usager lors de sa création. Le choix de l'un ou l'autre de ces deux modes d'ordonnancement est fixé une fois pour toutes lors de la création de la liste, et dépend de la procédure appelée pour effectuer cette création: `Create` et `CreateOrd` respectivement. Dans le premier cas, on dit (par un léger abus de langage) qu'on a une liste de type *non ordonné*, alors que dans le second cas, on dit que la liste est de type *ordonné*.

Pour insérer des objets dans une liste, on utilise l'une des procédures `Insert` ou `InsertOrd`, selon le type de la liste. La fonction `Size` permet de connaître le nombre d'objets qui se trouvent dans une liste, et la fonction `Belongs` permet de savoir si un objet particulier s'y trouve.

Pour chaque liste, le système mémorise la position du dernier objet référencé ou inséré. Cet objet s'appelle l'objet courant. Les procédures `Remove` et `View` permettent de retirer un objet d'une liste, ou de consulter (observer) un objet d'une liste sans le retirer de la liste. 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. Cela permet, par exemple, de parcourir une liste pour y trouver un objet ayant une propriété particulière. On peut aussi, lorsqu'on connaît le nom d'un objet (c'est-à-dire la valeur du pointeur associé), utiliser la procédure `RemoveObject` pour le retirer d'une liste.

D'autres procédures permettent d'unir deux listes, de trier une liste, ou de la diviser en deux. `Concatenate` met bout-à-bout deux listes non ordonnées pour en faire une seule liste, tandis que `Merge` fusionne en une seule liste deux listes ordonnées selon la même fonction d'ordonnacement. Une liste ordonnée peut aussi être réordonnée, par la procédure `Sort`, selon une nouvelle fonction fournie par l'usager. Enfin, il est aussi possible, par la procédure `Split`, de diviser une liste en deux nouvelles listes au niveau de l'objet courant.

SIMOD recueille des statistiques automatiquement sur les listes, mais seulement si on le lui demande (cela permet d'accélérer l'exécution dans les cas où ce n'est pas nécessaire). Pour chaque liste pour laquelle on le désire, il suffit d'appeler la procédure `CollectStat`, et le système lui associe deux blocs statistiques, mis-à-jour automatiquement. L'un de ces blocs est de type `Accumulate`, et sert à mesurer l'évolution de la taille de la liste en fonction du temps, tandis que l'autre est de type `Tally`, et échantillonne les durées de séjour des objets dans la liste. Les fonctions `StatSize` et `StatSojourn` retournent ces blocs statistiques, ce qui permet d'appeler les fonctions du module `STAT` pour examiner certaines statistiques particulières. La procédure `Report` fournit par ailleurs un rapport statistique complet sur une liste donnée, et `InitStat` permet de réinitialiser les deux blocs statistiques associés à une liste.

La procédure `Init` vide une liste de tous ses objets et réinitialise sa tête (y compris les blocs statistiques associés, s'il y a lieu), et `Delete` détruit une liste et récupère l'espace occupé par la tête de liste. A noter cependant que ces procédures ne détruisent pas les objets créés et gérés par l'usager, c'est-à-dire ceux qui correspondent aux pointeurs qui se trouvent dans la liste créée ou détruite. L'usager peut donc réutiliser ces objets à d'autres fins. En général, lorsqu'on effectue plusieurs répétitions d'une simulation, on doit vider la plupart des listes après chaque répétition. On peut utiliser `Init` si les objets qui s'y trouvent n'ont pas à être détruits parce qu'ils peuvent être réutilisés d'une répétition à l'autre. Lorsque les objets qui s'y trouvent doivent être détruits pour en récupérer l'espace, l'usager doit alors les retirer de la liste un à un pour les détruire.

```
DEFINITION MODULE LIST;

FROM STAT   IMPORT Block;
FROM SYSTEM IMPORT ADDRESS;

TYPE
   List;
```

Pour chaque liste qu'il veut utiliser, l'usager doit déclarer une variable de type `List`, puis créer cette liste.

```
   OutPosition = (First, Last, Previous, Next, Current);
```

Type de position où l'on peut observer ou retirer un objet dans une liste.

```
   InPosition = [First..Next];
```

Type de position où l'on peut insérer un objet dans une liste non ordonnée.

```
   OrdProc = PROCEDURE (ADDRESS, ADDRESS) : BOOLEAN;
```

Type de procedure utilisé comme paramètre dans les procédures `CreateOrd` et `Sort`.

```
   CondProc = PROCEDURE (ADDRESS) : BOOLEAN;
```

Type de procedure utilisé comme paramètre dans la procédure `FindFirstSuchThat`.

```
ActionProc = PROCEDURE (ADDRESS);
```

Type de procedure utilisé comme paramètre dans la procédure `ForEachInListDo`.

```
PROCEDURE Create
   ( VAR  L : List;
     Name   : ARRAY OF CHAR
   );
```

Crée une liste `L`, de type non ordonné, initialement vide. On pourra y insérer des objets en utilisant la procédure `Insert`. Le paramètre `Name` sert à identifier la liste lors des traces et des rapports. On recommande que le nombre de caractères ne dépasse pas 32.

```
PROCEDURE CreateOrd
   ( VAR  L  : List;
     Precede : OrdProc;
     Name    : ARRAY OF CHAR
   );
```

Crée une liste `L`, de type ordonné, initialement vide, et dans laquelle on pourra insérer des objets en utilisant la procédure `InsertOrd`. Cette liste sera maintenue en ordre automatiquement, selon la fonction `Precede` fournie par l'usager. On peut passer directement l'identificateur de notre fonction comme argument en autant qu'elle soit compatible avec la définition du type `OrdProc`. Cette fonction doit retourner la valeur `TRUE` si et seulement si l'objet `A` doit précéder l'objet `B` dans la liste. Pour le paramètre `Name`, voir `Create`.

```
PROCEDURE CollectStat
   ( L : List
   );
```

Déclenche un recueil automatique de statistiques pour la liste `L` (déjà créée). Lorsqu'on appelle cette procédure, deux blocs statistiques sont créés et initialisés. 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. Ce dernier ne recueille comme observations que les durées de séjour des objets insérés dans la liste *après* l'appel à `CollectStat`, et qui *quittent* la liste pendant la période d'observation, c'est-à-dire entre la dernière initialisation de ce bloc et l'instant courant. Les fonctions `StatSize` et `StatSojourn` retournent les blocs statistiques en question. `CollectStat` appelle aussi automatiquement `InitStat` pour initialiser ces deux blocs et faire une mise à jour pour le premier. Lorsqu'on appelle cette fonction, il est recommandé de le faire tout de suite après la création de la liste.

```
PROCEDURE InitStat
   ( L : List
   );
```

Réinitialise les deux blocs statistiques associés à la liste `L`, par des appels à `STAT.Init`, et fait une mise à jour pour celui qui mesure l'évolution de la longueur de la liste. Ces deux blocs doivent avoir été créés auparavant par un appel à `CollectStat`.

```
PROCEDURE Init
   ( L : List
   );
```

Vide la liste `L` de tous ses objets et réinitialise ses blocs statistiques (s'il y a lieu). Ne détruit pas, cependant, les objets qui s'y trouvent. Ces objets peuvent donc servir ultérieurement à d'autres fins. Seul l'usager peut détruire, lorsqu'il le désire et afin de récupérer l'espace qu'ils occupent, les objets qu'il a lui-même créés.

```
PROCEDURE Insert
   ( P : ADDRESS;
     Where : InPosition;
     L : List
   );
```

Insère l'objet `P` dans la liste `L`. 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 `P` devient l'objet courant. La liste `L` doit être de type non ordonné, c'est-à-dire avoir été créée à l'aide de la procédure `Create`.

```
PROCEDURE InsertOrd
   ( P : ADDRESS;
     L : List
   );
```

Insère l'objet `P` dans la liste `L`. Cette liste doit être de type ordonné (créée par la procédure `CreateOrd`), et est donc maintenue en ordre selon la fonction d'ordonnancement fournie par l'usager.

```
PROCEDURE Belongs
   ( P : ADDRESS;
     L : List
   ) : BOOLEAN;
```

Prend la valeur `TRUE` si l'objet `P` est dans la liste `L`, sinon prend la valeur `FALSE`. Lorsque la fonction prend la valeur `TRUE`, `P` devient l'objet courant, sinon l'objet courant demeure inchangé.

```
PROCEDURE View
   ( VAR P  : ADDRESS;
     Which  : OutPosition;
     L      : List
   );
```

Permet d'observer un objet `P` qui se trouve dans la liste `L`. Selon la valeur du paramètre `Which`, cet objet sera le premier objet de la liste (`First`), le dernier (`Last`), l'objet courant (`Current`), celui qui précède l'objet courant (`Previous`) ou celui qui le suit (`Next`). La valeur `NIL` sera retournée pour `P` si on tente d'observer l'objet précédent le premier de la liste, ou celui suivant le dernier de la liste, ou encore si on tente d'observer un objet dans une liste vide. Si `P` est différent de `NIL`, alors il devient l'objet courant.

```
PROCEDURE Remove
   ( VAR P : ADDRESS;
     Where : OutPosition;
     L     : List
   );
```

Retire un objet de la liste `L` et retourne cet objet dans la variable `P`. Selon la valeur du paramètre `Where`, cet objet sera le premier objet de la liste (`First`), le dernier (`Last`), l'objet courant (`Current`), celui qui précède l'objet courant (`Previous`) ou celui qui le suit (`Next`). La valeur

`NIL` sera retournée pour `P` si on tente de retirer l'objet précédent le premier de la liste, ou celui suivant le dernier de la liste, ou encore si on tente de retirer un objet dans une liste vide. 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).

```
PROCEDURE RemoveObject
   ( P : ADDRESS;
     L : List
   );
```

Retire de la liste L un objet particulier `P`, spécifié par l'usager. Une erreur sera signalée si on tente de retirer un objet d'une liste vide, ou un objet ne faisant pas partie de la liste. Cette procédure regarde d'abord si l'objet courant est l'objet cherché, sinon elle parcourt la liste pour le retrouver. Lorsqu'on retire l'objet courant, le nouvel objet courant est choisi de la même façon que dans `Remove`.

```
PROCEDURE Size
   ( L : List
   ) : CARDINAL;
```

Retourne le nombre d'objets se trouvant dans la liste L.

```
PROCEDURE Concatenate
   ( VAR L1, L2 : List
   );
```

Concatène la liste `L2` à la fin de la liste `L1`, puis détruit `L2`. `L1` et `L2` doivent avoir été créées par `Create` (c'est-à-dire être non ordonnées). L'objet courant de `L1` n'est pas modifié. Si des statistiques étaient recueillies sur `L1`, elles sont réinitialisées.

```
PROCEDURE Sort
   ( L : List;
     Precede : OrdProc
   );
```

Trie la liste `L` selon la fonction d'ordonnancement `Precede` fournie par l'usager. On peut passer directement l'identificateur de notre fonction comme argument en autant qu'elle soit compatible avec la définition du type `OrdProc`. Cette fonction doit retourner la valeur `TRUE` si l'objet A doit précéder l'objet `B` dans la liste `L`. La liste `L` doit être de type ordonné (créée à l'aide de la procédure `CreateOrd`). La fonction `Precede` utilisée lors du tri peut être différente de celle spécifiée lors de la création. Dans ce cas, la liste est tout simplement retriée selon la nouvelle fonction, qui sera utilisée par la suite.

```
PROCEDURE Merge
   ( VAR L1, L2 : List
   );
```

Fusionne les listes ordonnées `L1` et `L2` en une seule liste ordonnée `L1`, puis détruit `L2`. `L1` et `L2` doivent être toutes les deux ordonnées selon la même procédure `Precede` fournie par l'usager. La nouvelle liste s'appellera `L1` et sera triée selon l'ordre des listes de départ. L'objet courant de `L1` n'est pas modifié. Si des statistiques étaient recueillies sur `L1`, elles sont réinitialisées.

```
PROCEDURE Split
   ( VAR L, L1, L2 : List;
     Name1, Name2 : ARRAY OF CHAR
   );
```

Divise la liste `L` en deux nouvelles listes `L1` et `L2`, à partir de l'objet courant de `L`, puis détruit la liste `L`. Ces deux nouvelles listes auront le même type d'ordonnancement que la liste L. La liste `L2` contiendra l'objet courant de `L` et tous ses suivants, dans le même ordre, et la liste `L1` contiendra tous les objets précédant l'objet courant de `L`, dans le même ordre que dans `L`. Les objets courants de `L1` et `L2` deviennent leurs premiers objets respectifs. Les paramètres `Name1` et `Name2` servent à donner des descripteurs aux nouvelles listes, pour les identifier lors des traces à l'écran. Si des statistiques étaient recueillies sur `L`, elles sont réinitialisées et on en recueillera sur `L1` et `L2`.

```
PROCEDURE FindFirstSuchThat
   ( L : List;
     VAR P : ADDRESS;
     Cond  : CondProc
   );
```

Parcourt la liste `L` linéairement à partir de l'objet courant (inclusivement), et retourne le premier objet `P` dans la liste tel que `Cond (P) = TRUE`. Si aucun des objets à partir de l'objet courant jusqu'à la fin de la liste ne satisfait la condition, la valeur de `P` retournée est `NIL`. Si on veut que toute la liste soit parcourue, à partir du début, on peut appeler `View` juste avant l'appel de cette procédure. A noter que la procédure `Cond` ne doit pas modifier (par des effets de bord, par exemple) la valeur du pointeur sur l'objet courant de la liste `L`, sinon les résultats obtenus sont imprévisibles.

```
PROCEDURE ForEachInListDo
   ( L : List;
     Action : ActionProc
   );
```

Parcourt la liste `L` linéairement à partir de l'objet courant (inclusivement), et appelle la procédure `Action (P)` pour chacun des objets `P` de la liste, à partir de l'objet courant jusqu'à la fin de la liste. Si on veut que toute la liste soit parcourue, à partir du début, on peut appeler `View` juste avant l'appel de cette procédure. A noter que la procédure `Action` ne doit pas modifier (par des effets de bord, par exemple) la valeur du pointeur sur l'objet courant de la liste `L`, sinon les résultats obtenus sont imprévisibles.

```
PROCEDURE StatSize
   ( L : List
   ) : Block;
```

Retourne le bloc statistique qui mesure l'évolution de la longueur de la liste en fonction du temps. Il s'agit d'un bloc de type `Accumulate`. Ce bloc n'existe que si on a déjà appelé `CollectStat` pour cette liste. Sinon, cette fonction imprime un message d'erreur.

```
PROCEDURE StatSojourn
   ( L : List
   ) : Block;
```

Retourne le bloc statistique qui mesure les durées de séjour des objets dans la liste `L`. Il s'agit d'un bloc de type `Tally`. Ce bloc n'existe que si on a déjà appelé `CollectStat` pour cette liste. Sinon, cette fonction imprime un message d'erreur.

```
PROCEDURE Report
   ( L : List
   );
```

Fournit un rapport statistique complet sur la liste `L`. On doit avoir auparavant appelé `CollectStat` pour cette liste.

```
PROCEDURE Delete
   ( VAR L : List
   );
```

Vide la liste `L` de tous ses objets et détruit cette liste. Tout comme `Init`, cette procédure ne détruit pas les objets se trouvant dans cette liste.

```
END LIST.
```

RES

Le module `RES` fournit le type `Resource`, et un ensemble de procédures permettant la synchronisation des processus qui utilisent des ressources à capacité limitée. Un objet de type `Resource` représente une unité de service (une ressource), dont la capacité correspond au nombre de serveurs. Ces serveurs sont tous identiques.

Un processus (voir le module `PROCS`) doit demander (`Request`) un certain nombre d'unités de la ressource, et obtenir ces unités, avant de pouvoir les utiliser. Lorsqu'il en a terminé, il libère ce nombre d'unités (`Release`). Il n'est pas nécessaire de demander ou de libérer d'un seul coup toutes les unités de ressource que l'on veut utiliser. Un processus peut aussi détenir simultanément des unités de plusieurs ressources différentes. Lorsqu'un processus demande une ressource et que le nombre demandé d'unités est disponible, il obtient immédiatement ces unités. Sinon, il est placé dans une file d'attente et doit attendre qu'un nombre suffisant d'unités soit libéré avant d'obtenir la ressource en question et poursuivre son exécution.

Pour chaque ressource qu'il veut utiliser, l'usager doit déclarer une variable de type `Resource`, puis créer cette ressource en appelant la procédure `Create`. Chaque ressource possède une file d'attente unique, de capacité infinie, et dont la politique de service est fixée lors de la création. Cette politique peut être `Fifo` (premier arrivé, premier servi), `Lifo` (dernier arrivé, premier servi) ou `Prior` (avec priorités). Dans le cas d'une file d'attente avec priorités, chaque processus qui demande la ressource doit le faire en utilisant la procédure `RequestWithPrior`, en indiquant sa priorité. Les priorités sont des valeurs réelles, et les priorités plus élevées passent avant les priorités plus faibles. La file d'attente est ordonnée d'abord selon les priorités, et les processus ayant des priorités égales sont placés selon leur ordre d'arrivée.

Lorsque des unités de ressource se libèrent, et que le premier processus dans la file 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 par les unités disponibles, si un tel processus existe. De même, si un processus qui a besoin de plus d'unités que ce qui est disponible attend dans la file, et si un second processus arrive et demande un nombre d'unités qui ne dépasse pas ce qui est disponible, la requête du second processus sera satisfaite immédiatement (même si ce dernier a une priorité inférieure). Si on veut qu'un processus s'empare des unités de ressource au fur et à mesure que celles-ci deviennent disponibles, même si elles ne sont pas en nombre suffisant pour ses besoins, on peut lui faire demander une à une les unités de ressource désirées. Du point de vue des statistiques, toutefois, ce sera comme si toutes ces requêtes venaient de processus différents.

La fonction `Avail` permet d'obtenir le nombre d'unités disponibles, pour une ressource donnée, et la procédure `ChangeCapacity` permet de modifier la capacité d'une ressource.

A chaque ressource sont associées deux listes (ce sont des objets de type `List`, du module `LIST`), soit la liste des processus en attente pour la ressource (la file d'attente) et la liste des processus en service. Les fonctions `WaitList` et `ServList` retournent ces deux listes, respectivement, et permettent donc d'observer ce qu'il y a dans ces listes, d'accéder à leurs blocs statistiques, etc. A noter que la taille de `ServList` correspond au nombre de processus qui occupent la ressource, mais pas nécessairement au nombre d'unités de la ressource qui sont utilisées. Les objets manipulés par ces deux listes ne sont pas vraiment des objets de type `ProcessInstance` du module `PROCS`. En fait, chaque fois qu'un processus demande une ressource, on lui ouvre un dossier contenant un pointeur sur ce processus, en plus d'information additionnelle. Le type `UserDossier`, prédéfini dans le module, représente un pointeur sur un tel dossier. Les objets manipulés par `WaitList` et `ServList` sont des variables de type `UserDossier`. Les fonctions `User` et `NbUnits` retournent respectivement le nom du processus pour lequel un dossier donné a été créé, et le nombre demandé d'unités de la ressource. Ces fonctions peuvent permettre, par exemple, de parcourir l'une des deux listes pour retrouver un processus particulier, de savoir combien d'unités un tel processus demande ou occupe, etc. Il est possible d'insérer ou de retirer directement de ces listes des objets de type `UserDossier`, mais il n'est pas recommandé de le faire à moins que ce soit vraiment nécessaire (par exemple si on veut simuler des préemptions). En particulier, lorsqu'on effectue directement de telles insertions ou de tels retraits, le bloc statistique retourné par `StatUtil` (voir ci-bas) n'est pas mis à jour automatiquement.

Le progiciel peut recueillir des statistiques automatiquement sur les ressources lorsqu'on le lui demande. Lorsqu'on appelle `CollectStat` pour une ressource donnée, *SIMOD* appelle `LIST.CollectStat` pour les deux listes associées, et associe aussi à cette ressource trois autres blocs statistiques, qui servent à mesurer respectivement l'évolution de la capacité et l'utilisation de la ressource en fonction du temps, et les durées de séjour des processus sur cette ressource. Les fonctions `StatCapacity`, `StatUtil` et `StatSojourn`, respectivement, retournent ces blocs statistiques, ce qui permet par exemple d'appeler les fonctions du module `STAT` pour observer certaines statistiques particulières. La procédure `Report` permet par ailleurs d'obtenir un rapport statistique complet sur une ressource donnée, tandis que `InitStat` permet de réinitialiser toutes les statistiques associées à une ressource donnée, y compris celles des listes des processus en attente et en service. Lorsqu'on ne veut recueillir des statistiques que sur les listes `WaitList` ou `ServList`, on peut appeler `CollectStat` seulement pour la ou les liste(s) qui nous intéresse(nt). Cela peut permettre d'accélérer l'exécution du programme.

Enfin, la procédure `Init` permet d'initialiser une ressource, de vider les deux listes qui sont associées, et de réinitialiser ses statistiques, tandis que `Delete` permet de détruire la ressource.

```
DEFINITION MODULE RES;

FROM STAT  IMPORT Block;
FROM LIST  IMPORT List;
FROM PROCS IMPORT ProcessInstance;

TYPE
   Resource;
```

> Pour chaque type de ressource qu'il veut utiliser, l'usager doit déclarer une variable de ce type, puis appeler la procédure `Create` pour créer cette ressource.

`ServicePolicy = (Fifo, Lifo, Prior);`

Politique de service pour une ressource, détermine l'ordonnancement de la file d'attente. `Fifo`: les processus sont placés dans la file selon leur ordre d'arrivée, les premiers arrivés ayant priorité. `Lifo`: les processus sont placés dans l'ordre inverse de leur arrivée, les derniers arrivés ayant priorité. `Prior`: Les processus doivent donner leur priorité lorsqu'ils demandent la ressource (`RequestWithPrior`), et les priorités les plus élevées passent en premier.

`UserDossier;`

Chaque fois qu'un processus demande pour une ressource, le module crée un dossier contenant un pointeur sur ce processus, le nombre d'unités demandées, etc. Le type `UserDossier` est un pointeur sur un tel dossier. Les listes retournées par les fonctions `WaitList` et `ServList` sont des listes d'objets de ce type.

```
PROCEDURE Create
   ( VAR R     : Resource;
     Policy    : ServicePolicy;
     Capac     : CARDINAL;
     Name      : ARRAY OF CHAR
   );
```

Crée la ressource `R`. Le paramètre `Policy` indique la politique de service, et `Capac` indique la capacité initiale de cette ressource. Cette capacité peut être modifiée par la suite en appelant `ChangeCapacity`. Le paramètre `Name` sert à identifier la ressource dans les rapports statistiques.

```
PROCEDURE CollectStat
   ( R : Resource
   );
```

Déclenche un recueil automatique de statistiques pour la ressource `R`. Cette ressource doit avoir été créée auparavant. Cette procédure appelle les procédures `LIST.CollectStat(WaitList(R))` et `LIST.CollectStat(ServList(R))`, afin que des statistiques soient recueillies automatiquement sur les listes des processus en attente et en service. Elle crée et initialise aussi trois autres blocs statistiques associés à cette ressource, qui seront mis à jour automatiquement. Les fonctions `StatCapacity`, `StatUtil` et `StatSojourn` retournent ces blocs statistiques, respectivement. Le premier sert à mesurer l'évolution de la capacité de la ressource en fonction du temps (il est mis à jour seulement lorsqu'on modifie la capacité de la ressource), le second sert à mesurer l'utilisation de la ressource (le nombre d'unités occupées) en fonction du temps, et le troisième recueille des statistiques sur les durées de séjour (attente + service) des processus; il échantillonne une valeur pour chaque dossier qui quitte `R` après l'appel à `CollectStat`. Les deux premiers blocs sont de type `Accumulate` et le troisième est de type `Tally` (voir le module `STAT`).

Lorsqu'on appelle cette procédure, il est recommandé de le faire juste après la création de la ressource. Note : Lorsqu'on tue un processus qui occupe une ressource, cela peut avoir des conséquences imprévisibles sur les statistiques.

```
PROCEDURE InitStat
   ( R : Resource
   );
```

Réinitialise tous les blocs statistiques associés à la ressource `R`, par des appels à `LIST.InitStat` et `STAT.Init`, et effectue une mise-à-jour pour les blocs de type `Accumulate`. On doit avoir appelé `CollectStat` auparavant pour cette ressource.

```
PROCEDURE Init
   ( R : Resource
   );
```

Vide les deux listes associées à la ressource `R`, en récupérant l'espace occupé par les dossiers. Les processus qui s'y trouvent demeurent suspendus. Réinitialise aussi tous les blocs statistiques associés à cette ressource, s'il y a lieu.

```
PROCEDURE ChangeCapacity
   ( R : Resource;
     N : INTEGER
   );
```

Modifie de `N` unités (augmente si `N` > 0, diminue si `N` < 0) la capacité de la ressource `R`, c'est-à-dire le nombre d'unités de cette ressource. Si `N` > 0 et s'il y a des processus en attente dont la requête peut maintenant être satisfaite, on leur accorde la ressource. Si `N` < 0, le nombre d'unités disponibles (la capacité moins le nombre d'unités occupées) pour la ressource `R` doit être supérieur ou égal à $-$`N`, sinon une erreur sera signalée. En particulier, la capacité d'une ressource ne peut jamais devenir négative.

```
PROCEDURE SetCapacity
   ( R : Resource;
     N : CARDINAL
   );
```

Fixe à `N` unités la capacité de la ressource `R`. Similaire à `ChangeCapacity`.

```
PROCEDURE Avail
   ( R : Resource
   ) : CARDINAL;
```

Fonction qui retourne le nombre d'unités de la ressource `R` qui sont présentement disponibles, c'est-à-dire la capacité de `R` moins le nombre d'unités occupées.

```
PROCEDURE Request
   ( N : CARDINAL;
     R : Resource
   );
```

Le processus qui appelle cette procédure demande `N` unités de la ressource `R`. Si le nombre d'unités disponibles suffit pour répondre à sa requête, on lui accorde les unités demandées et il les détient jusqu'à ce qu'il appelle `Release` pour les libérer. Ce processus est aussi inséré dans la liste des processus en service pour cette ressource. Par contre, si ce nombre n'est pas suffisant, le processus est placé dans la file d'attente pour `R`, dans l'état `Suspended`, et son exécution est bloquée jusqu'à ce qu'il obtienne les unités de ressource demandées.

```
PROCEDURE RequestWithPrior
   ( N : CARDINAL;
     R : Resource;
     Pri : LONGREAL
   );
```

Similaire à la procédure `Request`, sauf que le processus appelant demande la ressource avec une priorité. On doit utiliser cette procédure dans le cas de ressources dont la politique de service est de type `Prior`. Si le nombre d'unités disponibles est suffisant pour répondre à sa requête, on lui accorde les unités demandées et il les détient jusqu'à ce qu'il appelle `Release` pour les libérer. Le paramètre `Pri` indique la priorité, et les priorités plus élevés passent avant les priorités plus faibles. Les processus ayant des valeurs de priorités égales sont classés selon leur ordre d'arrivée.

```
PROCEDURE Release
   ( N : CARDINAL;
     R : Resource
   );
```

Le processus appelant libère `N` unités de la ressource `R`. Si ce processus occupait exactement `N` unités de la ressource `R`, il est retiré de la liste des processus en service pour cette ressource. S'il occupait moins de `N` unités, un message d'erreur est imprimé. S'il y a d'autres processus en attente pour cette ressource, les unités libérées leur sont maintenant disponibles et peuvent leur être allouées.

```
PROCEDURE WaitList
  ( R : Resource
  ) : List;
```

Retourne la liste des dossiers sur les processus en attente pour la ressource `R`. Les objets de cette liste sont de type `UserDossier`.

```
PROCEDURE ServList
  ( R : Resource
  ) : List;
```

Retourne la liste des dossiers sur les processus en service pour la ressource `R`, c'est-à-dire les processus qui occupent au moins une unité de cette ressource. Les objets de cette liste sont de type `UserDossier`.

```
PROCEDURE User
   ( U : UserDossier
   ) : ProcessInstance;
```

Retourne le processus auquel correspond le dossier `U`.

```
PROCEDURE NbUnits
   ( U : UserDossier
   ) : CARDINAL;
```

Retourne le nombre d'unités de la ressource qui avait été demandé par le processus auquel correspond le dossier `U`, lors de la création de ce dossier.

```
PROCEDURE StatCapacity
   ( R : Resource
   ) : Block;
```

Retourne le bloc statistique qui mesure l'évolution de la capacité de la ressource `R` en fonction du temps. Il s'agit d'un bloc de type `Accumulate`. Ce bloc n'existe que si on a déjà appelé `CollectStat` pour cette ressource. Sinon, cette fonction imprime un message d'erreur.

```
PROCEDURE StatUtil
   ( R : Resource
   ) : Block;
```

Retourne le bloc statistique qui mesure l'évolution du nombre d'unités de la ressource `R` qui sont occupées, en fonction du temps. La moyenne associée à ce bloc, divisée par la capacité de cette ressource, correspond au taux d'utilisation. Il s'agit d'un bloc de type `Accumulate`. Ce bloc n'existe que si on a déjà appelé `CollectStat` pour cette ressource. Sinon, cette fonction imprime un message d'erreur. A noter que le bloc retourné par `StatSize (ServList (R))` s'intéresse au nombre de *dossiers* en 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.

```
PROCEDURE StatSojourn
   ( R : Resource
   ) : Block;
```

Retourne le bloc statistique qui mesure les durées de séjour des "`UserDossier`"'s pour la ressource `R`. Il s'agit d'un bloc de type `Tally`, auquel une observation est ajoutée chaque fois qu'un processus libère toutes les unités de ressource `R` dont il dispose. Ce bloc n'existe que si on a déjà appelé `CollectStat` pour cette ressource. Sinon, cette fonction imprime un message d'erreur.

```
PROCEDURE Report
   ( R : Resource
   );
```

Fournit un rapport statistique complet sur la ressource `R`, dans le fichier de sortie courant. On doit avoir auparavant appelé `CollectStat` pour cette ressource. 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.

```
PROCEDURE Delete
   ( VAR R : Resource
   );
```

Détruit la ressource `R` et récupère l'espace mémoire qu'elle occupait.

```
END RES.
```

BIN

Le module `BIN` fournit le type `Bin`, qui permet d'établir des relations de type producteur/consommateur entre les processus. Un `Bin` correspond essentiellement à une boîte (ou une pile) de jetons indifférentiables, et une liste de processus qui attendent pour des jetons. On peut ajouter des jetons à la pile, en appelant `Give`, ou demander des jetons, en appelant `Take`. Dans le premier cas, on joue le rôle de producteur, tandis que dans le second cas, on joue le rôle de consommateur. Un producteur peut ajouter des jetons à la pile sans les avoir demandés auparavant, et un consommateur peut en demander sans jamais les remettre dans la pile. En fait, un appel à `Give` peut être vu comme une *création* de jetons, et un appel à `Take` comme une demande de destruction de jetons. Un appel à `Give` peut se faire de n'importe où, mais un appel à `Take` ne peut se faire qu'à partir d'un processus. Le nombre de jetons dans la pile doit toujours être supérieur ou égal à zéro, de sorte qu'un processus qui demande un nombre de jetons plus grand que ce qui est disponible se voit bloqué et placé dans une file d'attente.

L'utilisation du module `BIN` se fait de façon assez semblable à celle du module `RES`. Pour chaque pile de jetons que l'on veut utiliser, on doit déclarer une variable de type `Bin`, puis l'initialiser en appelant `Create`. Chaque pile possède une file d'attente unique, de capacité infinie, et dont la politique de service est fixée lors de la création. Cette politique peut être `Fifo` (premier arrivé, premier servi), `Lifo` (dernier arrivé, premier servi) ou `Prior` (avec priorités). Dans le cas d'une file d'attente avec priorités, chaque processus qui demande des jetons doit le faire en utilisant la procédure `TakeWithPrior`, en indiquant sa priorité. Les priorités sont des valeurs réelles, et les priorités plus élevées passent avant les priorités plus faibles. La file d'attente est ordonnée d'abord selon les priorités, et les processus ayant des priorités égales sont placés selon leur ordre d'arrivée.

Lorsque des nouveaux jetons sont placés dans la pile, et que le premier processus dans la file demande plus de jetons que ce qui est disponible, les jetons sont alloués au prochain processus dans la file dont la requête peut être satisfaite par les jetons disponibles, et ne sont alloués que si un tel processus existe. De même, si un processus qui a besoin de plus de jetons que ce qui est disponible attend dans la file, et si un second processus demande un nombre de jetons qui ne dépasse pas ce qui est disponible, la requête du second processus sera satisfaite immédiatement (même si ce dernier a une priorité inférieure). Si on veut qu'un processus s'empare des jetons au fur et à mesure que ceux-ci deviennent disponibles, même s'ils ne sont pas en nombre suffisant pour ses besoins, on peut lui faire demander un à un les jetons désirés. Toutefois, du point de vue des statistiques sur cette pile, ce sera comme si toutes ces requêtes venaient de processus différents. La fonction `Avail` permet d'obtenir le nombre de jetons disponibles, pour une pile donnée.

A chaque pile est associé un objet de type `List`, du module `LIST`, qui correspond à la liste des processus qui attendent pour avoir des jetons de cette pile. La fonction `WaitList` retourne cette liste, ce qui permet d'accéder à ses blocs statistiques, d'observer les dossiers sur les processus en attente, etc. A noter que les objets manipulés par cette liste ne sont pas vraiment des objets de type `ProcessInstance` du module `PROCS`. En fait, chaque fois qu'un processus est placé dans la file d'attente pour une pile, on lui ouvre un dossier contenant un pointeur sur ce processus, en plus d'informations additionnelles. Le type `UserDossier`, prédéfini dans le module, représente un pointeur sur un tel dossier. Les objets manipulés par `WaitList` sont de type `UserDossier`. Les fonctions `User` et `NbTokens` retournent respectivement le nom du processus pour lequel un dossier donné a été créé, et le nombre de jetons demandés. Il est possible d'insérer ou des retirer directement de la liste des objets de type `UserDossier`, mais cela n'est pas vraiment recommandé. Enfin, la procédure `Init` permet d'initialiser une pile, et `Delete` permet de la détruire.

```
DEFINITION MODULE BIN;

FROM LIST  IMPORT List;
FROM PROCS IMPORT ProcessInstance;

TYPE
```

```
Bin;
```

Pour chaque pile de jetons qu'il veut utiliser, l'usager doit déclarer une variable de ce type, puis appeler `Create` pour initialiser cette pile.

```
ServicePolicy = (Fifo, Lifo, Prior);
```

Types de politiques de service déterminant l'ordonnancement de la file d'attente pour une pile de jetons. `Fifo`: les processus sont placés dans la file selon leur ordre d'arrivée, les premiers arrivés ayant priorité. `Lifo`: les processus sont placés dans l'ordre inverse de leur arrivée, les derniers arrivés ayant priorité. `Prior`: Les processus doivent donner leur priorité lorsqu'ils demandent des jetons (`TakeWithPrior`), et les priorités les plus élevées passent en premier.

```
UserDossier;
```

Chaque fois qu'un processus demande des jetons, le module crée un dossier contenant un pointeur sur ce processus, le nombre de jetons demandés, etc. Le type `UserDossier` est un pointeur sur un tel dossier. La liste retournée par la fonction `WaitList` est une liste d'objets de ce type.

```
PROCEDURE Create
   ( VAR B    : Bin;
     Policy   : ServicePolicy;
     Name     : ARRAY OF CHAR
   );
```

Crée la pile de jetons `B`, initialement vide. Le paramètre `Policy` indique la politique de service. Le paramètre `Name` sert à identifier cette pile dans les rapports ou les messages d'erreurs. Seuls les 32 premiers caractères sont considérés.

```
PROCEDURE Init
   ( B : Bin
   );
```

Vide la liste associée à `B` (file d'attente), en récupérant l'espace occupé par les dossiers. Les processus qui s'y trouvent demeurent suspendus. Remet le nombre de jetons à zéro.

```
PROCEDURE Avail
   ( B : Bin
   ) : CARDINAL;
```

Retourne le nombre de jetons disponibles dans `B`.

```
PROCEDURE Take
   ( N : CARDINAL;
     B : Bin
   );
```

Cette procédure ne peut s'appeler qu'à partir d'un processus. Celui qui l'appelle demande `N` jetons de la pile `B`. Si le nombre de jetons disponibles suffit pour répondre à sa requête, on lui accorde les jetons demandés. Sinon, le processus est placé dans la file d'attente pour `B` (dans l'état `Suspended`), et son exécution est bloquée jusqu'à ce qu'il obtienne les jetons demandés.

```
PROCEDURE TakeWithPrior
   ( N : CARDINAL;
     B : Bin;
     Priority : LONGREAL
   );
```

Similaire à la procédure `Take`, sauf que le processus appelant demande `N` jetons avec une priorité. On doit utiliser cette procédure dans le cas d'une pile dont la politique de service est de type `Prior`. Le paramètre `Priority` indique la priorité, et les priorités plus élevées passent (dans la file d'attente) avant les priorités plus faibles. Les processus ayant des valeurs de priorités égales sont classés selon leur ordre d'arrivée.

```
PROCEDURE Give
   ( N : CARDINAL;
     B : Bin
   );
```

Place `N` jetons dans la pile `B`. S'il y a des processus en attente de jetons pour cette pile, ces nouveaux jetons maintenant disponibles peuvent leur être alloués et ces processus peuvent reprendre leur exécution.

```
PROCEDURE WaitList
  ( B : Bin
  ) : List;
```

Retourne la liste des dossiers sur les processus en attente pour `B`. Les objets de cette liste sont de type `UserDossier`.

```
PROCEDURE User
   ( U : UserDossier
   ) : ProcessInstance;
```

Retourne le processus auquel correspond le dossier `U`.

```
PROCEDURE NbTokens
   ( U : UserDossier
   ) : CARDINAL;
```

Retourne le nombre de jetons demandés par le processus en attente, auquel correspond le dossier `U`.

```
PROCEDURE Delete
   ( VAR B : Bin
   );
```

Détruit la pile `B` et récupère son espace mémoire.

```
END BIN.
```

COND

Ce module offre le type `Condition`, qui correspond essentiellement à un indicateur booléen et à une liste de processus qui attendent que la condition devienne vraie. Un processus qui appelle `Wait` pour une condition donnée peut poursuivre son exécution seulement si la condition est vraie. Sinon, il se bloque et doit attendre que la condition soit mise à “vrai” avant de reprendre son exécution. Les procédures `SetTrue` et `SetFalse` permettent de mettre à vrai ou à faux l'état de la `Condition`, et `CondState` permet de connaître son état.

Pour chaque condition que l'on veut utiliser, on doit déclarer une variable de type `Condition`, puis créer cette condition en appelant la procédure `Create`. Chaque condition possède une liste de processus en attente, et la fonction `WaitList` retourne cette liste. Les objets manipulés par cette liste sont de type `UserDossier`, tout comme dans `RES` et dans `BIN`. La fonction `User` retourne le nom du processus correspondant à un dossier donné. La procédure `Init` permet de réinitialiser une condition, et `Delete` permet de la détruire.

```
DEFINITION MODULE COND;

FROM LIST  IMPORT List;
FROM PROCS IMPORT ProcessInstance;

TYPE

   Condition;
```

Pour chaque condition que l'on veut utiliser, on doit déclarer une variable de ce type, puis appeler `Create` pour l'initialiser.

```
   UserDossier;
```

Chaque fois qu'un processus appelle `Wait` pour une condition qui n'est pas vraie, le module crée un dossier contenant un pointeur sur ce processus, et place le pointeur sur ce dossier (qui est de type `UserDossier`) dans la liste des processus en attente pour cette condition (c'est-à-dire la liste retournée par `WaitList`).

```
PROCEDURE Create
   ( VAR C : Condition;
     V     : BOOLEAN;
     Name  : ARRAY OF CHAR
   );
```

Crée la condition `C` et l'initialise à la valeur `V`. Le paramètre `Name` sert à identifier cette `Condition`. Seuls les 32 premiers caractères sont considérés.

```
PROCEDURE Init
   ( C : Condition;
     V : BOOLEAN
   );
```

Vide la liste associée à `C` (file d'attente), en récupérant l'espace occupé par les dossiers. Les processus qui s'y trouvent demeurent suspendus. Réinitialise la `Condition` à `V`.

```
PROCEDURE Wait
   ( C : Condition
   );
```

Le processus qui appelle cette procédure ne peut pousuivre son exécution que si la condition `C` est vraie. Sinon, le processus est placé dans la liste d'attente pour `C`, dans l'état `Suspended`, et son exécution est bloquée jusqu'à ce que cette `Condition` soit mise à vrai.

```
PROCEDURE SetTrue
   ( C : Condition
   );
```

Met la condition `C` à vrai. Tous les processus en attente peuvent reprendre leur exécution.

```
PROCEDURE SetFalse
   ( C : Condition
   );
```

Met la condition `C` à faux.

```
PROCEDURE CondState
  ( C : Condition
  ) : BOOLEAN;
```

Retourne l'état de la condition `C`.

```
PROCEDURE WaitList
  ( C : Condition
  ) : List;
```

Retourne la liste des dossiers sur les processus en attente pour `C`. Les objets de cette liste sont de type `UserDossier`.

```
PROCEDURE User
   ( U : UserDossier
   ) : ProcessInstance;
```

Retourne le processus auquel correspond le dossier `U`.

```
PROCEDURE Delete
   ( VAR C : Condition
   );
```

Détruit la condition `C` et récupère l'espace mémoire qu'elle utilisait.

```
END COND.
```

MASLA

Le module `MASLA` permet de modéliser des relations de type maître/esclave entre des processus. Un objet de type `MasterSlave`, un type prédéfini dans `MASLA`, correspond à un point de rencontre entre des processus "maîtres" et des processus "esclaves". Chaque `MasterSlave` possède deux files d'attente: une pour les maîtres et une pour les esclaves. Un processus (agissant comme maître) peut demander un esclave en appelant la fonction `Request`, qui retourne le processus qui agira comme esclave. Si aucun esclave n'est disponible à ce point de rencontre au moment de la demande, le processus demandeur est placé dans la file d'attente des maîtres, et ne peut poursuivre son exécution que lorsqu'un esclave devient disponible. Un processus devient un esclave en appelant `Wait` pour un point de rencontre donné. Si la file d'attente des maîtres est vide, ce processus est placé dans la file d'attente des esclaves et doit attendre qu'un maître le demande. Sinon, le premier maître de la file obtient le nouvel esclave et peut maintenant poursuivre son exécution. Dans les deux cas, l'esclave doit attendre que son maître le libère en appelant `Release` avant de poursuivre son exécution. A noter qu'un maître peut détenir simultanément plusieurs esclaves, obtenus un à un par le biais du même ou de différents point(s) de rencontre.

Pour chaque point de rencontre maître/esclave qu'il veut utiliser, l'usager doit déclarer une variable de type `MasterSlave`, puis créer ce point de rencontre en appelant la procédure `Create`. Chaque point de rencontre possède deux files d'attente, de capacité infinie, et dont la politique de service est fixée lors de la création. Cette politique peut être `Fifo` (premier arrivé, premier servi), `Lifo`, (dernier arrivé, premier servi) ou `Prior` (avec priorités). Dans le cas d'une file d'attente avec priorités, chaque processus qui demande à devenir un maître ou un esclave doit le faire en appelant `WaitWithPrior` ou `RequestWithPrior`, selon le cas, en indiquant sa priorité. Les priorités sont des valeurs réelles, et les priorités plus élevées passent avant les priorités plus faibles. Les files d'attente sont ordonnées d'abord selon les priorités, et les processus ayant des priorités égales sont placés selon leur ordre d'arrivée.

A chaque point de rencontre sont associées deux listes, soit la file d'attente des maîtres et la file d'attente des esclaves. Les fonctions `MastersList` et `SlavesList` retournent ces deux listes, respectivement, et permettent donc d'observer ce qu'il y a dans ces listes, d'accéder à leurs blocs statistiques, etc. Les objets manipulés par ces deux listes sont de type `UserDossier`. Ce sont des pointeurs sur des blocs d'information contenant, entre autres choses, des pointeurs sur les processus qu'ils représentent. La fonction `User` retourne le nom du processus pour lequel un dossier donné a été créé.

Enfin, la procédure `Init` permet d'initialiser un point de rencontre, en vidant les deux listes qui lui sont associées, et `Delete` permet de le détruire.

```
DEFINITION MODULE MASLA;

FROM LIST  IMPORT List;
FROM PROCS IMPORT ProcessInstance;

TYPE
```

`MasterSlave;`

Pour chaque point de rencontre qu'il veut utiliser, l'usager doit déclarer une variable de ce type, puis appeler la procédure `Create` pour la créer.

`ServicePolicy = (Fifo, Lifo, Prior);`

Politique de service pour l'une ou l'autre des files d'attente d'un point de rencontre. `Fifo`: les processus sont placés dans la file selon leur ordre d'arrivée, les premiers arrivés ayant priorité. `Lifo`: les processus sont placés dans l'ordre inverse de leur arrivée, les derniers arrivés ayant priorité. `Prior`: Les processus doivent donner leur priorité lorsqu'ils demandent à devenir des maîtres ou des esclaves (`WaitWithPrior` ou `RequestWithPrior`), et les priorités les plus élevées passent en premier.

`UserDossier;`

Chaque fois qu'un processus est placé dans une file d'attente pour un point de rencontre, le module crée un dossier contenant un pointeur sur ce processus, et quelques informations additionnelles. Le type `UserDossier` est un pointeur sur un tel dossier. Les listes retournées par les fonctions `MastersList` et `SlavesList` sont des listes d'objets de ce type.

```
PROCEDURE Create
   ( VAR M         : MasterSlave;
     MastersPolicy : ServicePolicy;
     SlavesPolicy  : ServicePolicy;
     Name          : ARRAY OF CHAR
   );
```

Crée le point de rencontre `M`. Les paramètres `MastersPolicy` et `SlavesPolicy` indiquent les politiques de service respectives des files d'attente des maîtres et des esclaves. Le paramètre `Name` sert à identifier ce point de rencontre dans d'éventuels rapports. On ne retient que les 32 premiers caractères.

```
PROCEDURE Init
   ( M : MasterSlave
   );
```

Vide les deux listes associées à `M`, en récupérant l'espace occupé par les dossiers.

```
PROCEDURE Wait
   ( M : MasterSlave
   );
```

Le processus qui appelle cette procédure devient un esclave au point de rencontre `M`. Si la file d'attente des maîtres est vide, ce processus est placé dans la file d'attente des esclaves, dans l'état `Suspended`, et doit attendre qu'un maître le demande. Sinon, le premier maître de la file obtient cet esclave et peut maintenant poursuivre son exécution. Dans les deux cas, cet esclave doit attendre que son maître le libère en appelant `Release` avant de poursuivre son exécution. En attendant, il demeure dans l'état `Suspended`.

```
PROCEDURE WaitWithPrior
   ( M : MasterSlave;
     Priority : LONGREAL
   );
```

Similaire à `Wait`, sauf que le processus appellant possède une priorité. On doit utiliser cette procédure lorsque la file d'attente des esclaves a une politique de service de type `Prior`. Le paramètre `Priority` indique la priorité, et les priorités plus élevées passent avant les priorités plus faibles. Les processus ayant des priorités égales sont placés selon leur ordre d'arrivée.

```
PROCEDURE Request
   ( M : MasterSlave
   ) : ProcessInstance;
```

Un processus appelant cette fonction devient un maître au point de rencontre `M`. La fonction retourne le processus qui agira comme esclave. Si aucun esclave n'est disponible à ce point de rencontre, le processus (maître) demandeur est placé dans la file d'attente des maîtres, dans l'état `Suspended`, et ne peut poursuivre son exécution que lorsqu'un esclave devient disponible. Le processus esclave ne pourra poursuivre son exécution que lorsqu'on l'aura libéré en appelant `Release`.

```
PROCEDURE RequestWithPrior
   ( M : MasterSlave;
     Priority : LONGREAL
   ) : ProcessInstance;
```

Similaire à `Request`, sauf que le processus appellant possède une priorité. On doit utiliser cette procédure lorsque la file d'attente des maîtres a une politique de service de type `Prior`. Le paramètre `Priority` indique la priorité, et le priorités plus élevées passent avant les priorités plus faibles. Les processus ayant des priorités égales sont placés selon leur ordre d'arrivée.

```
PROCEDURE Release
   ( P : ProcessInstance
   );
```

Permet au processus `P`, qui était un esclave au point de rencontre `M`, de poursuivre son exécution. Equivaut à `ResetDelay (0.0)`. Si on veut que le processus `P` ne reprenne son exécution que dans `D` unités de temps, on peut plutôt appeler `ResetDelay (D)`.

```
PROCEDURE MastersList
  ( M : MasterSlave
  ) : List;
```

Retourne la liste des dossiers sur les processus en attente dans la file des maîtres au point de rencontre `M`. Les objets de cette liste sont de type `UserDossier`.

```
PROCEDURE SlavesList
  ( M : MasterSlave
  ) : List;
```

Retourne la liste des dossiers sur les processus en attente dans la file des esclaves au point de rencontre `M`. Les objets de cette liste sont de type `UserDossier`.

```
PROCEDURE User
   ( U : UserDossier
   ) : ProcessInstance;
```

Retourne le processus auquel correspond le dossier `U`.

```
PROCEDURE Delete
   ( VAR M : MasterSlave
   );
```

Détruit le point de rencontre `M` et récupère l'espace mémoire qu'il utilisait.

```
END MASLA.
```

B. Guide d'utilisation sur VAX/VMS.

SIMOD sous VAX/VMS a été implanté en utilisant le compilateur Modula-2 de Logitech [6] (version 3.1–9). Les programmes SIMOD doivent donc être traités par ce compilateur. A noter qu'il y a des différences mineures entre la version PC et la version VAX, à cause de la différence de taille pour les types `REAL`, `INTEGER`, et `CARDINAL`.

Tous les modules décrits dans ce guide sont précompilés, selon les spécifications de [6]. Les fichiers du progiciel sont placés dans un répertoire accessible aux usagers éventuels. Ce répertoire, appelons-le "`<dir>`", peut être désigné comme contenant la librairie principale de l'usager en effectuant les commandes :

```
$ ASSIGN <dir> MOD$LIBRARY
$ MLINK :== @<dir>MLINK
```

La commande `MLINK`, définie dans le répertoire "`<dir>`", effectue l'édition de liens d'un programme Modula-2 avec tous les modules de SIMOD. Pour réduire la taille du code exécutable, il est habituellement préférable de faire l'édition de liens seulement avec les modules effectivement utilisés (directement ou indirectement), en utilisant la commande `LINK` du système. Pour connaître les noms des modules utilisés indirectement, il suffit d'examiner le diagramme de la section 2.

Après avoir effectué les opérations ci-haut, la compilation, l'édition de liens et l'exécution d'un programme SIMOD se font de la même façon que pour un programme Modula-2. Si on veut faire l'édition de liens avec tous les modules de SIMOD, on remplace la commande "`LINK`" par "`MLINK`". Avec la commande `MLINK`, il n'est pas nécessaire de nommer les modules de SIMOD lors de l'édition de liens. Par exemple, pour le programme de la section 3.1, on fait :

```
$ MODULA Queue
$ MLINK  Queue
$ RUN    Queue
```

C. Guide d'utilisation sur IBM-PC.

SIMOD sous MS-DOS est implanté en utilisant le compilateur *TopSpeed Modula-2* [28]. Les programmes SIMOD doivent donc être traités par ce compilateur.

D. Guide d'utilisation sur SUN avec SUN/Modula-2.

SIMOD sous SUN/OS est implanté en utilisant le compilateur Modula-2 de SUN. Les programmes SIMOD doivent donc être traités par ce compilateur.

Au DIRO (Université de Montréal), SIMOD est compilé sous forme d'une librairie, située dans

`/home/anodin/stochas/m2lib/libsimod.a`

Pour utiliser cette librairie de même que la librairie `mylib` (utilisée par SIMOD), il faut fournir à la commande de compilation les options:

`-L/home/anodin/stochas/m2lib -lsimod -lmylib`

Références

[1] Birtwistle, G. M., Lomow, G., Unger, B., and Luker, P. (1986). Process Style Packages for Discrete Event Modelling. *Transactions of the Society for Computer Simulation*, **3**, 4, 279–318.

[2] Birtwistle, G. M. (1979). *Demos — A System for Discrete Event Modelling on Simula.* MacMillan.

[3] Birtwistle, G. M., Dahl, O. J., Myhrhaug, B., and Nygaard, K. (1979). *Simula begin.* Lund:Studentlitteratur.

[4] Bratley, P., Fox, B. L. et Schrage, L. E. (1987). *A Guide to Simulation.* Second edition, Springer-Verlag, New York.

[5] Concepcion, A. I. et Zeigler, B. P. (1988). DEVS Formalism: A Framework for Hierarchical Model Development. *IEEE Trans. on Software Engineering*, **14**, 2, 228–241.

[6] Eckhardt, H., Koch, J., Mall, M. et Putfarken, P. (1985). *Logitech Modula-2 User's Manual*, Logitech Inc., Redwood City, California.

[7] Ford, G. A. et Wiener, R. S. (1985). *Modula-2 : A Software Development Approach.* Wiley, New-York.

[8] Gleaves, R. (1984). *Modula-2 for Pascal Programmers.* Springer-Verlag, New-York.

[9] Kaubisch, W. H., Perrot, R. H. et Hoare, C. A. R. (1976). Quasi-parallel programming. *Software Practice and Experience*, **6**, 341–356.

[10] Knapp, V. (1986). The Smalltalk Simulation Environment. *1986 Winter Simulation Conference Proceedings*, 125–128.

[11] Kreutzer W. (1986). *System Simulation — Programming Styles and Languages.* Addison Wesley.

[12] Law, A. M. et Kelton, W. D. (1982). *Simulation Modeling and Analysis.* McGraw-Hill.

[13] L'Ecuyer, P. (1987). SIMPascal 2.0: Guide de l'usager pour VAX/VMS. Rapport no. DIUL-RT-8705, Département d'informatique, Université Laval.

[14] L'Ecuyer, P. et Côté, S. (1987). A Random Number Package with Splitting Facilities. Rapport no. DIUL-RR-8705, Département d'informatique, Université Laval. A paraître dans *ACM Transactions of Mathematical Software.*

[15] L'Ecuyer, P. et Giroux, N. (1987). A Process-Oriented Package Based on Modula-2. *1987 Winter Simulation Conference Proceedings*, Atlanta, Déc. 1987, 165–174.

[16] L'Ecuyer, P. (1988). Efficient and Portable Combined Random Number Generators. *Communications of the ACM*, **31**, 6, 742–749 et 774.

[17] L'Ecuyer, P., Mayrand, M. et Dror, M. (1988). Dynamic Scheduling of a Robot Servicing Machines on a One-Dimensional Line. Rapport no. DIUL-RR-8806, Département d'informatique, Université Laval. A paraître dans *IIE Transactions*.

[18] Livny, M. (1987). DELAB—A Simulation Laboratory. *1987 Winter Simulation Conference Proceedings*, Atlanta, Déc. 1987, 486–494.

[19] Malloy, B. et Soffa, M. L. (1986). SIMCAL : The Merger of Simula and Pascal. *1986 Winter Simulation Conference Proceedings*, 397–403.

[20] Moffat, D. (1984). Some Concerns about Modula-2. *SIGPLAN Notice* **19**, 41–47.

[21] Muller, C. (1986). Modula–Prolog: A software Development Tool. *IEEE Software*, **3**, 6, 39–45.

[22] Pegden, C. D. (1985). *Introduction to SIMAN*. Systems Modeling Corporation.

[23] Pritsker, A. A. B. (1986). *Introduction to Simulation and SLAM II*. Wiley, New-York.

[24] Russel, E. C. (1983). *Building Simulation Models with SIMSCRIPT II.5*. C. A. C. I., Los Angeles.

[25] Sharma, R. et Rose, L. L. (1988). Modular Design For Simulation. Research report, Dept. of Computer Science, University of Pittsburg (à paraître dans *Software Practice and Experience*).

[26] Schwetman H. (1986). SIMCAL : A C-Based, Process-oriented simulation language. *1986 Winter Simulation Conference Proceedings*, 387–396.

[27] Stairmand, M. C. et Kreutzer, W. (1988). POSE: a Process-Oriented Simulation Environment embedded in SCHEME. *Simulation*, **50**, 4, 143–153.

[28] *TopSpeed Modula-2 User's Manual*, Jensen and Partners International (1988).

[29] Unger, B. W., Lomow, G. A. et Birtwistle, G. M. (1984). *Simulation Software and ADA*. The Society for Computer Simulation, La Jolla, California.

[30] Vaucher, J. G. (1984). Process-oriented Simulation in Standard Pascal. Dans *Simulation in Strongly Typed Languages : Ada, Pascal, Simula, ...*, ed. par R. Bryant and B. W. Unger, SCS Simulation Series, **13**, 2, 37–43.

[31] Vaucher, J. G. et Lapalme, G. (1987). Process-oriented Simulation in Prolog. Dans *Simulation and AI*, ed. par P. A. Luker at G. Birtwistle, SCS Simulation Series, vol. 18, no. 3, 41–46.

[32] Wirth, N. (1985). *Programming in Modula-2*. Third ed., Springer-Verlag, New-York.