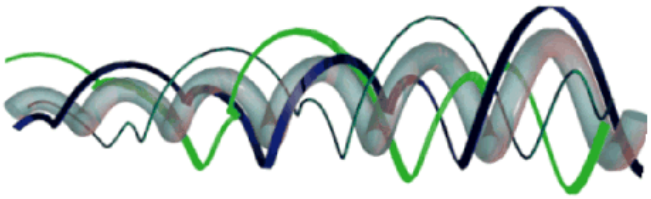
INSTITUT NATIONAL DE LA RECHERCHE AGRONOMIQUE

# DIESE :
# un outil de modélisation et de simulation de systèmes d'intérêt agronomique

**Jean-Pierre RELLIER**

RAPPORT UBIA TOULOUSE N° 2005/01 - JUIN 2005

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES



INRA

# DIESE :
# un outil de modélisation et de simulation de systèmes d'intérêt agronomique

Jean-Pierre RELLIER
INRA – Unité de Biométrie et Intelligence Artificielle – Auzeville, BP 52627, 31326 Castanet Tolosan (France)
*rellier@toulouse.inra.fr*

**Résumé**

Ce texte présente DIESE, une bibliothèque de classes orientée vers la modélisation et la simulation de systèmes, comme une alternative aux pratiques en cours, notamment dans la communauté des agronomes. Le modélisateur d'un système dans un domaine particulier réutilise les composants et les services de la bibliothèque, il les spécialise et les intègre en un programme exécutable : le simulateur. DIESE repose sur le paradigme de la simulation à événements discrets, mais permet de représenter les aspects continus et discontinus du fonctionnement d'un système. Un système est perçu par les trois points de vue structurel, fonctionnel et dynamique, qui fondent les trois classes de base d'un méta-modèle : les entités, les processus et les événements.

Après avoir décrit brièvement le contenu et les principes d'utilisation de DIESE, on donne l'exemple d'un simulateur écrit à l'aide de ce cadre de modélisation et avec ces outils de programmation. On fait ensuite la description plus détaillée et systématique des différentes classes. On présente enfin quelques outils d'interface qui permettent au modélisateur d'exploiter plus commodément les classes et les services, et qui fournissent à l'utilisateur des simulateurs une boite à outil interactive.

Par rapport aux pratiques en cours en modélisation des systèmes d'intérêt agronomique, DIESE se distingue par l'appui sur le méta-modèle. L'avantage est une définition plus nette des frontières du système, une identification rapide de ses composants, une sémantique claire des relations structurelles et fonctionnelles. Les capacités d'expression sont plus étendues sur les fonctions du système et les événements qui gèrent son évolution.

Dans la perspective, pour la communauté des agronomes, de créer et de partager des connaissances structurelles et fonctionnelles sur les systèmes agronomiques pour la capitalisation des connaissances, le développement rapide de modèles, pour mieux communiquer avec les autres disciplines (sciences de la décision, ergonomie, gestion de production) en jeu dans l'analyse et la résolution de problèmes complexes, DIESE peut favoriser l'émergence de cette culture.

**Plan**

# 1 Introduction

## *1.1 Le besoin en modélisation et en simulation*

Si les recherches et études sur les systèmes d'intérêt agronomique exploitent de manière croissante la modélisation et la simulation, c'est parce que l'évolution plus rapide du contexte de la production agricole nécessite des moyens appropriés de résolution de problèmes. L'expérimentation virtuelle permet, par rapport à l'expérimentation au champ dont elle se nourrit, des réponses plus rapides sur des systèmes simples et le traitement des problèmes sur des systèmes complexes. D'autre part, la compétence en mathématique et en informatique progresse dans les laboratoires de recherche agronomique et les institutions professionnelles agricoles. Enfin, les moyens informatiques sont vulgarisés, de plus en plus puissants et leurs systèmes d'exploitation sont plus conviviaux qu'aux débuts de l'Histoire.

Les besoins en modélisation et simulation portent sur des systèmes à tous les niveaux d'intégration. Du fonctionnement des plantes et des couverts végétaux dans leur environnement pédoclimatique à la question de la durabilité environnementale et économique des systèmes de culture, en passant par l'alimentation, la croissance et la reproduction les systèmes d'élevage ou la gestion conjointe de différents ateliers de production dans une exploitation agricole. Du point de vue fonctionnel, on voudrait simuler les phénomènes chimiques et physiques dans les systèmes de production (azote et matière organique des sols, variables climatiques dans les serres, etc.) ou les phénomènes spatialisés (érosion sur un territoire, flux de populations, etc.).

## *1.2 Les moyens actuels*

Le modèle d'un système devient un simulateur s'il est codé dans un langage informatique. De ce point de vue, on peut distinguer les langages dédiés à la simulation et les langages de programmation à vocation générale. Les premiers sont largement utilisés dans le monde industriel depuis l'avènement de SIMULA67. C'est une grande famille, avec générations, filiation et fratries. On y trouve des langages comme GPSS, SLAM, SIMSCRIPT, … pour citer quelques historiques (*cf.* Evans, 1988, chapitre 6 pour un aperçu général). De nombreux produits universitaires et commerciaux en sont issus. Ils sont utilisés pour des applications en gestion de production ou en conception de réseaux (télécommunication, informatique), au caractère discret très marqué, mais jamais pour des systèmes d'intérêt agronomique. Ceux-ci ont toujours été codés dans un langage de programmation général. D'abord FORTRAN, pour les modèles les plus anciens : la famille CERES (Ritchie et Otter, 1985 ; Jones C. et Kiniry, 1986), EPIC (Willians *et al.*, 1990), TOMGRO (Jones J.-W. *et al.* 1991), qui modélisaient des phénomènes de nature continue (physiologiques, bioclimatologiques ou portant sur le sol). Puis C pour les tentatives plus récentes, certaines visant la construction d'un simulateur des phénomènes biophysiques paramétrable pour nombreuses cultures (STICS, -Brisson *et al.*, 1998-), d'autres ciblés sur la représentation de la conduite des cultures (DECIBLE, -Meynard, 1997-) ou de systèmes plus complexes (ANSYL -Gibon *et al.*, 1989-, OTELO – Attonaty *et al.*, 1993-, SEPATOU, - Cros *et al.*, 2001-). Les modèles constitutifs d'une application sont parfois intégrés dans une plate-forme qui permet de les manipuler plus aisément (APSIM, -McCown *et al.*, 1996).

Les modélisateurs des systèmes d'intérêt agronomique ont toujours écrit leur code directement dans le langage de programmation choisi. C'est dire que les possibilités offertes par les environnements de génération de code n'ont pas été utilisées, à l'exception récente du modèle Magma (Guerrin, 2001), développé avec Vensim®. Le principe de ces outils est de transformer automatiquement une expression conceptuelle du modèle en un code compilable. L'expression des concepts (structure du système, flux, relations, contraintes) est généralement graphique. Vensim® est particulièrement dédié à la modélisation des systèmes continus, ou vus comme tels. D'autres produits sont dans le domaine universitaire tels OMNet++, ciblé sur les systèmes discrets (Varga, 2001), Modelica, ciblé sur les systèmes physiques (Elmqvist *et al.*, 2000), et adevs, pour les systèmes parallèles (Nutaro, voir 'http://www.ece.arizona.edu/~nutaro/adevs-docs/index.html') ou sur le marché tel OBJECTEERING (voir 'http://www.objecteering.com'), ouvert aux systèmes discrets et continus.

Une évolution possible est de rapprocher les attitudes ci-dessus. D'une part, il existe une tradition de programmation procédurale des phénomènes intervenant dans les systèmes d'intérêt agronomique, justifiée par leur nature essentiellement continue et granulaire : on écrit des modules correspondant aux différents modèles mathématiques des différents phénomènes. D'autre part, une tendance lourde conduit à étudier les interactions entre systèmes de natures variées et sous l'angle de leur pilotage : apparaît alors le caractère discret de la dynamique de tels systèmes. Enfin, une évolution naturelle des travaux de modélisation conduit à une conceptualisation des systèmes d'intérêt agronomique : les concepts ainsi partagés peuvent être la base d'un cadre de modélisation, et l'instrumentalisation de ce cadre constitue un environnement logiciel d'aide à la modélisation. On peut donc envisager, et c'est l'idée à la base de ce travail, la construction d'un environnement de simulation des systèmes d'intérêt agronomique, perçus dans leur nature hybride (continue et discrète), proposant un cadre de modélisation à un certain niveau d'abstraction et permettant le développement distribué et le partage de modules liés à des domaines de connaissance particuliers. Ce cadre de modélisation doit conférer aux produits des qualités qu'on présente dans le paragraphe suivant.

## *1.3 Nouvelles spécifications informatiques pour les modèles*

Les modèles inclus dans les simulateurs d'intérêt agronomique ont des spécifications scientifiques liées à l'objectif qui a justifié leur écriture. On ne les discute pas ici (se reporter par exemple à Brisson *et al.*, 1998). On ne propose ici que des spécifications informatiques, celles qui, quel que soit leur degré d'introduction dans un modèle, ne modifient pas son contenu scientifique.

L'Intelligibilité, la communicabilité, la lisibilité forment un ensemble de spécifications liées, selon lesquelles les connaissances se retrouvent aisément dans le codage du modèle, les erreurs de codage sont faciles à détecter, le code peut être lu par quelqu'un qui ne l'a pas écrit, et celui qui l'a écrit le comprend définitivement.

Un modèle est modulaire et permet l'interchangeabilité si toute connaissance fait l'objet d'un morceau de code aux frontières claires, l'ajout ou le remplacement d'une connaissance n'affecte en rien le reste du code du modèle.

La généricité est la propriété d'un modèle à partir duquel il est possible d'écrire le modèle d'un système plus spécialisé, alors que la généralité est la propriété d'un modèle qui s'applique sans changement à plusieurs systèmes (aux valeurs des paramètres près).

L'interopérabilité d'un modèle est sa capacité à communiquer avec d'autres composants logiciels (y puiser ses entrées, faire connaître son état ou ses sorties).

Un modèle est portable son écriture informatique peut être opérée sur des machines de calcul diverses.

L'efficacité informatique d'un modèle, plus exactement du simulateur associé, c'est le rapport entre le temps simulé et le temps effectif. Le temps simulé, c'est la longueur de la période pendant laquelle on simule le fonctionnement du système (par exemple un cycle cultural). Le temps effectif, c'est le temps que met le calculateur pour exécuter cette simulation.

La liberté, l'ouverture, le caractère partageable sont des qualités telles que le codage informatique du modèle est accessible à ceux qui voudraient le comprendre, l'améliorer, l'étendre, et rendre publics ces changements.

## *1.4 La bibliothèque DIESE*

DIESE est une bibliothèque de classes, fondée sur quelques principes de l'Object Modeling Technique (OMT, -Rumbaugh *et al.*, 1991), et qui tend à réunir cet ensemble de spécifications. Elle est écrite dans le langage de programmation général C++. Du point de vue du modélisateur, c'est un ensemble de composants logiciels réutilisables, autrement dit une 'bibliothèque' (structures de données et procédures de traitement de ces structures). Ces composants encapsulent la connaissance sur la structure et le fonctionnement des systèmes d'intérêt agronomique. On imagine à ce stade qu'elle se situe à un niveau d'abstraction relativement élevé. Ainsi, bien que livrée sous forme de classes C++, la bibliothèque est fondée sur ce qu'on peut appeler un méta-modèle indépendant du langage. La réutilisation consiste à piocher dans l'ensemble les éléments dont on a besoin dans un domaine d'application particulier, à les spécialiser autant que de besoin et à les assembler de manière appropriée. La restriction aux systèmes d'intérêt agronomique correspond au domaine dans lequel DIESE a été validée comme cadre d'analyse et moyen d'expression, et dans lequel l'application de DIESE est d'abord envisagée. La généralité du contenu du méta-modèle permet cependant d'envisager une applicabilité plus large.

Du point de vue du programmeur du simulateur (souvent la même personne), c'est un ensemble de primitives de langage de programmation. Certaines primitives servent à construire en mémoire les structures codant le système. D'autres mettent en oeuvre un moteur de simulation général. Ces primitives doivent être assemblées dans un programme soumis à compilation. DIESE fournit des macro-instructions (séquences standard de primitives) qui simplifient considérablement ce travail d'assemblage.

Du point de vue de la technologie de l'information, DIESE est un outil d'écriture de systèmes à base de connaissance, puisqu'il permet au programmeur de se concentrer sur la déclaration de ses connaissances, la gestion desquelles est prise en charge par un moteur général de simulation. Le moteur est fondé sur le paradigme de la simulation à événements discrets, décrit brièvement dans la section suivante.

Dans la suite de ce texte, on fait une introduction générale au contenu de la bibliothèque, on résume la manière d'en tirer un simulateur, et on donne un exemple d'application pour une comparaison superficielle avec les pratiques actuelles de modélisation. La section 4, par la description plus détaillée des classes, met en lumière en quoi DIESE apporte un gain d'expressivité. La section 5 aborde la question des outils d'interface, qui doivent permettre la création et l'utilisation d'un simulateur à partir d'une expression des connaissances et des données la plus naturelle possible.

# 2 Vue générale et principes d'utilisation

## *2.1 La simulation à événements discrets*

Lors de l'analyse d'un système, on doit évaluer rapidement sa nature discrète, continue ou bien hybride. La nature se réfère à la structure (un peuplement est un ensemble discret de plantes, un sol est un continuum) ou aux changements d'état du système (l'émergence d'une plante est ponctuelle, l'humidification du sol est continue). Si on cherche à simuler le comportement de ce système, on doit décider si la simulation sera de nature discrète, continue ou bien hybride. Il faut en effet distinguer la nature du système et la nature de sa simulation : on peut effectuer une simulation discrète d'un système continu (un horizon de sol est partitionné en couches) ou une simulation continue d'un système discret (croissance et développement d'un peuplement). Enfin, dans le cas d'une simulation discrète, on doit choisir l'un des deux modes de contrôle de la dynamique du système (Evans, 1988). Le contrôle par une horloge consiste à progresser dans le temps selon un pas fixé, l'unité de temps de l'horloge. Pour les systèmes à changements d'état continus, on fait une approximation en plaçant les changements entre deux 'tops' d'horloge sur l'une des deux bornes du pas de temps. Le contrôle par événements consiste à faire progresser dans le temps selon les dates d'occurrence d'événements placés dans un échéancier. Le cas particulier où l'écart de dates entre deux événements est constant équivaut à un contrôle par horloge. Si le contrôle par événements semble approprié aux systèmes discrets, il concerne aussi les continus. En effet, un événement peut être le début ou la fin d'un processus, ou d'une activité, qui change l'état du système de manière continue. On désigne donc par simulation à événements discrets la simulation, discrète et contrôlée par un échéancier d'événements, du fonctionnement d'un système discret et/ou continu.

Les événements placés dans l'échéancier d'une simulation à événements discrets peuvent être de deux natures différentes. Si on perçoit seulement le système comme un ensemble de processus ou d'activités, sa dynamique peut être contrôlée par les événements de début et de fin. Ces événements sont alors des artéfacts, construits pour le besoin du contrôle. Si la perception du système comprend les événements qui surviennent dans son environnement et qui sont analysés comme influant sa dynamique, alors ces événements ne sont pas des artéfacts mais des éléments de connaissance non redondants et qu'on modélise en tant que tels. La conception de DIESE autorise ce dernier point de vue.

## *2.2 Le contenu de la bibliothèque*

### 2.2.1 Quelques éléments sur OMT

OMT est une méthode d'analyse de système et de conception d'un logiciel intégrant un modèle du système. La méthode propose d'assembler les perceptions du système selon trois points de vue : structurel, fonctionnel et dynamique.

Le modèle structurel décrit la structure (statique) du système : ses composants, leurs relations, leurs attributs et enfin les opérations qu'ils subissent ou bien savent faire.

Le modèle fonctionnel décrit ce que fait le système vis-à-vis de son environnement, et ce que font les sous-systèmes vis-à-vis des autres. Ce que fait un système (ses fonctions) passe par les opérations du modèle structurel.

Le modèle dynamique décrit la mise en œuvre des fonctions dans le temps : nature des événements qui déclenchent les fonctions, à quelles conditions sur la structure du système.

### 2.2.2 Les trois classes de base pour la modélisation

La classe BasicEntity enferme le modèle structurel de tout système et de tout composant d'un système (on dira 'entité' de manière générale). Ce qui caractérise une entité, c'est l'ensemble de ses attributs (sans parler encore de leurs valeurs) et les relations qui la lient à d'autres entités. Ces relations sont celles de contenant à contenu ou d'antécédent à image. Les attributs d'une entité sont des descripteurs de son état (grandeurs scalaires ou vectorielles, constantes ou variables) ou bien des procédures qu'on peut lui appliquer. Cette classe permet de représenter une entité qui est un ensemble d'éléments de même nature ou bien une entité composite, c'est-à-dire qui est l'assemblage d'entités de natures différentes. A titre d'exemple, un peuplement végétal est un ensemble de plantes, alors qu'une serre est composée d'un peuplement et d'équipements techniques. On appelle instance de cette classe (et il est sera de même pour les autres classes) une structure de donnée informatique construite en respectant intégralement et seulement le schéma conceptuel qui définit la classe.

La classe Process est le modèle de toute fonction (on dira 'processus') d'un système, ou d'une fonction de l'environnement vis-à-vis du système. Ce sont les processus qui font évoluer l'état du système, de manière instantanée ou progressive. Ils peuvent pour cela invoquer les procédures attachées comme attributs aux entités. L'attribut essentiel d'un processus est sa fonction de transition d'état, qui spécifie le nouvel état dans lequel se trouve le système après sa

mise en œuvre. La même fonction fera l'objet de deux processus différents si elle est assurée de deux manières différentes.

La classe Event code le modèle de tout événement contrôlant la dynamique d'un système. La nature de l'événement n'est pas dans le modèle. Seuls sont représentés la date de son occurrence et ce qu'il provoque en termes de directives sur les processus. Un système ne peut évoluer s'il n'est soumis à aucun événement[1]. Un seul événement peut suffire à déclencher une évolution durable si le processus touché est progressif ou si celui-ci provoque d'autres événements. Plusieurs événements peuvent contrôler le même processus (démarrage, pause, reprise, arrêt). Un événement peut contrôler plusieurs processus.

### 2.2.3 Les classes complémentaires

La classe Descriptor modèlise tout attribut descriptif (on dira 'descripteur') d'une entité (caractéristique statique ou variable d'état dynamique).

La classe Monitor permet de spécifier un mécanisme (on l'appellera un 'démon') déclenché automatiquement lors de l'accès à la valeur d'un descripteur, ou lors de l'établissement de sa valeur, ou encore lors de la modification de la structure d'une entité.

La classe EntitySpec sert à la spécification fonctionnelle d'un ensemble d'entités, c'est-à-dire en les explicitant non pas nommément mais par un ensemble de propriétés à exhiber. Une telle spécification permet de faire dépendre le contenu d'un ensemble des conditions en cours au moment où l'ensemble est requis, sans en changer la définition. De manière générale, la notion de spécification fonctionnelle procure un gain de flexibilité au modèle, par rapport à une spécification explicite.

La classe DescValueSpec sert à la spécification concise d'un ensemble de valeurs, c'est-à-dire sans en expliciter tous les éléments *a priori*. Cette notion est intimement liée à celle de descripteur, car on crée typiquement une instance de DescValueSpec pour spécifier le domaine de valeur d'un descripteur particulier.

La classe Method est le modèle de tout attribut procédural (on dira 'méthode') d'une entité, d'un processus ou d'un événement. Le corps de la méthode est une opération que l'entité subit ou bien sait faire. On voit bien que les processus ne sont pas les seuls objets à contenir une connaissance procédurale. On peut exploiter la connaissance procédurale sur une entité en dehors de toute simulation d'un système incluant cette entité (par exemple en donner une visualisation). Si simulation il y a, elle mettra par contre obligatoirement en jeu des processus, et les connaissances qu'on a sur lui (comment l'exécuter, l'initialiser, le faire progresser, l'arrêter) : ce sont ses méthodes.

La classe Argument est le modèle de tout argument d'une méthode. Il est bien classique dans les langages de programmation courants qu'une fonction possède des arguments (on dit aussi des paramètres formels). Par exemple une fonction de zoom prend en argument de facteur de zoom. La classe Argument joue vis-à-vis de la classe Method le rôle que jouent les paramètres formels vis-à-vis d'une fonction dans un langage de programmation.

La classe SequentialFile encapsule les propriétés d'un fichier externe de données séquentielles (c'est-à-dire dont les enregistrements sont indexés par une date) et la structure des enregistrements. Un exemple typique est le fichier des enregistrements de variables climatiques en un lieu, sur une certaine période et avec une certaine périodicité. DIESE fournit les services de déclaration et de lecture du fichier, tels qu'un changement de la structure du fichier ne se traduit pas par un changement du code de lecture dans le simulateur.

### 2.2.4 Les classes dérivées

Les classes DescriptedEntity et Entity sont des spécialisations de BasicEntity. La première ajoute à BasicEntity une liste de descripteurs constants et/ou variables. La seconde ajoute à DescriptedEntity une liste de méthodes. Une entité qui ne serait caractérisée que par la nature de ses composants ou éléments peut n'être une instance que de BasicEntity. Dès qu'on la dote d'attributs descriptifs ou procéduraux, le modèle de BasicEntity est insuffisant.

Les descripteurs à valeur entière ou réelle, ceux dont la valeur est une chaîne de caractères, ou bien une entité, etc. font l'objet d'une sous-classe particulière de Descriptor. Plus exactement soit de la sous-classe VariableDescriptor soit de la sous-classe ConstantDescriptor selon que le descripteur est susceptible ou non d'être équipé d'un démon.

Une hiérarchie similaire est installée sous la classe Argument, correspondant aux différents types d'arguments : à valeur entière, flottante, etc.

Les méthodes, quant à elles, ont des corps qui renvoient ou non une valeur. Dans la première situation, cette valeur est booléenne, ou bien entière, etc. Une sous-classe correspond à chaque cas.

La classe DiscreteProcess modélise les processus ponctuels, ceux tels que l'état courant et l'état résultant de l'invocation de la fonction de transition se réfèrent au même instant.

[1] Même dans le cas particulier d'un système autonome, il faut déclarer un événement déclenchant une fonction interne au système.

La classe ContinuousProcess est le modèle d'un processus à effet non ponctuel (on le qualifie de continu). Il est caractérisé par un degré croissant de réalisation de l'effet, initié à 0 et limité à 1. La fonction de transition spécifie le nouvel état du système après un intervalle de temps appelé le 'pas', une des caractéristiques du processus. La fonction est invoquée itérativement tant que le degré de progression peut encore croître. Dans un système, des processus continus de pas différents peuvent se succéder ou coexister : se pose alors le problème de leur synchronisation, traité en section 4.

### 2.2.5 Les classes dédiées à la simulation

Une instance de la classe Simulation est la structure de donnée (on dira 'objet', aux deux sens de concept et d'espace qui lui est alloué en mémoire) dans laquelle on spécifie, entre autres choses, la fenêtre temporelle au cours de laquelle on simule le fonctionnement du système. Un attribut important est l'horloge, dont la valeur est l'instant courant dans la fenêtre. Ses opérations essentielles sont la mise en œuvre du moteur de simulation (dont la gestion de l'agenda d'événements attaché à l'instance).

DIESE fournit des classes dont les méthodes permettent de manipuler le temps : calculs de durées, changements d'unité, affichage.

La classe Parameter permet de déclarer des paramètres globaux, c'est-à-dire dont la valeur est accessible par tout composant du système (notamment les entités) et par toute fonction. DIESE fournit les services d'accès à la valeur des paramètres.

La classe OutputSpecification permet de poser une requête d'information en sortie de la simulation. Les informations que l'utilisateur peut requérir portent sur l'évolution de la structure ou de telle ou telle variable d'état de telle ou telle entité ou ensemble d'entités, ou bien sur les événements qui ont sont survenus au cours de la simulation. On crée autant d'objets que d'informations requises. On peut ainsi, sans changer le code du simulateur, enchaîner deux simulations avec les mêmes entrées mais des sorties différentes, ou plus ou moins détaillées.

### 2.2.6 Le graphe OMT des classes de DIESE

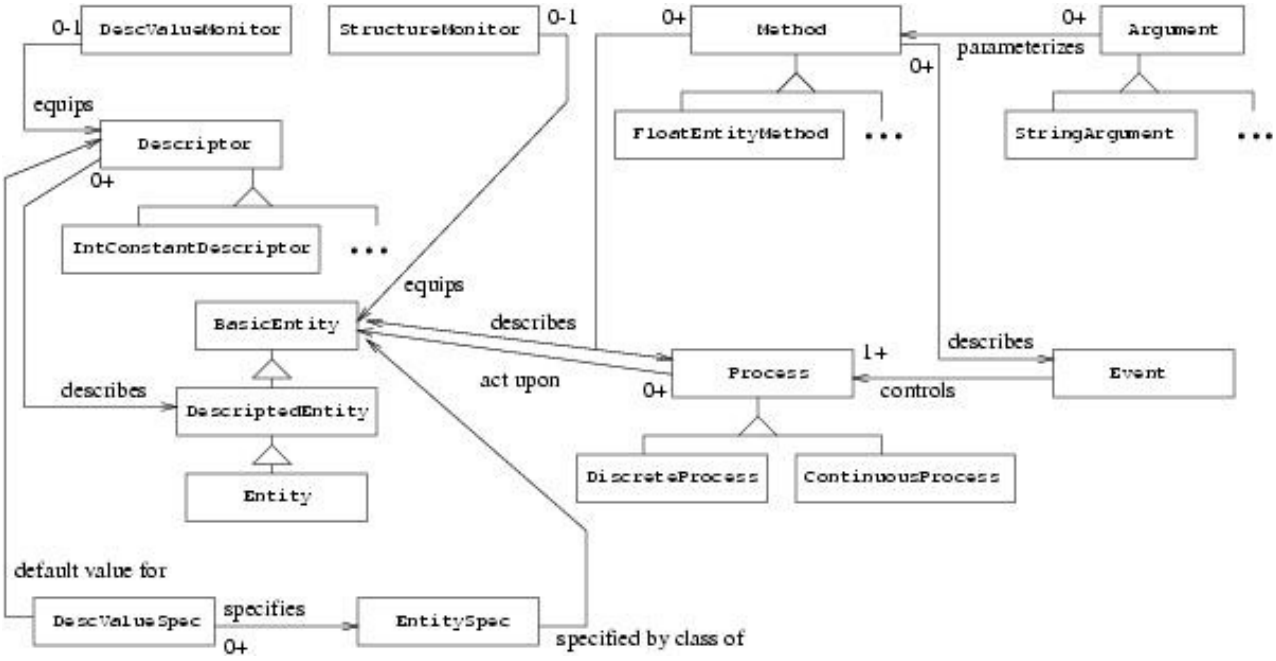



Figure 1 : le graphe OMT simplifié des classes de DIESE.
L'objet d'une relation est pointé par la flèche, son sujet est au point de départ. L'arité par défaut est 1.

## *2.3 Utilisation de la bibliothèque*

On se place dans un scénario d'analyse dont les acteurs sont un expert du domaine à modéliser et un ingénieur de la connaissance qui prend en charge le codage du simulateur. En fait, une connaissance de base du langage C++, l'expérience acquise et l'utilisation de l'outil d'interface (cf. section 5.1) doivent permettrent à l'expert de coder lui-même

le simulateur. On suppose un cycle simple de développement, débutant par la conception puis le codage du modèle et se terminant par l'utilisation du code (la simulation). En réalité, il existe souvent des retours de l'utilisation vers le codage voire la conception. En outre, pour se situer au niveau des principes, on décrit la phase de codage dans sa forme archaïque, celle qui, n'exploitant pas les outils d'interface décrits en section 5, demande au programmeur une prise en charge complète du codage.

### 2.3.1 La modélisation d'un système dans un domaine

#### *2.3.1.1 Identification des classes du domaine, dérivées des classes de la bibliothèque*

Dans cette première étape, on s'intéresse d'abord aux entités : on identifie une sous-classe de BasicEntity pour chaque terme, portant le sens d'un élément structurel, trouvé dans le discours de l'expert (après élimination des synonymes). On recherche les attributs caractéristiques de chaque sous-classe : ils feront l'objet d'un descripteur (à valeur constante ou variable) ou d'une méthode.

On traite ensuite les processus et les événements : l'analyse des aspects fonctionnels et dynamiques est conjointe. On peut détecter un événement dans le discours de l'expert et chercher son incidence sur le système (quel processus est initié, ou stoppé, ou mis en pause), ou bien détecter une fonction (et donc identifier une sous-classe de Process) et préciser ensuite la dynamique de sa mise en œuvre (créer une sous-classe d'Event).

#### *2.3.1.2 Spécification des liens entre classes dérivées*

On se demande si une entité est un cas particulier d'une autre, ou bien se situe à un niveau d'abstraction plus élevé. On détecte en quoi deux entités spécialisent le même concept de deux manières différentes. On établit ainsi les liens de nature hiérarchique.

Quand on contraint ce que doit être la classe des éléments d'une entité, quand on fixe la nature des composants d'une entité, on établit des liens structurels.

Pour établir un lien relationnel, on attache à une classe d'entités un descripteur dont la sémantique est une relation entre l'entité (l'antécédent) et une autre (l'image). La valeur du descripteur est l'image. Par exemple la relation 'a pour génotype' se code par un descripteur 'variété' attaché à la classe 'plante' : dans les instances, sa valeur sera l'entité encapsulant les propriétés de la variété.

Un lien fonctionnel est celui qui unit un processus à un composant particulier du système, ou au système lui-même, en tous cas l'entité sur laquelle porte le processus. On peut contraindre la classe du composant sur lequel porte le processus.

Un lien dynamique, c'est, pour chaque événement, l'indication de ce qu'il provoque, c'est-à-dire un couple 'directive/processus', où la directive est le démarrage, la continuation ou l'arrêt.

### 2.3.2 Le codage

#### *2.3.2.1 Codage des classes du domaine*

Essentiellement, c'est la déclaration et ce qu'on appelle la réalisation (le tout en C++[2]) des constructeurs des classes. On déclare de quelle classe (BasicEntity ou une de ses sous-classes, Process, Event) celle en cours d'écriture hérite les propriétés. Puis on utilise, dans la réalisation, les primitives fournies par DIESE pour attacher des descripteurs et des méthodes, et de manière générale pour spécifier toute sorte de lien. Pour attacher un descripteur (ou une méthode), il faut au moins l'avoir déclaré (préalablement).

#### *2.3.2.2 Codage du modèle d'un système particulier*

Un système particulier est un exemplaire concret du système général qui n'existe que conceptuellement. Le codage consiste à assembler certaines classes du domaine (générales) et un 'module principal' écrit spécifiquement pour ce système particulier.

Dans le 'module principal', on commence par créer des instances des classes générales. Les instances sont des structures de données auxquelles sont alloués des espaces référençables en mémoire. La création des instances se fait par appels aux constructeurs. C'est à ce stade qu'on attribue des valeurs aux descripteurs (possiblement à partir d'une spécification par défaut au niveau de la classe, et en exploitant éventuellement les valeurs des paramètres –*cf.* §2.3.2.3.1). Les constructeurs des événements peuvent, dès la phase de conception, encapsuler la construction d'un

[2] Toute fonction (et aussi toute variable) doit être déclarée avant toute utilisation de son nom. La première utilisation ne tient pas lieu de déclaration. La déclaration, c'est une ligne du programme avec le nom de la fonction, le type de la valeur qu'elle renvoie et la liste des types se des arguments. C'est la signature de la fonction. Il n'est pas nécessaire à ce stade d'en écrire le corps. La réalisation, c'est l'écriture complète de la fonction (signature et corps).

processus de la classe adéquate. Dans ce cas, le programmeur n'a pas à invoquer lui-même le constructeur du processus en question.

On établit ensuite les liens entre les instances. On utilise les services fournis par DIESE pour ajouter des éléments ou des composants aux instances, pour désigner les entités opérées par les processus et, éventuellement, pour associer aux événements les couples directive/instance de processus.

#### *2.3.2.3 Codage du simulateur*

Présenté ici séparément, ce codage est en fait intimement lié à celui du modèle du système particulier. C'est en effet l'ajout au 'module principal', des instructions de mise en œuvre du moteur de simulation sur le modèle du système particulier.

Il s'agit d'abord de l'ajout des instructions de lecture de fichiers de données. Ce sont des services fournis par DIESE, qui lisent le fichier des paramètres (ce qui provoque la création d'instances de Parameter), et celui des spécifications de sortie (ce qui provoque la création d'instances d'OutputSpecification). Chacun de ces deux fichiers peut être vide.

On crée ensuite une instance de la classe Simulation, par invocation du constructeur de la classe, puis on attribue des valeurs aux descripteurs définissant la fenêtre temporelle de simulation, on détermine l'unité de temps de l'horloge, c'est-à-dire l'intervalle de temps atomique (*e.g.* une seconde, cinq minutes) correspondant à une incrémentation de la valeur d'horloge. Enfin, on attache un agenda d'événements initial, contenant tout (généralement) ou partie des instances d'événements qu'on vient de créer, rangées par ordre croissant d'instant d'occurrence.

Vers la fin du programme, on ajoute l'instruction de lancement du moteur de simulation. C'est simplement le message[3] Run() adressé à l'instance de Simulation. Cette méthode est le point d'entrée dans un ensemble de fonctions du moteur qui gèrent l'agenda d'événements (dépilement/empilement, mise en œuvre des directives sur les processus, mise en œuvre de la fonction du processus sur l'entité désignée, synchronisation de l'avancée des processus concomitants).

### 2.3.3 La simulation

La phase combinant modélisation et codage concerne la production du simulateur. La phase de simulation concerne son utilisation. Il est important de faire en sorte que les deux phases puissent être conduite par des agents différents, même si ce n'est pas toujours le cas dans la réalité.

Pour la préparation des fichiers de données, c'est l'utilisateur qui, d'une simulation à l'autre, établit, puis change au besoin, la valeur des paramètres et ses spécifications de sortie. On notera ici qu'un simulateur construit avec la bibliothèque DIESE ne peut pas envoyer de requête à un système de gestion de bases de données.

On lance ensuite au système d'exploitation d'une commande standard. De manière universelle, son premier terme est le nom d'un module exécutable et les suivants sont des paramètres. Ici, le module exécutable est le résultat de la compilation du simulateur. Les paramètres de l'appel sont les noms des fichiers de paramètres et de spécifications de sortie :

```
main fichier_parametres fichier_specifications_sortie
```

Noter que le nom du répertoire dans lequel on veut retrouver les fichiers en sortie doit être la valeur d'un paramètre standard. L'exécution de la commande provoque donc la lecture des fichiers de données et le lancement du moteur de simulation. La simulation est terminée quand la commande-système (`main`) a été exécutée par le processeur. Une nouvelle commande peut alors être lancée au système d'exploitation, notamment pour inspecter les résultats ou pour relancer une simulation à partir de nouvelles données en entrée.

[3] Un objet (émetteur) adresse un message à un autre (récepteur) quand le corps d'une fonction attachée à l'émetteur invoque une fonction attachée au récepteur.

# 3 Exemple : un modèle de peuplement de plants de tomate (TOMMY)

TOMMY est, d'une part, le résultat de la re-ingénierie du modèle TOMGRO dans le cadre fourni par DIESE et en utilisant ses services[4]. TOMGRO a déjà fait l'objet d'une telle tentative, GPSF (Gauthier *et al.*, 1999), qui a visé un modèle dont la généralité était limitée aux peuplements cultivés, pour ne pas être fondé sur un méta-modèle indépendant de l'application. Le contrôle temporel de la simulation y repose classiquement sur une double boucle d'itération. Il est important de noter que l'analyse du système qui fonde TOMGRO et GPSF a été conservée dans TOMMY. Seuls la conception du logiciel et évidemment son codage sont, nettement, différents. Si toutes les connaissances incluses dans TOMGRO ont été reprises dans TOMMY, celui-ci comporte en plus un modèle physique de la serre (notamment fonction du climat extérieur) et un modèle de régulation (déterminant les actions par lesquelles on contrôle le climat intérieur). On donne ici un simple aperçu du modèle à la base du simulateur et des modules de programmation dont celui-ci est constitué. Cette illustration de la section 2 pourra aussi être exploitée lors de la lecture de la section 4.

## *3.1 Principaux éléments du modèle*

Les entités placées aux niveaux d'intégration successifs de la structure sont les suivantes. Le système biophysique est constitué de une ou plusieurs serres. Une serre abrite le peuplement et les équipements de serre. Un peuplement est un ensemble de plantes, elles-mêmes composées d'organes. Les organes d'une plante sont de plusieurs types : l'axe et le système racinaire, mais aussi, comme éléments d'un axe, les sympodes, eux-mêmes composés d'un élément de tige, d'un feuillage (un ensemble de feuilles) et d'un bouquet (un ensemble de fruits). Les équipements sont les systèmes d'aération (les ouvrants), de chauffage, de brumisation, d'alimentation en gaz carbonique. La serre est caractérisée par un climat interne, et est placée dans un environnement modélisé par la météo extérieure. A noter que la classe des plantes (figure 2) est spécialisée en la classe des plantes à fruits (les deux certainement exploitables dans les applications sur d'autres espèces) puis en celle des plants de tomate, par l'addition de descripteurs spécifiques.

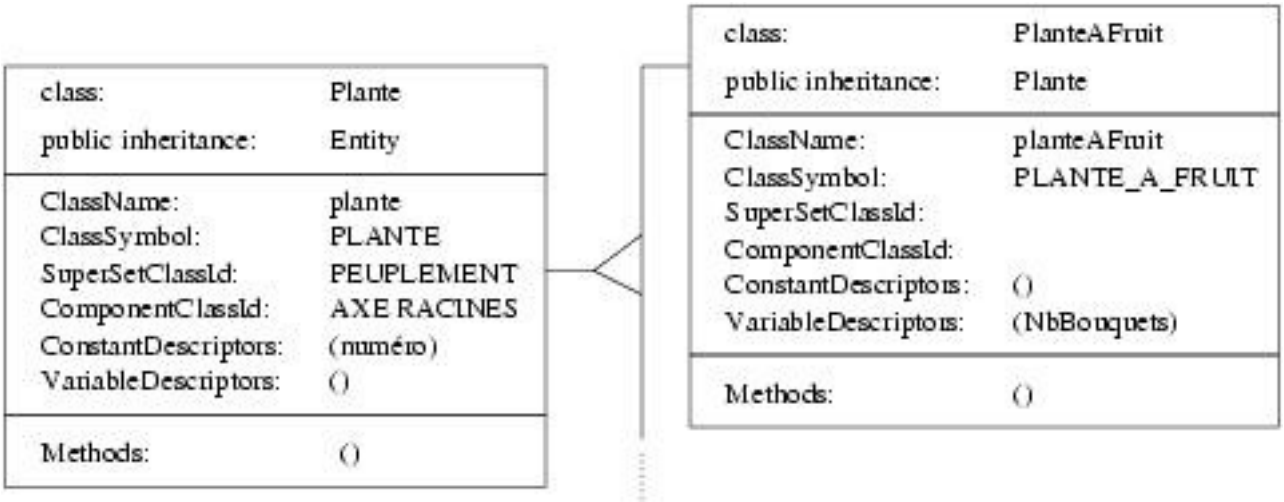



Figure 2 : la représentation OMT des classes des plantes et des plantes à fruits.

Comme résultat de l'analyse originelle, les événements qui rythment la dynamique sont simplement trois événements artificiels survenant en début de simulation. Ils ne font qu'initier et lancer trois processus de pas différents : le jour, l'heure et le quart d'heure. Chacun de ces processus encapsule un certain groupe de fonctions : celles pour lesquelles il est nécessaire et suffisant de mettre à jour l'état du système (résultant de leur effet) à un rythme correspondant au pas du processus. Ainsi, le processus nommé EvolutionQuartHeure encapsule le modèle de régulation et le modèle physique de la serre. A l'opposé, le processus nommé EvolutionJournaliere encapsule entre autres le modèle d'initiation des organes et celui de répartition de la matière sèche. Au total, ce sont quatorze processus qui agissent en parallèle sur le système.

## *3.2 Le programme du simulateur*

La figure 3 donne le code de la classe des plantes à fruits correspondant à l'analyse de la figure 2. Les trois premières lignes du constructeur sont tout à fait standardisées : elles identifient la classe et l'installent dans la hiérarchie. On fait

[4] Ce travail a été réalisé au sein du laboratoire de bioclimatologie de l'INRA à Avignon (France) sous la direction de Ch. Gary et M. Tchamitchian, avec la contribution essentielle de B. Montbroussous.

ensuite référence à une classe de descripteurs (NbBouquets) : c'est la particularité des plantes à fruits par rapport à la classe « mère » des plantes.

```
class PlanteAFruit : public Plante {
  public PlanteAFruit();  //
constructeur
}

PlanteAFruit::PlanteAFruit() {
   Identify(PLANTE_A_FRUIT,
"planteAFruit");
   SetParentclassId(PLANTE);
   AddInstanceToClass();
   Descriptor* pD = new NbBouquets();
   AddVariableDescriptor (pD);
};
```

Figure 3 : la déclaration et le constructeur de la classe des plantes à fruits.

La figure 4 donne le code d'une fonction de transition d'état activée par un processus. Le processus est celui qui modélise l'évolution du climat à l'extérieur de la serre. La fonction établit les nouvelles valeurs du climat à la fin du pas du processus. Le pas a été dit égal à la périodicité des enregistrements dans le fichier des donnés climatiques (ici 15 minutes) pour que chaque lecture d'enregistrement fournisse les données pour la valeur courante de l'horloge. Ce code illustre la manière d'accéder aux composants du système et modifier leur état. Les deux premières lignes, tout à fait standardisées, renvoient l'entité opérée par le processus (*cf.* §4 .2.1) ici une serre. La troisième accède à l'ensemble dont la serre est un élément. La quatrième ligne renvoie la valeur du descripteur du système biophysique qui pointe sur son environnement, avec une sémantique de condition météorologique. Les dernières instructions attribuent des valeurs aux descripteurs de la météo à partir des champs de l'enregistrement météo (*cf.* §4.8).

```
void lectureMeteoOnestep_Body(ProcessMethod* pM) {
     ContinuousProcess* pCP = pM->DescribedContinuousProcess();
     Entity* pSerre = pCP->ProcessedEntity();
     Entity* pSBP = pSerre->GetSuperSet();
     Entity* pMeteo = pSBP->GetEntityValue(ENVIRONNEMENT);
     int k = ReadNextEnrgt(FILE_METEO);
     if (k<0) { cout <<"\n!!! End of FILE_METEO !!!\n"; exit(0); }
     pMeteo->SetDoubleValue(HR,GetDoubleFieldValue(FILE_METEO,FIELD_HR));
     pMeteo-
>SetDoubleValue(V_VENT,GetDoubleFieldValue(FILE_METEO,FIELD_V_VENT));
     pMeteo->SetDoubleValue(RG,GetDoubleFieldValue(FILE_METEO,FIELD_RG));
     pMeteo-
>SetDoubleValue(TMP_A,GetDoubleFieldValue(FILE_METEO,FIELD_TMP_A));
};
```

Figure 4 : le corps de la méthode de progression du processus de lecture de la météo.

La figure 5 montre les éléments essentiels du programme principal. On rappelle qu'on a pris l'option de présenter la forme archaïque du code, celle qui n'exploite par les outils d'interface présentés en section 5. On verra alors que le code est non seulement plus simple, mais automatiquement généré par l'interface, pour une fonctionnalité bien sûr identique. Les deux premières instructions lisent les deux fichiers dont les noms ont été placés en argument de la commande. Certains attributs de l'instance de la clase Simulation prennent des valeurs lues dans le fichier des paramètres. On installe ensuite le système simulé en créant les entités et en établissant leurs liens. On voit notamment que l'instance de la classe Meteo est la valeur d'un descripteur du système biophysique, ce qui justifie la façon d'y accéder (GetEntityValue) dans le corps de la méthode en figure 4. Ensuite, la figure n'indique complètement que la manière d'insérer dans l'agenda l'événement qui met en œuvre les processus dont la pas est 15 minutes. Le code pour les autres se complète facilement. Noter l'argument de la fonction qui fixe le pas : sa valeur est de une seule unité d'horloge, puisque la valeur de cette-ci est justement 15 minutes. Le programme se termine par des instructions standards.

```
main (int argc, char* argv[]) {
  //
  // Préambule : lecture paramètres et spécifications de sortie
  // ------------------------------------------------------------
  ParseParameterFile(argv[1]);
  ParseOutputSpecFile(argv[2]);
  //
  // Initialisation (phase 1) : paramètres de la simulation
  // ------------------------------------------------------
  pCurrentSim = new Simulation();
  pCurrentSim->ClockTimeUnit(MINUTE);
  pCurrentSim->ClockTimeUnitQuantity(15);
  Date dd;
    dd.mday = GetIntParameterValue("DateDebut","Jour");
    dd.month = GetIntParameterValue("DateDebut","Mois");
  pCurrentSim->BeginningDate(dd.mday, dd.month, 1994);
  Date df;  // idem dd
  ...
  //
  // Initialisation (phase 2) : le système simulé
  // --------------------------------------------
  Meteo* pMeteo = new Meteo();
  SystemeBioPhysique* pSBP= new SystemeBioPhysique();
  pSBP->SetEntityValue(ENVIRONNEMENT, pMeteo);
  Serre* pSerre = new Serre();
  pSBP->AddElement(Serre);
  ...
  pCurrentSim->SimulatedObject(pSBP);
  //
  // Initialisation (phase 3) : l'agenda d'événements
  // ------------------------------------------------
  EvolutionQuartHeure* pEvolutionQuartHeure = new
EvolutionQuartHeure();
  pEvolutionQuartHeure->Pas(1);
  pEvolutionQuartHeure->ProcessedObject(pSerre);
  pEvolutionQuartHeure->PostInitAndProceedEvent(0,4);
  //
  EvolutionHoraire* pEvolutionHoraire = new EvolutionHoraire();
  pEvolutionQuartHeure->Pas(4);
   ...
  //
  EvolutionJournaliere* pEvolutionJournaliere = new
EvolutionJournaliere();
  pEvolutionQuartHeure->Pas(96);
  ...
  //
  // la simulation, proprement dit
  // -----------------------------
  pCurrentSim->Run();
  //
  // Epilogue  : - dés-allocation des entités du système
  //             - sauvegarde des résultats sur fichiers
  // -------------------------------------------------------
  DeleteDomainObjectInstances();
  DisplayEntityOutputSpecs(O_ENTITY);
  ...
  delete pCurrentSim;
}
```

Figure 5 : le code partiel du programme principal de TOMMY

### *3.3 Exemple d'entrées/sorties*

On souhaite simuler l'évolution du système composé d'une serre abritant un peuplement de tomate, entre le 1er octobre 1994 et le 31 juillet de l'année suivante, dans un environnement météorologique décrit par les enregistrements du fichier meteo_15mn.txt, et sous une stratégie de régulation sous la forme des consignes enregistrées dans le fichier consigne.txt. Ces informations sont programmées dans le module principal (*cf.* figure 5). Un extrait du fichier des paramètres du système est en figure 6.

```
...
DateDebut       Jour            <- 1 ;
DateDebut       Mois            <- 10 ;
DateFin         Jour            <- 31 ;
DateFin         Mois            <- 7 ;
Bouquet         NBFPT           <- 5;
PlantDeTomate   PLAR            <- 0.1;
Peuplement      PLM2            <- 2.2;
...
```

Figure 6 : extrait du fichier des paramètres du système, en entrée du simulateur TOMMY.

On souhaite étudier, sur les bouquets, l'évolution parallèle du nombre de fruits et du poids. On écrit donc les spécifications de sortie telles que dans la figure 7. La figure 8 montre un extrait du fichier bouquet.txt, désigné dans les spécifications de sortie. Le temps d'exécution a été de 32 secondes sur une machine monoprocesseur cadencée à 8Ghz, dotée de 8 GO de RAM et d'un disque SCSI U160.

```
SAVE DESCRIPTOR     "bouquet"  ALL   "bouquet.txt"   NEW
    DATE_DMY
         "dw"       3
         "nbFruits" 0;
```

Figure 7 : le fichier des spécifications de sortie, en entrée du simulateur TOMMY.

```
#bouquet        day   month year  dw
     nbFruit
...
bouquet_47      18    10    1994  0.011
     1
bouquet_47      19    10    1994  0.011
     1
bouquet_47      20    10    1994  0.040
     1
bouquet_47      21    10    1994  0.140
     2
bouquet_47      22    10    1994  0.185
     2
bouquet_47      23    10    1994  0.269
     2
bouquet_47      24    10    1994  0.469
     3
bouquet_47      25    10    1994  0.722
     3
bouquet_47      26    10    1994  0.851
     4
bouquet_47      27    10    1994  1.088
     4
bouquet_47      28    10    1994  1.227
     4
bouquet_47      29    10    1994  1.619
     5
bouquet_47      30    10    1994  1.969
     5
bouquet_47      31    10    1994  2.156
     5
...
```

Figure 8 : un extrait du fichier des valeurs demandées dans les spécifications de sortie.

# 4 Description détaillée des classes de DIESE

On n'illustre ici que les services accessibles à l'utilisateur, c'est-à-dire pas les services non publics utilisés pour le codage interne de la bibliothèque. Lorsqu'on décrit une classe, on liste les caractéristiques communes à tous les objets (les instances de la classe) qui correspondent exactement au concept véhiculé par la classe (par exemple le concept d'entité). Ces propriétés sont de deux natures différentes. En premier lieu, le fait même de posséder tel ou tel attribut est une caractéristique. En second lieu, deux ensembles d'objets qui possèderaient le même ensemble d'attributs devraient tout de même relever de deux classes distinctes si les valeurs admissibles pour les attributs étaient différentes dans les deux ensembles.

## *4.1 Les classes d'entités*

Dans tout système, on peut identifier des composants de natures différentes, qui relèvent donc *a priori* de classes différentes. DIESE se conforme à cette vue, mais impose que toutes les classes d'entités ainsi conçues soient des spécialisations d'une classe unique, la classe BasicEntity. Les deux justifications de cette règle sont, d'une part, qu'il existe à un niveau conceptuel supérieur des différences entre les notions d'entité, processus et événement, qu'il faut bien capter dans des classes différentes. D'autre part, la reconnaissance de propriétés communes à toutes les entités, de quelques natures qu'elles soient, autorise l'universalité du traitement de ces propriétés, c'est-à-dire la proposition de services communs à toutes les classes. Plus encore, DIESE assure que toute propriété d'une classe d'entités dans un domaine d'application quelconque figure dans les propriétés communes : la classe peut ainsi être entièrement gérée avec les services fournis par DIESE.

Les propriétés de toute entité sont ainsi les suivantes, exclusivement. On évoque au fur et à mesure la sous-classe au niveau de laquelle sont attachées ces propriétés et les principaux services qu'offre DIESE pour les gérer.

- La double identification de la classe, par un nombre entier (appelé le 'symbole' de la classe, parce que C++ permet facilement d'associer un symbole à un entier : par exemple pour la classe Axe de TOMMY, l'entier est 112 correspond au symbole AXE) et une chaîne de caractères (qui est le 'nom' de la classe : pour la classe Axe, le nom est « axe »). Ces deux propriétés, attachées à BasicEntity, prendront la même valeur dans toutes les instances de la classe. On doit affecter des valeurs à ces deux propriétés (exclusivement dans le constructeur de la classe) et demander leur valeur.

- Au niveau des DescriptedEntity, deux listes de descripteurs (les constants et les variables). La valeur de la propriété, c'est la nature des éléments de la liste, c'est-à-dire la sémantique des attributs, et non leurs valeurs. Deux entités de la même classe sont donc distinguables, et c'est normal, non par la liste des descripteurs, mais par leurs valeurs. Ces listes peuvent être vides. On peut ajouter un descripteur, accéder à un descripteur désigné par son symbole ou son nom, donner une la valeur à un descripteur désigné par son symbole, ou accéder à cette valeur.

- Au niveau des Entity, une liste des méthodes, à laquelle s'appliquent les remarques ci-dessus. Noter cependant que deux instances de la même classe ne sont généralement pas distinguées sur la valeur des éléments, puisque cette valeur est ici une procédure. On peut ajouter une méthode, donner un corps à une méthode désignée par son symbole, donner une valeur à un argument d'une méthode, désigné par son symbole. On peut enfin demander l'exécution du corps d'une méthode et en récupérer la valeur éventuellement renvoyée.

- Au niveau des BasicEntity, une référence éventuelle à une entité de nature ensembliste et qui contient l'entité décrite comme élément, et une référence éventuelle à une entité composite et qui contient l'entité décrite comme composant. Les valeurs sont automatiquement attribuées lors de la désignation de l'entité décrite comme élément ou composant (voir plus loin). On peut accéder à ces valeurs. On peut contraindre une entité ensembliste à être une instance d'une certaine classe.

- Toujours au niveau des BasicEntity, l'une ou l'autre, mais jamais les deux, des deux indications suivantes : la nature (unique) des éléments ou bien une liste indiquant la nature de chacun des composants. Une entité de nature ensembliste est donc en outre caractérisée par la liste de ses éléments. On peut ajouter un ou plusieurs éléments et en retirer un. Accéder à un élément, désigné par son nom ou son rang. Connaître le nombre d'éléments. On peut contraindre les éléments à être des instances d'une certaine classe. Une entité composite est, de son côté, caractérisée par la liste des composants eux-mêmes. On peut installer un composant (DIESE vérifie que son symbole est dans la première liste) et en remplacer un par un autre de la même classe. Accéder à un composant, désigné par son nom ou son symbole de classe.

D'autres types de services sont offerts, tels l'affichage de l'entité (structure et/ou valeurs des attributs), et ceux qui gèrent la place de la classe d'entités dans la hiérarchie des classes. Sur ce dernier titre, on peut par exemple savoir si une entité est ou non une instance, directe ou non, d'une classe désignée par son symbole. Savoir si une classe est l'ancêtre

ou la descendante d'une autre. Ces services sont accessibles parce que DIESE gère, de manière opaque à l'utilisateur, un réseau d'objets encapsulant les connaissances sur les classes elles-mêmes[5].

## *4.2 Les classes de processus*

### 4.2.1 Considérations générales

Un processus est essentiellement caractérisé par l'effet qu'il produit sur l'état du système, autrement dit sa structure et la valeur de ses variables d'état. Les processus identifiés dans un domaine ayant, normalement, des effets différents, ils font l'objet de sous-classes distinctes de la classe de base Process et, plus précisément, soit de la classe DiscreteProcess soit de la classe ContinuousProcess. L'effet est codé par le modélisateur dans le corps d'une fonction (la fonction de transition d'état). Ce corps modélise l'effet complet (cas d'un processus ponctuel) ou bien une partie de l'effet (cas d'un processus continu). DIESE offre une commande simple, à utiliser dans le constructeur de la classe applicative, pour doter ses instances d'une telle fonction. La mise en œuvre de la fonction de transition d'état est provoquée par l'occurrence d'un événement à un instant *t*.

Dans le cas d'un processus ponctuel, cet événement inclut une directive dite d'exécution sur le processus. Le moteur de simulation invoque le corps de la fonction (en fin d'exécution, l'horloge indique toujours *t*), puis s'occupe de l'événement suivant (qui survient en *t* ou plus tard).

Dans le cas d'un processus continu, cet événement inclut une directive dite de progression du processus. Le moteur de simulation invoque le corps de la fonction (en fin d'exécution, l'horloge indique toujours *t*). A ce stade, le moteur regarde si l'événement suivant survient avant l'instant *t+p*, où *p* est le pas du processus. Si c'est le cas, le moteur fait intervenir cet événement prioritaire. Mais il a pris soin de replacer dans l'agenda, en l'instant *t+p*, un nouvel événement avec la directive de progression sur le processus. S'il n'y a pas d'événement prioritaire, le moteur regarde si l'effet est réalisé complètement. (lire quelques détails plus loin). Si c'est le cas, il s'occupe de l'événement suivant (qui survient au plus tôt en *t+p*). Si l'effet est incomplètement réalisé, le moteur avance l'horloge de la valeur *p* et invoque à nouveau le corps de la fonction de transition d'état, à partir de l'état laissé par l'invocation précédente. Le moteur examine à nouveau l'agenda, et ainsi de suite, jusqu'à ce que l'effet soit complètement réalisé. On comprend que le corps de la fonction de transition d'état doit n'exprimer que le changement effectué en un pas, à partir de l'état en début de pas, et non le passage à l'état terminal, caractérisant l'effet complet, à partir de l'état initial. Noter que seuls les états en début et en fin de pas sont susceptibles d'être exploités par les autres fonctions du simulateur : si la transition entre début et fin de pas passe par des états intermédiaires, ceux-ci restent privés à la fonction. Le modélisateur doit donc donner au pas (et adapter la fonction de transition de manière cohérente) la plus grande valeur (par mesure d'économie) telle que toute état dans l'intervalle n'a pas plus d'intérêt que l'un ou l'autre des états en début et en fin de pas.

En réalité, un processus continu est aussi doté de deux autres fonctions, elles aussi écrites par le modélisateur. La première a pour effet de placer le système dans l'état initial requis par la progression ultérieure : c'est l'état en début du premier pas. La seconde sert à arrêter le processus, même si l'effet n'est pas complètement réalisé. Cette fonction peut être invoquée n'importe où dans le code. Dans son corps, le modélisateur inclut obligatoirement une commande de DIESE qui indique à l'algorithme de progression ci-dessus qu'il doit s'arrêter, mais aussi toute instruction complémentaire appropriée dans la situation. Un processus peut aussi être arrêté en plaçant dans l'agenda un événement avec une directive d'arrêt, mais par cette voie le processus sera simplement stoppé, sans mise en œuvre d'instructions complémentaires.

Enfin, tout processus peut être doté d'une fonction exprimant une précondition à sa mise en œuvre. Un processus ponctuel ne peut être exécuté, et un processus continu ne peut être initialisé, que si sa précondition est satisfaite. Une fois initialisé, un processus continu progresse sans examiner à nouveau la précondition.

DIESE offre naturellement la possibilité de désigner l'entité, composant du système ou le système lui-même, sur laquelle est réalisé l'effet du processus, ou bien qui est regardée par la précondition. On l'appelle l'entité opérée, et c'est un attribut de tout processus. Cette entité n'est en fait qu'un point de référence dans le système : certes, c'est généralement l'état de cette entité que le modélisateur choisira de modifier dans le corps de la fonction de transition d'état, mais rien n'empêche d'en modifier d'autres aussi ou encore d'autres seulement, par exemple ses entités éléments.

### 4.2.2 Le mécanisme de synchronisation des processus continus

La possibilité de donner aux différents processus continus des pas qui leurs sont propres est une mesure de lisibilité du modèle et d'économie. Dans les simulateurs structurés autour d'une boucle, c'est le processus le plus exigeant (celui de pas le plus court) qui impose son rythme de progression aux autres, lesquels sont artificiellement maintenus constants lors de certaines itérations : cela impose des tests, et rend le code peu naturel et moins commodément extensible. Il est possible de faire « tourner » économiquement des processus de pas différents à la condition forte qu'on puisse imbriquer

[5] Il faut (on peut voir cela comme une contrainte) mais il suffit au programmeur d'insérer dans le constructeur les invocations aux fonctions AddInstanceToClass et SetHierarchy.

des boucles correspondant aux différents pas (par exemple, une boucle qui change les jours, une en dessous qui change les heures). Le système de boucles est un mécanisme de synchronisation qui a son équivalent fonctionnel dans DIESE. C'est un des éléments importants du moteur de simulation. Il fonctionne de la manière suivante, en considérant un cas particulier pour une meilleure illustration[6].

Soit une simulation qui commence en l'instant 0 et deux processus p1 et p2 avec des pas respectifs de 5 et 4, initiés respectivement en 0 et 2. p1 provoque un changement d'état en 0, 5, 10, etc. p2 en provoque un en 2, 6, 10, etc. On programme deux événements, l'un initie p1 en 0 et le lance, l'autre fait de même pour p2 en 2. En 0, l'effet de p1 progresse et l'horloge marque virtuellement 5. Le prochain événement est l'initialisation de p2 en 2 : l'horloge passe à 2, on programme en 5 la continuation de p1 et on réalise l'initialisation de p2 et sa première progression. L'horloge marque virtuellement 6. La continuation de p1 demandée en 5 est prioritaire : l'horloge passe à 5, on programme en 6 la continuation de p2 et p1 progresse. L'horloge marque virtuellement 10. Un événement de progression de p2 est programmé en 6. Il est prioritaire : l'horloge passe à 6, on programme en 10 la continuation de p1 et on fait progresser p2. L'horloge marque encore virtuellement 10. En 10, deux progressions doivent intervenir : celle de p1, déjà programmée dans l'agenda, et celle de p2, en continuation de la dernière effectuée en 6. Le moteur doit décider laquelle est prioritaire : il le fait sur la base d'un des attributs des événements : la priorité (cf. ci-dessous). Dans les deux cas, l'horloge passe à 10. Si la progression de p1 est prioritaire, le moteur programme un événement de progression de p2 en 10 et fait progresser p1. Dans le cas contraire, la progression de p2 est continuée, puis l'horloge passe virtuellement à 14, ce qui entraînera la progression immédiate (programmée en 10) de p1, et ainsi de suite.

## *4.3 La classe des événements*

On l'a vu dans la section précédente, un événement doit intervenir pour qu'un processus réalise son effet. Inversement, un événement ne peut toucher la dynamique d'un système que par l'intermédiaire d'un processus, ce qui explique que le couple directive/processus est un attribut essentiel d'un événement. On développera donc ici un autre aspect qui est celui de la spécification de l'occurrence des événements.

On rappelle auparavant que la nature de l'événement associé à un processus n'est pas un élément du modèle du système, car cette connaissance ne porte que sur l'environnement. Seule la connaissance sur l'effet est utile. Par exemple, il n'est pas utile de savoir que c'est le lever du soleil qui provoque, chaque début de jour, de démarrage d'un processus de photosynthèse. En conséquence, il n'est pas nécessaire de faire correspondre à deux événements rencontrés dans le discours d'un expert deux sous-classes de la classe de base Event : il suffit de créer deux instances de Event et de les différencier sur l'effet, c'est-à-dire les couples directive/processus. On pourra cependant préférer de créer une sous-classe pour chaque processus, comme si la nature d'un événement n'était pas ce qu'il est mais ce qu'il fait. On code alors dans le constructeur l'attachement du processus en question. Cette vue est admissible même si on ne crée jamais qu'une seule instance de la sous-classe.

Le modélisateur spécifie l'occurrence d'un événement au moyen de deux ensembles d'attributs : deux attributs descriptifs (la date d'occurrence et la priorité) et des attributs procéduraux (qui codent la capacité d'un événement à en reprogrammer un autre en un instant futur).

### 4.3.1 L'occurrence

La date d'occurrence est simplement une valeur de l'horloge (bien que l'utilisateur puisse l'exprimer en date calendaire puis utiliser les services de changement d'unité de temps). DIESE offre des services pour rendre cette valeur aléatoire, (uniformément, par défaut) entre deux bornes déclarées. La priorité sert à ordonnancer dans l'agenda deux instances d'événement qui ont la même date d'occurrence. Les priorités doivent être déterminées après analyse des interactions entre les processus en jeu. Elles doivent assurer que tous les processus ont accès à un état du système convenablement mis à jour pour leurs propres besoins. Dans l'exemple développé en section 4.2.2, si le processus p2 exploite, sans le modifier, l'état d'un composant A mis à jour par p1, alors p1 doit avoir une priorité plus forte que p2 (notamment pour l'instant 10, pour que l'état de A capté par p2 soit celui mis en jour en 10 et non en 5). Si p2 touchait aussi A, le modélisateur devrait décider si une obsolescence de 4 unités d'horloge du point de vue de p1 (dernière mise à jour par p2 en 6) est plus ou moins grave qu'une obsolescence de 5 unités d'horloge du point de vue de p2 (dernière mise à jour par p1 en 5). Si les deux obsolescences sont inadmissibles, c'est que le pas d'au moins un des deux processus est trop grand.

### 4.3.2 La génération automatique

Dans certaines applications, un événement d'une certaine nature peut se répéter, de manière régulière dans le temps ou non. Si la série d'occurrence est connue dès la modélisation du système (en général ou pour un système particulier), on peut déclarer chacun des événements et les ranger dans l'agenda avant le démarrage de la simulation. Non seulement

[6] On se place aussi dans le cas d'une machine séquentielle (et non parallèle), ce qui est le cas général en simulation de systèmes d'intérêt agronomique.

parce que cette opération peut être laborieuse (par exemple, installer 365 levers du soleil pour simuler une année de fonctionnement), mais aussi parce que, dans certains cas, la série ne peut pas être explicitée *a priori* mais seulement spécifiée de manière fonctionnelle, DIESE permet de ne coder que l'occurrence du premier événement de la série et prend en charge la génération des suivants. Ainsi, la génération des événements dépend des conditions courantes, appréciées dynamiquement en cours de simulation. Les connaissances nécessaires à cela sont exprimées par le modélisateur dans des fonctions qu'il écrit lui-même et qu'on qualifie d'autogénération. Leur écriture et leur attachement aux événements sont simplifiés par les services fournis par DIESE. Par exemple, pour régénérer un lever de soleil, il suffit de programmer dans la fonction d'autogénération une copie de l'événement en cours et d'incrémenter la date d'occurrence ; DIESE invoque lui-même la fonction d'autogénération, copie le processus en jeu, l'attache à l'événement copié et place l'événement dans l'agenda. Dans ce schéma, on ne peut générer qu'un événement du même effet que le précédent. DIESE propose et gère un autre schéma analogue, c'est la notion de post-conséquence d'un événement, permettant de générer un événement d'une autre nature. Par exemple, un lever du soleil peut générer un événement qui déclenche une alarme.

## *4.4 Les classes de descripteurs et les démons*

Un des traits marquants de DIESE est que les attributs descriptifs des entités du système sont eux-mêmes des objets sur lesquels on a des connaissances. C'est-à-dire qu'on décrit un descripteur indépendamment de tout attachement ultérieur à une ou plusieurs classes d'entités. Les connaissances sur un descripteur sont les suivantes :

- Sa nature constante ou variable. La différence n'est pas qu'on ne puisse jamais changer la valeur d'un descripteur constant, mais plutôt qu'on ne s'attend pas à ce qu'elle change. C'est pourquoi le système de démon (voir plus bas dans cette section) ne concerne que les descripteurs variables. Il serait en effet inutilement coûteux de chercher à savoir si une réaction automatique à un changement est attaché à un descripteur dont la valeur n'est pas censée changer.
- Le type de sa valeur. DIESE autorise qu'une valeur soit, pour ne citer que les types les plus courants, un nombre, une chaîne de caractères, la référence à une entité ou à une liste d'entités, ou encore un intervalle numérique. A chaque type, combiné avec le caractère constant ou variable, est associée une classe prédéfinie[7] dans DIESE. Par exemple, un descripteur 'numéro d'ordre' (utilisable dans toute classe d'entités) sera une sous-classe des descripteurs constants à valeur entière (IntConstantDescriptor).
- Son domaine de valeurs admissibles. Pour les descripteurs à valeurs numériques, c'est un ensemble d'intervalles. Pour ceux dont la valeur est une entité, c'est un ensemble explicite d'entités ou la référence à une classe, etc. Ne pas confondre la valeur d'un descripteur dont le type est un intervalle et le domaine de valeurs (un intervalle) d'un descripteur numérique scalaire.
- Sa valeur courante, bien entendu. Elle peut être établie par des messages (du type SetValue) adressés aux descripteurs ou bien par des messages adressés aux instances d'entités décrites, sous réserve d'indiquer le descripteur visé. Il en est de même pour l'accès à la valeur (messages du type GetValue). Normalement, cet attribut reçoit une valeur non au niveau de la classe, mais spécifiquement pour chacune des instances créées.
- Sa valeur par défaut enfin, celle qui devient automatiquement la première valeur courante si on ne la précise pas.

Une fois déclarée la classe de descripteurs, on peut en créer des instances et les placer dans une des deux listes de descripteurs qui sont attributs des entités. Puisque la valeur d'un descripteur peut (évidemment) différer d'une instance à l'autre de la même classe d'entités, ces deux entités portent deux instances distinctes de la même classe de descripteurs.

Un autre type de connaissance qu'on peut avoir sur un descripteur est que certaines de ses valeurs ont un rôle particulier : celui de déclencher quelque chose quand elles sont atteintes. Typiquement, c'est la notion d'alarme : lorsqu'une valeur limite est atteinte il faut exécuter une procédure appropriée. Ce système de surveillance automatique (appelé démon) évite au modélisateur d'ajouter des lignes de test de l'atteinte de cette limite et de lancement de la procédure après chaque instruction d'affectation de valeur. Un démon ne peut être attaché qu'aux descripteurs variables. Une fois attaché, on peut le désactiver puis l'activer à nouveau, au besoin.

Par extension, on peut associer à un descripteur un mécanisme automatique déclenché quand on accède à la valeur. Cela peut typiquement servir à lancer une procédure de calcul dynamique de cette valeur, ou à changer d'unité, etc.

Disons enfin ici qu'un mécanisme du même type peut être attaché aux entités : on peut spécifier une réaction qui sera automatiquement déclenchée lors de l'ajout ou du retrait d'un élément à une entité de caractère ensembliste (SetEntity).

[7] On qualifie de prédéfini un objet, ou un attribut d'un objet, qui fait partie de la bibliothèque DIESE. Prédéfini signifie défini avant de s'intéresser à un domaine d'application particulier.

## 4.5 Les classes de méthodes et d'arguments

Reprenant la même idée qu'au sujet des attributs descriptifs, on donne le statut d'objet aux attributs procéduraux, au lieu d'en faire de simples fonctions membres des classes d'entités de processus ou d'événements, telles que le sont les constructeurs ou les services prédéfinis par DIESE (accès à la structure des entités, aux valeur des descripteurs, préconditions des processus, fonctions d'autogénération des événements, etc.). L'intérêt est multiple.

En premier lieu, une procédure est un objet sur lequel on a des connaissances qu'il est naturel d' encapsuler dans une classe d'objets. La connaissance fonctionnelle est ainsi séparée de l'entité (ou processus ou événement) sur laquelle elle porte, parce qu'implantée dans un objet Method, lequel n'est que référencé par l'entité ; cela augmente la déclarativité et la granularité de la représentation, et autorise l'attachement dynamique et temporaire de connaissances fonctionnelles à une entité. Cela donne aussi la possibilité de distinguer deux instances de la même sous-classe d'Entity (ou de Process ou d'Event) par deux corps distincts de la même méthode. Les attributs de la classe Method sont les suivants :

- Essentiellement, un corps, qui est le morceau de programme qui effectue le traitement qui définit la méthode.
- Un type, qui est celui de la valeur éventuellement renvoyée par le corps de la méthode.
- La liste des symboles de classe des arguments : les arguments de la méthode sont des valeurs, évaluées dynamiquement au moment de son invocation, qui sont exploitées par le corps de la méthode pour adapter le traitement à ces valeurs courantes.
- La liste des arguments eux-mêmes (voir plus loin).
- Enfin l'objet sur lequel la méthode est une connaissance procédurale, c'est-à-dire une instance d'Entity, de Process ou d'Event.

En second lieu, la structure des entités du domaine devient unifiée : une série de fonctions membres est remplacée dans tous les cas par un unique attribut de type 'liste de fonctions' (la même remarque peut être faite sur l'unification de la représentation des variables d'état et des constantes, sous la forme de listes d'instances de descripteurs). C'est pour cela qu'il a été possible d'inclure dans DIESE des services d'applicabilité universelle. Pour pouvoir élargir la gamme de ces services, deux contraintes sont imposées par DIESE :

- le type des valeurs renvoyées est dans une liste exhaustive (répondant *a priori* à tous les besoins identifiés, et de toutes façons facilement extensible) ;
- quelle que soit la classe, le corps de la fonction, qui en est un des attributs, a un et un seul paramètre formel[8] : justement l'instance de Method dont le corps est attribut. Il est alors aisé d'accéder dans le corps (grâce, là encore, à des services prédéfinis) aux arguments explicites (les éléments de l'attribut 'liste d'arguments') et à ce qui constitue un argument complémentaire implicite : l'objet (et par là même son état) sur lequel la méthode est une connaissance procédurale.

On verra que ces deux contraintes sont exploitées positivement dans les outils d'interface présentés en section 5.

Les objets qui sont les valeurs de l'attribut 'liste des arguments' sont des instances de sous-classes, propres à l'application, de la classe de base Argument. Ce sont des structures proches des descripteurs. C'est dire que leur attribut essentiel est la valeur, qu'on peut établir et récupérer soit dans le corps de la méthode en adressant un message à l'instance de méthode (valeur de l'unique paramètre explicite) soit dans n'importe quelle procédure en adressant un message à une entité (ou un processus ou un événement) en ayant soin de passer la référence à la méthode de l'entité qui possède l'argument en question.

Noter que DIESE exploite la classe Method pour ses propres besoins. Ainsi, les fonctions de précondition et d'exécution des processus, de post-conséquence et d'autogénération des événements, les procédures définissant les démons, font l'objet de sous-classes prédéfinies de Method, auxquelles sont attachées des listes d'arguments eux aussi prédéfinis. En conséquence, le modélisateur n'a qu'à coder le cœur de la procédure, dans laquelle il utilise de manière relativement mécanique les services d'accès aux arguments.

## 4.6 La classe des spécifications d'ensembles d'entités

On rappelle que, la spécification fonctionnelle d'un ensemble d'entités permet de définir un ensemble d'entités sans les expliciter nommément (section 2.2.3). La seule contrainte est que ces entités doivent être instance d'une classe, appelée la classe de référence, dont le symbole est un des attributs de la classe de base EntitySpec. Cette contrainte est faible, parce qu'il est possible que cette classe soit la classe Entity. On utilise généralement la notion de spécification d'ensemble d'entités à deux fins :

[8] Ne pas confondre les paramètres du corps de la méthode (il n'y en a d'ailleurs qu'un) et les arguments de la méthode. Le paramètre est dans la signature du corps. Les arguments (et le corps) sont des attributs de la méthode.

- Celle de sélectionner un sous-ensemble parmi les instances déjà créées de la classe de référence. A cette fin, la classe EntitySpec est dotée d'une méthode prédéfinie construction (le prédicat de sélection) dont le seul argument prédéfini est une entité candidate à la sélection. Le corps de la méthode, écrit par le modélisateur, doit renvoyer une valeur booléenne décidant si l'entité correspond ou non à la définition de l'ensemble. Il convient d'appliquer la méthode à toutes les instances de la classe. C'est bien entendu DIESE qui s'en charge lorsqu'on adresse à la spécification un message dit d'expansion.
- Celle de créer un ensemble de nouvelles instances de la classe de référence. Dans ce cas, c'est le corps d'une autre méthode prédéfinie (la fonction de construction) qui fait le travail d'instanciation. DIESE fournit une version standard de ce code simple (appel au constructeur), qui peut être complétée ou surchargée par le modélisateur pour répondre à des besoins spécifiques.

Dans les deux cas, on peut préciser la taille de l'ensemble à construire.

Les spécifications de domaines de valeurs sont un outil complémentaire au prédicat de sélection et à la fonction de construction pour spécifier un ensemble d'entités. Lorsque le modélisateur d'une classe l'a dotée de spécifications de domaines de valeurs, la procédure d'expansion regarde d'abord si une entité candidate satisfait ces dernières puis invoque le prédicat de sélection (dans ce cas celui sera fréquemment une forme prédéfinie renvoyant toujours vrai). Dans le cas de création d'instances nouvelles, les spécifications de domaines de valeurs permettent de définir les valeurs initiales des descripteurs de l'entité.

## *4.7 La classe des spécifications de domaines de valeurs*

Une spécification de domaine de valeur est l'expression concise d'un ensemble de valeurs, c'est-à-dire sans en expliciter tous les éléments *a priori*. Cette classe fournit des services pour deux objectifs :

- donner un domaine de valeurs admissibles à un descripteur constant ou variable, protégeant l'utilisateur d'une affectation de valeur ne satisfaisant pas la sémantique du descripteur ;
- établir un ensemble d'entités répondant à un critère, dynamiquement en cours de simulation, c'est-à-dire sur la base des valeurs courantes des descripteurs. La présente classe est alors à exploiter conjointement avec la classe EntitySpec (voir ci-dessus). La spécification de domaine de valeur doit alors être dotée d'une référence à une classe de descripteurs. Une entité satisfait la spécification si la valeur courante du descripteur de la classe spécifiée est dans l'ensemble spécifié.

Cette classe possède deux attributs : le symbole de classe des descripteurs vis-à-vis desquels la spécification aura un sens, et un domaine de valeurs, sous-ensemble des valeurs possibles pour cette classe de descripteurs.

## *4.8 La classe des fichiers de données séquentielles*

Un système évolue dans un environnement, qui le contrôle et dans lequel il capte de la matière ou de l'énergie (on parle d'input[9]). Le captage est actif quand c'est le système qui décide lui-même la nature de l'input, la quantité et le moment. Il est alors réalisé par les fonctions internes attachées aux composants du système. Le captage est passif quand l'introduction est subie par le système. Il est alors réalisé par l'occurrence d'un événement d'origine externe (donc inséré dans l'agenda par le modélisateur). L'information sur l'input est nécessairement extérieure au système (ni déclarée en lui, ni générée par lui), puisque seulement déterminée par le fonctionnement de l'environnement. Elle peut être générée de manière dynamique par l'événement d'origine externe (captage passif), ou entièrement déclarée a priori dans une structure statique (captages actif et passif). Entièrement implique que la structure contient autant d'éléments que de moments d'introduction de l'input. Une telle structure est classiquement appelée un fichier, et les éléments sont appelés fiches ou enregistrements. DIESE propose un modèle de cette structure statique : la classe des fichiers séquentiels. Les attributs de cette classe sont les suivants.

- L'adresse en mémoire où est placée le fichier.
- La sémantique des informations captées dans un enregistrement. L'enregistrement contient les informations sur tous les inputs que le système peut éventuellement capter. On indique ici une liste de symboles rappelant la nature des seuls inputs effectivement captés. L'enregistrement peut aussi contenir des informations d'indexation, notamment la valeur du moment. Chaque item d'information est appelé un champ de l'enregistrement.
- La manière d'interpréter la structure d'un enregistrement. Pour chaque champ, on indique l'emplacement exact de l'information et son format individuel. Cela constitue ce qu'on appelle le format de l'enregistrement.

[9] Certains systèmes captent aussi des inputs de nature informationnelle.

Pour la simulation de systèmes d'intérêt agronomique, il est fait souvent usage de fichiers séquentiels sur les inputs de nature climatique[10]. Les inputs sont des quantités de pluie, l'énergie solaire, celle du vent (par sa vitesse), etc., enregistrés selon une périodicité constante (généralement l'heure ou le jour) entre deux dates. Les fichiers sont produits indépendamment de leur usage. Il est donc souvent nécessaire de ne capter qu'une partie du contenu des enregistrements. D'autre part, leur format et parfois leur emplacement s'imposent au modélisateur, et peuvent varier d'une simulation à l'autre.

Le modèle d'un système comprend donc une classe de fichier séquentiel (SequentialFile) pour chaque fichier d'inputs. On caractérise la classe par les champs d'enregistrement que le simulateur doit capter dans le fichier. Puis on code dans le module principal la création d'une instance à laquelle on attribue des valeurs d'emplacement du fichier et de format de lecture des champs. Ces informations sont normalement déclarées dans un fichier de paramètres placé en entrée du simulateur : on peut donc changer de format d'enregistrement sans modifier le code du simulateur.

[10] Les valeurs des inputs climatiques sont aussi parfois générées automatiquement à partir d'une spécification (une loi de probabilité et ses paramètres). La spécification est déterminée de manière experte ou par l'analyse des données climatiques historiques dans le contexte d'étude du système (données d'apprentissage).

# 5 Outils d'interface

On l'a vu en section 2.3, le modélisateur utilisant DIESE est censé disposer de deux et seulement deux outils : un éditeur de textes et un compilateur du langage C++. Avec l'éditeur, il écrit le code des classes de l'application ; avec le compilateur, il génère le simulateur exécutable. L'utilisateur du simulateur est censé disposer, outre du simulateur, seulement d'un éditeur de textes, avec lequel il prépare les entrées et inspecte les sorties. Dans cette situation, les acteurs manipulent les connaissances et les données sous leur forme « terminale », la plus proche possible du processeur de calcul. En fait, la bibliothèque DIESE est accompagnée de quelques outils (développés dans le langage Java™) qui proposent aux acteurs de manipuler des formes plus proches de leur propre acception, et tendent ainsi à faciliter son adoption. Ceci étant, rien ne dispense le modélisateur d'une connaissance pratique de la programmation et d'une connaissance profonde de son domaine, et la personne en charge des simulations d'une maîtrise totale du sens des entrées et des sorties du simulateur.

## *5.1 Pour le modélisateur*

### 5.1.1 Solfege

C'est une interface qui aide le modélisateur à créer les classes propres à l'application et à construire le simulateur.

Solfege propose un menu des classes de base prédéfinies. Pour chaque classe de base, une ou plusieurs fenêtres de dialogue permettent de créer les classes propres à l'application (on les appelle des éditeurs de classe, figure 9). Le modélisateur déclare les attributs spécifiques de la classe et, pour les attributs prédéfinis, peut indiquer une valeur caractéristique de la classe. Pour une entité, on détermine par exemple le symbole de classe de ses éléments en choisissant un item dans la liste des symboles des classes déjà créées (affichée dynamiquement par Solfege). De la même manière, on ajoute un descripteur (préalablement créé, même incomplètement, à l'aide de la fenêtre de dialogue pour cette classe de base), ou une méthode. A ce stade, Solfege prend en charge une partie de la vérification de cohérence du modèle.

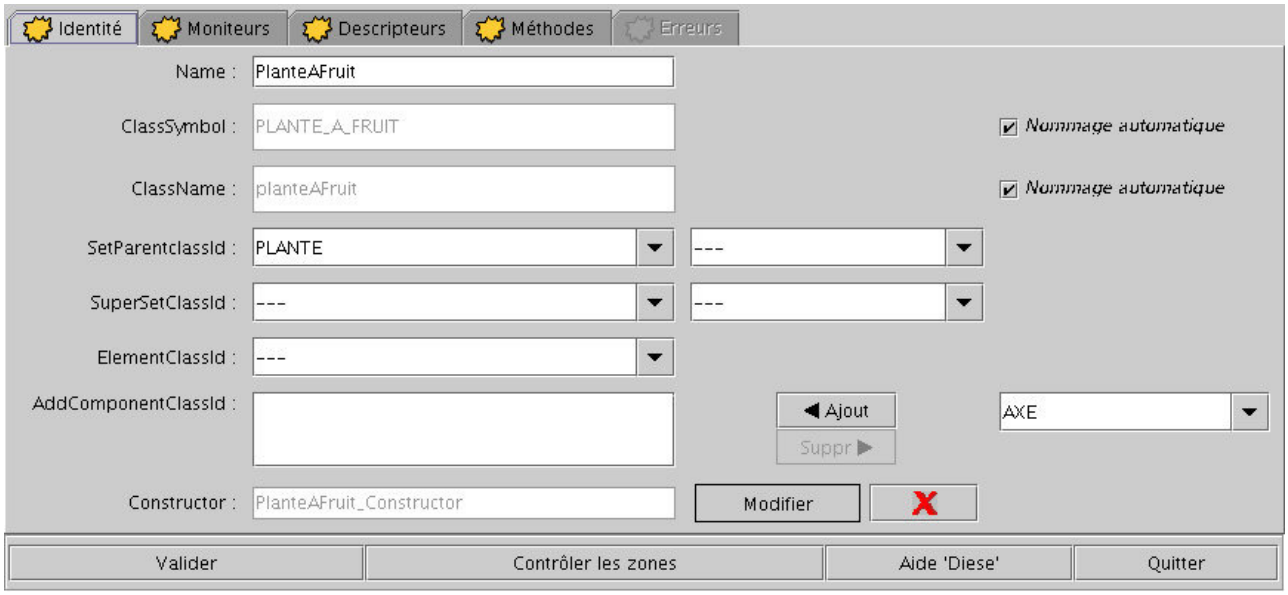



Figure 9 : la fenêtre d'édition de la classe des entités, lors de la déclaration des plantes à fruits.

Pour la construction du simulateur, Solfege génère entièrement les déclarations et le code des constructeurs des classes (voir l'exemple de la figure 3) à partir des informations écrites dans les fenêtres de dialogue. Solfege facilite le codage des corps des méthodes (les prédéfinies et les applicatives, telle celle de la figure 4) en ouvrant un éditeur de texte incluant d'entrée toute la partie non spécifique à l'application (signature, accès à l'objet porteur de la méthode, renvoi éventuel de la valeur). Le codage des connaissances spécifiques (par exemple, les fonctions de transition d'état des processus) reste naturellement à la charge du programmeur. Une fonction particulière est le module principal, auquel s'applique les remarques précédentes. On verra dans la section suivante que l'utilisation de fichiers externes pour déclarer les différentes informations nécessaires à la simulation permet de réduire le module principal à une série courte et universelle d'instructions, dont la génération est alors entièrement prise en charge par Solfege. La fonction de compilation génère le simulateur exécutable sans intervention du modélisateur. Elle a été rendue générale en fermant la liste des fichiers source de l'application. La compilation assure une autre part de la vérification de cohérence du modèle.

L'interface génère aussi une version de base de certains fichiers d'entrée (à l'exception évidente des fichiers de données séquentielles), dont le contenu est de la responsabilité de l'utilisateur du simulateur. Le fichier des paramètres du système est généré après recherche, dans l'ensemble du code, de toutes les occurrences d'un accès à la valeur d'un paramètre. Une valeur par défaut est suggérée, qui sera normalement toujours modifiée par l'utilisateur. Un second fichier de paramètres porte non pas sur le système, mais sur la simulation elle-même, avec pour exemples notables les dates de début et de fin de la période étudiée et les informations sur les fichiers de données séquentielles. Le fichier des paramètres de la simulation est généré avec autant de lignes qu'une simulation quelconque peut en requérir, avec suggestion d'une valeur servant surtout à en rappeler le format.

Solfege ne propose pas d'outil sophistiqué de gestion des versions du simulateur. La documentation du code peut être générée automatiquement à l'aide des outils du domaine public.

## *5.2 Pour l'utilisateur des simulateurs*

L'utilisation d'un simulateur, c'est la répétition d'un cycle démarrant par l'exécution d'une ou plusieurs simulations et se terminant par l'analyse des informations en sortie (trace de l'occurrence des événements, trace de l'état du système au cours de la période étudiée, et notamment l'état final). DIESE ne fournit aucun service d'analyse. Quant à l'exécution d'une simulation, elle comporte deux phases.

La première phase incombe à l'utilisateur, c'est la préparation de la commande à adresser au système d'exploitation, notamment la préparation des fichiers dont les noms en constituent les paramètres. Pour les fichiers de paramètres, la préparation, débutée de manière générale avec Solfege, peut être finalisée avec l'outil MI_DIESE présenté ci-après. Pour tous, des langages de déclaration des informations ont été élaborés, et les interpréteurs correspondants ont été programmés et inclus dans le paquet des services de DIESE. Un interpréteur lit le flot de caractères en entrée et sait reconnaître les structures lexicales et syntaxiques qui définissent le langage. Pour chaque fichier, il suffit d'inclure dans le module principal un appel à son interpréteur. Cette pratique est plus simple, plus souple et plus conviviale que celle qui consiste à programmer la lecture itérative des enregistrements d'un fichier. Plus simple, parce que l'appel à l'interpréteur est toujours identique, quelle que soit l'application et quel que soit le contenu du fichier. Plus souple, parce qu'on peut changer étendre ou modifier le langage de déclaration sans devoir modifier le code du simulateur (il suffit de modifier l'interpréteur), et parce que moins de contraintes pèsent sur l'ordre des déclarations. Plus conviviale parce que la structure des informations déclarées est plus proche de la manière de penser de l'utilisateur. Ces langages sont brièvement présentés dans le paragraphe ci-dessous. Les interpréteurs sont générés par des outils du domaine public, à partir d'une spécification du langage par son lexique (les termes autorisés) et par sa grammaire (les phrases correctement formées).

La seconde phase, c'est l'envoi de la commande de simulation et son exécution par le système d'exploitation. Le paragraphe 5.2.2 évoque l'aide apportée par MI_DIESE dans ce travail. Disons ici que l'option est prise dans MI_DIESE de toujours placer en paramètres de la commande (`main`) non pas seulement les fichiers de paramètres du système et de spécification de sorties (*cf.* §2.3.3) mais aussi le fichier des paramètres de la simulation, celui qui décrit la structure du système et enfin un fichier de directives de simulation. Ainsi, MI_DIESE génère et envoie une commande de la forme :

```
main paramètres_système paramètres_simulation spécifications_sortie
     structure_système directives_simulation
```

où le module principal du main est dans la forme universelle que sait générer Solfege (*cf.* §5.1.1).

### 5.2.1 Des langages pour déclarer les données en entrée

#### *5.2.1.1 La structure du système*

Ce langage sert d'abord à déclarer la structure et l'état initial du système simulé. Ses éléments lexicaux sont puisés dans DIESE et dans la conception des classes applicatives. Par exemple, pour déclarer un peuplement de 10 plantes numérotées, on écrira :

```
+ ENTITE  peuplement ppt1,
      POUR i1 = (1) A (10)
            + ELEMENT   plante plante{i1},
                        numero =: {i1};
                  plm2 = <p>< Peuplement  PLM2 > ;
            ;
      FIN POUR
;
```

Les mots `ENTITE` et `ELEMENT` font référence aux concepts de DIESE. Lorsque l'interpréteur rencontre le mot « + », il sait qu'il doit faire appel au constructeur de la classe dont le nom suit le mot `ENTITE` ou le mot `ELEMENT` : c'est la grammaire du langage qui le spécifie. Le mot suivant est le nom qui sera donné à l'instance. Grâce à la structure de la phrase dans laquelle il est inclus, le mot `numero` est compris comme le nom d'un descripteur de la classe applicative

dont le nom est `plante`. Pour chaque plante, nommée `plante_1`, puis `plante_2`, etc., l'interpréteur lui donnera pour valeur la valeur de la variable `i1` qui contrôle l'itération.

Ce langage sert aussi à déclarer, avec les mêmes principes, les processus en jeu et les événements à placer dans l'agenda.

#### *5.2.1.2 Les paramètres du système*

Chaque déclaration d'un paramètre fait l'objet d'une phrase telle que la suivante :

```
Peuplement     PLM2      <- 2.2 "nb/m2";
```

Les deux premiers mots sont les identifiants du paramètre. Le dernier (ici un nombre flottant) est la valeur qui lui sera donnée. Bien que la valeur de ce paramètre serve à initialiser un descripteur du peuplement déclaré dans la structure, les deux mots identifiant sont libres. Il suffit que la documentation du simulateur indique la signification correcte du paramètre, et que le rédacteur du fichier de la structure réalise les affectations correctes de valeurs. C'est là que le choix des mots a son importance ergonomique. Un commentaire libre peut être écrit après la valeur.

L'identification par deux termes provient de l'utilisation la plus fréquente des paramètres du système : déterminer la structure et l'état des entités. Une bonne recette est alors que le premier mot rappelle la classe d'entités et que le second rappelle un descripteur ou une relation. On comprend que ce fichier doive être interprété avant celui de la structure du système.

#### *5.2.1.3 Les paramètres de la simulation*

L'interprétation de ce fichier provoque la nécessaire création de l'instance de la classe Simulation (cf. §2.3.2.3.2). Un nombre fini de phrases permettent de préciser, par exemple, la période de simulation et l'unité de temps simulé. Par ailleurs, c'est là qu'on identifie les fichiers de données séquentielles. Voici une version de ce fichier pour TOMMY :

```
INITIALISATION SIMULATION
       UNITE_TPS MINUTE;          NB_UNITE_TPS 15;
     DATE_DEBUT 01 OCT 1994;      DATE_FIN 31 JUL 1995;
          OUT_DIR  "../out/";
;
OUVERTURE FICHIER_SEQUENTIEL FICHIER_METEO
     NOM_PHYSIQUE  "../in/meteo_alenya.txt";
     FORMAT_CHAMPS "%d %d %d %d %lf %lf %lf %lf %lf";
;
OUVERTURE FICHIER_SEQUENTIEL FICHIER_CONSIGNES
     NOM_PHYSIQUE  "../in/consigne.txt";
     FORMAT_CHAMPS "%d %d %d %d %lf %lf %lf %lf %lf %lf";
;
```

#### *5.2.1.4 Les spécifications de sortie*

Les phrases admissibles commencent par un verbe qui indique la requête, et se continuent par des termes qui précisent le « quoi » et le « comment ». Pour les entités, les requêtes possibles sont de sauvegarder en mémoire ou sur fichier, ou de visualiser dynamiquement à l'écran, les valeurs de certains descripteurs pour certaines classes. On écrira par exemple la phrase suivante pour sauvegarder dans le fichier "`../out/f_Axe.txt`" préalablement vidé, tous les changements de valeur de l'âge et du nombre de sympodes (sans décimales) sur toutes les instances de la classe Axe. La date enregistrée sera la valeur de l'horloge.

```
SAVE DESCRIPTOR "axe" ALL "f_Axe.txt" NEW CLOCK "ag" 0 "nbSympodes"
0;
```

Pour les événements et les processus, les requêtes portent sur la chronique des occurrences et des mises en œuvre.

#### *5.2.1.5 Le programme des actions de simulation*

Voici le fichier des directives de simulation de TOMMY, légèrement simplifié sur des points annexes. Il a été presque entièrement généré par Solfege, aux détails près du format de l'affichage de la durée de simulation et de la classe sur laquelle porte la directive de sortie sur les entités. C'est d'ailleurs à ce niveau qu'on peut restreindre et changer d'une simulation à l'autre le contenu des sorties, sans changer le fichier des spécifications de sortie, pour peu qu'on y ait déclaré toutes les requêtes appropriées. Les directives de nettoyage de la mémoire (`DELETE`) sont optionnelles, mais utiles lorsqu'on enchaîne une série de simulations avec MI_DIESE. Ce fichier est le seul pour lequel on doit porter attention à l'ordre dans lequel on fait les déclarations.

```
CHRONO INIT;
CHRONO START;
RUN;
CHRONO PAUSE;
CHRONO DISPLAY ENDL "duree de la simulation = ";
CHRONO CLOSE;
DISPLAY OUTPUTSPEC EVENT;
DISPLAY OUTPUTSPEC ENTITE axe;
DELETE ENTITE INSTANCE;
DELETE PARAMETRE;
DELETE OUTPUTSPEC;
DELETE ENTITYSPEC;
DELETE DESCVALUESPEC;
DELETE MONITOR;
DELETE FILE;
DELETE SIMULATION;
DELETE ENTITE CLASSE;
```

### 5.2.2 MI_DIESE :

C'est une interface graphique, alternative du lancement par l'utilisateur lui-même d'une ligne de commande au système d'exploitation (*cf.* §2.2.3.2) avec en paramètres des fichiers préparés avec un simple éditeur de texte. C'est par une option de la compilation du simulateur qu'on décide si celui-ci pourra être utilisé sous l'interface graphique ou non. Alors qu'il est de pensée courante que l'interface d'un logiciel doit satisfaire un cahier des charges spécifique élaboré avec les utilisateurs *in situ*, cette interface est d'applicabilité générale (c'est-à-dire peut manipuler tout simulateur créé dans le cadre de DIESE). L'idée d'une interface utilisateur générale a déjà été portée par GUICS (Acock et al., 1999), mais restreinte au cadre des simulations de culture. La généralité de MI_DIESE est celle de DIESE, car ses fonctionnalités exploitent exclusivement les structures de données connues de DIESE et les services qu'elle fournit. A titre d'exemple, lorsque l'interface graphique veut visualiser la valeur d'un descripteur d'une entité, elle adresse vers le simulateur un message de type GetValue (*cf.* §4.4). C'est le dialogue entre l'interface et l'utilisateur qui détermine l'entité destinataire du message et le symbole du descripteur visé. Les principales fonctionnalités sont les suivantes. On n'indique pas ici toutes les solutions techniques qui les autorisent.

- Etablissement/modification des entrées : MI_DIESE visualise les fichiers préparés par Solfege ou écrits par l'utilisateur comme l'aurait fait Solfege. On peut gérer une bibliothèque de fichiers et y puiser ceux valides pour la simulation en préparation.
- Lancement de la simulation : l'utilisateur peut se permettre ou ne pas se permettre d'interagir avec le simulateur pendant son exécution. Dans le premier cas seulement, il peut définir les moments où l'exécution du simulateur sera interrompue et mise en attente d'une requête en provenance de l'utilisateur *via* l'interface. Ces requêtes concernent notamment la visualisation de l'état courant du système. Une requête particulière est celle qui fait redémarrer l'exécution interrompue.
- Monitoring de l'évolution de variables : on peut énoncer, avant le démarrage d'une exécution, des requêtes de monitoring, c'est-à-dire de visualisation graphique de l'évolution de certaines variables d'état (figure 10). Les graphiques portent exclusivement le temps en abscisse (c'est-à-dire qu'on ne peut pas visualiser l'évolution d'une variable en fonction d'une autre, chaque nouveau point correspondant à l'instant, non précisé, de changement de l'une ou de l'autre). On peut aussi demander la trace de l'occurrence des événements, ou encore l'actualisation du graphique de la structure du système (ajout/retrait d'éléments, etc.).
- Inspection des résultats en fin de simulation : on peut, une fois la simulation terminée, avoir une vue statique du dernier état pris par le système. Cette fonctionnalité et les deux précédentes reposent sur une implantation originale de communication par « pipe line » entre les processus Java et C++. On peut aussi accéder, par des fonctions simples d'édition de texte, aux fichiers générés à partir des spécifications de sortie. C'est d'ailleurs ces fichiers qu'il faudrait exploiter pour développer des fonctionnalités de visualisation spécifiques au domaine d'application.
- Enchaînement automatique de simulations sur une série de jeux de données : dans certaines études, il est utile de lancer une série de simulations et d'en analyser globalement les résultats. La série peut être définie par un paramétrage différent à chaque simulation. Un autre cas de figure est de lancer de nombreuses fois une simulation avec un paramétrage donné, mais sur un système comportant un composant stochastique qui provoque des comportements différents à chaque simulation. MI_DIESE permet de spécifier les éventuels changements de paramétrage entre deux simulations successives, génère lui-même les jeux de fichiers appropriés, en enchaîne les simulations sans autre intervention de l'utilisateur (et notamment sans autoriser l'interaction).

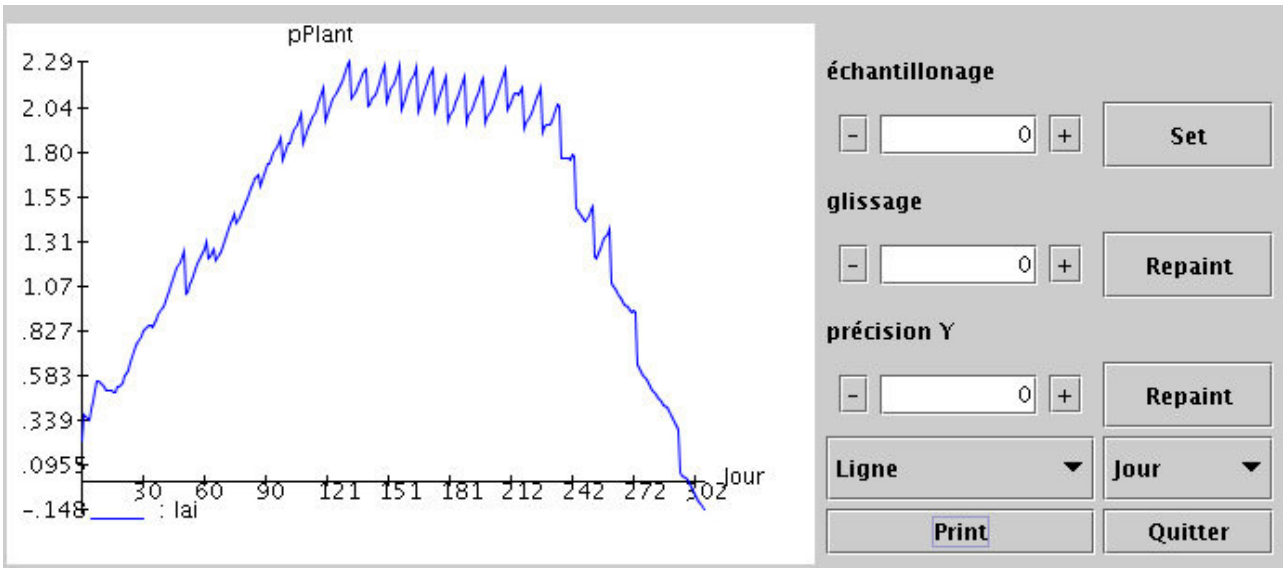



Figure 10 : la trace dynamique de l'évolution de l'indice de surface foliaire d'une plante.

Lorsqu'une valeur intervenant dans une procédure peut varier d'un contexte d'étude à une autre ou, pour un même contexte, d'une simulation à l'autre, le concepteur du simulateur en fait classiquement un paramètre, et lui prévoit une place dans un fichier placé en entrée du simulateur. L'utilisateur n'a qu'à éditer ce fichier pour indiquer la valeur appropriée. Ainsi, le simulateur n'a pas à être modifié donc compilé à nouveau. DIESE inclut, de manière encore expérimentale, une innovation technique qui étend cette possibilité aux connaissances fonctionnelles. Par exemple, on pourra laisser l'utilisateur choisir entre deux versions d'une fonction physiologique ou d'une formule de bilan économique. Il suffit de placer le code de la fonction dans un fichier particulier et d'indiquer au simulateur qu'il doit regarder dans ce fichier si une version, dite externalisée, existe et doit alors être privilégiée. Sinon, la version fournie par le développeur sera utilisée.

# 6 Résumé, discussion, perspectives et conclusion

En résumé, DIESE est un ensemble de classes d'objets qui instrumentalise un méta-modèle. On entend par là un modèle d'un niveau d'abstraction plus élevé que le modèle cible, celui du système étudié. Le méta-modèle impose les concepts sur lesquels reposeront tous les éléments du modèle cible. Il leur donne un sens et définit les relations entre eux. C'est un modèle en ce qu'il n'est qu'un point de vue, fondé sur le triptyque entité-processus-événement, sur une réalité. Mais cette réalité, à savoir les systèmes d'intérêt agronomique en général, est une abstraction des systèmes particuliers qui sont l'objet des études. L'ensemble des classes de DIESE est le composant essentiel d'un outil logiciel de développement de simulateurs. Dans la forme la plus fruste de cet outil, la bibliothèque n'est complétée que par la documentation des services offerts par les classes et le fichier universel des directives de compilation. Il revient alors au modélisateur de réutiliser les classes dans son propre programme avant de le compiler. Dans une forme plus évoluée, bien qu'encore sous la forme d'un prototype, l'outil comprend aussi l'interface qui guide le travail de modélisation et génère une partie du code du simulateur. Quant à l'utilisateur des simulateurs, une interface lui est fournie, incluant les fonctionnalités classiques, mais sous une forme commune à tous les domaines d'application[11].

Par rapport aux pratiques en cours en modélisation des systèmes d'intérêt agronomique, l'utilisation de DIESE se distingue par l'appui sur le méta-modèle. L'avantage est une définition plus nette des frontières du système, une identification rapide de ses composants, une sémantique claire des relations structurelles et fonctionnelles. Le modèle devient immédiatement intelligible par tout lecteur partageant le méta-modèle. Sur le développement informatique, l'utilisation de DIESE se distingue par la réutilisation de composants, ce qui entraîne simplification et standardisation de l'écriture. On pense d'une part aux classes d'objets (entités, descripteurs, etc.) : une simple instruction d'héritage intègre toute la connaissance sur une classe dans la définition d'une sous-classe. Les services fournis dans les classes de base sont souvent des macro-instructions pré-programmées. Ils permettent une navigation dans la structure du système à partir d'un petit nombre de fonctions standards. On pense d'autre part aux corps de méthodes : une connaissance procédurale valide pour un ensemble de situations, par exemple un modèle de circulation de l'eau dans le sol, est disponible pour toutes les études dans cet ensemble. Dans cet ordre d'idées, on rappelle la génération automatique de code offerte par l'interface Solfege et, dernière chose mais non la moindre, l'inclusion dans la bibliothèque d'un moteur de simulation, ce qui décharge le programmeur de l'écriture des instructions de contrôle du temps. Enfin, quant aux capacités d'expression, l'utilisation de DIESE se distingue par les possibilités de déclarer des processus de pas différents, d'automatiser la répétition (de manière éventuellement aléatoire) d'événements, d'installer des mécanismes de démons, de définir de manière flexible des ensembles d'entités, de spécifier la nature des sorties sans toucher au programme ni le recompiler. Noter enfin la possibilité d'écrire sur un fichier externe (placé en entrée du simulateur et interprété dynamiquement pendant l'exécution) de petites procédures exprimant une connaissance fonctionnelle qu'on ne souhaite pas compiler sous une forme unique dans le simulateur, mais au contraire faire varier d'une simulation à l'autre, à l'instar des paramètres.

Certainement, ces distinctions peuvent apparaître soit comme des contraintes soit comme des gadgets facilement programmables par les pratiques en cours. En effet, on voudrait avoir sa propre conception d'une plante à fruits, on peut programmer le domaine de valeurs admissibles d'un descripteur par un test à l'affectation, les processus en jeu n'ont pas des pas si variés qu'un système de boucles imbriquées ne suffise pas à les synchroniser, etc. Pour juger la durabilité de cette position, on peut évoquer la manière dont on conduit une étude statistique : non pas en réinventant le modèle linéaire, non pas en reprogrammant l'inversion de matrice, mais en décidant de la bonne méthode et en puisant son implémentation dans une bibliothèque de fonctions statistiques réutilisables. Il existe cependant un manque réel dans l'offre de DIESE : la déclaration des processus continus sous forme d'équations différentielles (au moyen d'un éditeur d'équations), et la fourniture, corollairement, d'une procédure d'intégration. La justification tient d'abord dans la difficulté, non affrontée dans DIESE, à synchroniser l'avancée de processus discrets et la résolution d'un système d'équations différentielles. Elle tient ensuite à la difficulté de faire référence, dans l'expression mathématique d'une équation différentielle, aux connaissances symboliques qui modélisent la structure du système. On remarque enfin que la plupart des simulateurs existants contiennent des reformulations d'équations différentielles (la forme théorique dans laquelle le modèle est publié) en équations aux différences. Leur codage en C++ n'écarte pas réellement les modélisateurs de leurs pratiques actuelles.

Dans l'hypothèse où les modélisateurs de systèmes d'intérêt agronomique devraient utiliser un environnement de simulation à événements discrets, qu'apporte DIESE au regard de ceux déjà disponibles dans les laboratoires de recherche ou sur le marché ? Les avantages sur les derniers sont, outre la gratuité[12], l'ouverture, la liberté d'extension et la partageabilité des développements, sous réserve d'en mettre en place les moyens. Quel que soit le statut public ou commercial de l'environnement, soit aucun méta_modèle de système n'est inclus (tel adevs, qui ne propose que les

[11] Les paquets logiciels peuvent être demandés à l'auteur, par courriel à rellier@toulouse.inra.fr

[12] DIESE est un produit gratuit, ouvert, libre d'utilisation et d'extension, sous réserve que les sous-produits soient gratuits et ne restreignent pas les caractères ouvert, libre d'utilisation et d'extension des modules de DIESE qu'ils réutilisent. Les sous-produits doivent mentionner cette disposition et l'origine de DIESE.

procédures de synchronisation de processus), soit le méta-modèle proposé est orienté sur un domaine particulier (souvent les réseaux de machines, tel OMNet++). Ou bien encore , lorsqu'il s'agit d'un méta-modèle général (on peut citer les projets PSL –Gruninger *et al.*, 2003-, TOVE –Fox and Gruninger, 1998, orientés sur les modèles d'entreprise), c'est l'instrument informatique qui n'est pas développé. Dans tous les cas, un lourd travail de développement serait nécessaire.

Une application de la généricité de DIESE est le développement en cours de CONTROL_DIESE, une bibliothèque de classes orientée vers la modélisation de systèmes de production, c'est-à-dire de systèmes naturels et/ou techniques gérés par un pilote humain détenteur d'un objectif de production. Les classes de CONTROL_DIESE sont exclusivement des spécialisations de celles de DIESE. Ainsi, et par exemple, le méta-modèle d'un système de production contient le concept d'activité qui hérite toutes les propriétés de la classe Entity et qui la spécialise par l'ajout d'attributs tels les dates de début et de fin. Les concepts qui sont implémentés dans CONTROL_DIESE ont été décrits sous la forme d'une ontologie (Martin-Clouaire et Rellier, 2003, 2004). La bibliothèque contient aussi, placées dans des corps de méthodes de l'entité modélisant le pilote, un modèle de la manière de conduire le système dans un environnement incomplètement connu et incertain. Le problème de l'interopérabilité trouve, dans CONTROL_DIESE, une solution simple. Le système piloté communique avec le système de pilotage dans un espace d'information qui leur est commun : celui où sont créées les instances des deux systèmes. Ainsi, le pilote a accès à l'état d'un composant du système piloté de la même manière qu'une fonction biologique d'un composant a accès à l'état d'un autre composant.

La généralité de DIESE se matérialise par sa capacité à supporter la modélisation de systèmes agronomiques variés. On a décrit le modèle d'un peuplement de tomate sous serre. Sont actuellement en développement deux autres modèles. Celui de la partie biologique et technique d'une exploitation d'élevage dans laquelle on veut étudier la gestion des effluents. La partie gestion sera modélisée dans le cadre de CONTROL_DIESE. Et d'autre part celui de l'atelier 'grandes cultures' d'une exploitation qui, pour ne parler que du domaine d'application de DIESE, soulève les questions spécifiques de la structuration spatiale et fonctionnelle de l'espace cultivé. Quant à la capacité de DIESE à modéliser des systèmes autres que ceux d'intérêt agronomique, rien ne permet de l'exclure puisqu'on peut vérifier que la présentation de ses principes et de son contenu ne s'appuie sur les systèmes agricoles que pour l'exemple. Il en est d'ailleurs de même pour la spécialisation CONTROL DIESE. Le renforcement de cette hypothèse passerait par des tentatives effectives, dans des domaines comme la production manufacturée.

Revenons sur la question de l'interopérabilité, définie comme la capacité à faire communiquer sans ambiguïté des outils logiciels, possiblement conçus avec des logiques différentes, pour réaliser une tâche déterminée. Le problème, particulier mais de plus en plus présent dans les études agronomiques, de la connexion entre le simulateur d'un système bio-technique et un module qui détermine les actions de pilotage trouve avec DIESE et son extension CONTROL_DIESE une solution technique élégante, commode et sûre. Dans les pratiques en cours, les actions de pilotage sont au pire déclarées de manière statique dans un fichier externe et, au mieux, déterminées par des morceaux de code plaqués sur celui du système bio-physique et simulant des règles de décision. Il n'y a pas d'interopérabilité entre modules différents. Un autre problème est celui de l'accès à des systèmes d'informations extérieurs et autonomes vis-à-vis des simulateurs : notamment bases de données et systèmes d'information géographiques, locales ou accessibles par réseaux de communication. Ce problème, plus général mais apparemment moins crucial, peut *a priori* avoir les mêmes solutions techniques dans la nouvelle pratique de modélisation proposée par DIESE et dans les pratiques en cours.

En conclusion, si on croit en l'intérêt pour la communauté des agronomes de créer et de partager des connaissances structurelles et fonctionnelles sur les systèmes agronomiques pour la capitalisation des connaissances, le développement rapide de modèles, pour mieux communiquer avec les autres disciplines (sciences de la décision, ergonomie, gestion de production) en jeu dans l'analyse et la résolution de problèmes complexes, DIESE peut favoriser l'émergence de cette culture.

# Références


Acock B., Pachepsky Y.A., Mironenko E.V., Whisler F.D., Reddy V.R (1999). GUICS : a generic user interface for on-farm crop simulations. Agron. J., 91, 657-665.

Attonaty J.M., Chatelin M.H., Poussin J.C. and Soler L.G. (1993). Advice and decision support systems in agriculture: new issues. In: Farm Level Information Systems. Woudschoten, Zeist, The Netherlands, May 10-14, 1993. (Eds. R.B.M. Huirne, S.B. Harsh and A.A. Dijkhuizen), 89-101.

Brisson N., Mary B., Ripoche D., Jeuffroy M.H., Ruget F., Gate P., Devienne-Barret F., Antonioletti R., Durr C., Nicoullaud B., Richard G., Beaudoin N., Recous S., Tayot X., Plenet D., Cellier P., Machet J.M., Meynard J.M., Delécolle R. (1998). STICS: a generic model for the simulation of crops and their water and nitrogen balance. I. Theory and parametrization applied to wheat and corn - Agronomie, 18: 311-346

Cros M.J., Duru M., Garcia F., Martin-Clouaire R. (2001). Simulating Rotational Grazing Management. Environment International 27(2001), 139-145.

Elmqvist H., Mattson S.E., Otter M. (2000). Object-oriented and hybrid modeling in Modelica. In: Proceedings of ADMP 2000, Dortmund, Germany.

Evans J.B. (1988). Structures of discrete event simulation: an introduction to the engagement strategy. John Wiley & Sons.

Fox, M.S., Gruninger, M. (1998). Enterprise Modelling, AI Magazine, AAAI Press, Fall 1998, 109-121.

Gauthier L., Gary Ch., Zekki H. (1999). GPSF: a generic and object-oriented framework for crop simulation. Ecological Modelling, 116, 253-268.

Gibon A., Lardon S., Rellier J.P. (1989), The heterogeneity of grassland fields as a limiting factor in the organization of forage systems. Development of a simulation tool of harvest management in the Central Pynénées. In: Etudes et Recherches sur les Systèmes Agraires et le Développement (16), A.Capillon Ed., 105-117.

Gruninger, M., Sriram, R.D., Cheng, J., Law, K. (2003). "Process Specification Language for Project Information Exchange," International Journal of IT in Architecture, Engineering & Construction, 2003.

Guerrin F. (2001). Magma: a model to help manage animal wastes at the farm level. Computer and Electronics in Agriculture, 33(1), 35-54.

Jones, C.A. and Kiniry, J. (1986). Ceres-N Maize: a simulation model of maize growth and development. Texas A&M University Press, College Station, Temple, TX.

Jones J.W., Dayan E., Allen L.H., Van Keulen H., Challa H. (1991). A dynamic tomato growth and yield model (TOMGRO). Trans. ASAE, 43(2), 663-672.

McCown R.L., Hammer G.L., Hargreaves J.N.G., Holzworth D., Freebairn D.M. (1996). Apsim: a novel software for model development, model testing and simulation in agricultural system reseach. Agric. Syst., 50, 255-271.

Martin-Clouaire R., Rellier, J-P. (2003). A conceptualization of farm management strategies. In: Proceedings of the EFITA 2003 Conference, Debrecen, Hungary, July 5-9, 2003, 719-726.

Martin-Clouaire R., Rellier, J-P. (2004). Fondements ontologiques des systèmes pilotés. Technical report INRA/UBIA, 2004. 84p. In french. Available on request to authors.

Meynard J.M. (1997). Which crop models for decision support in crop management ? Example of the DÉCIBLÉ system. Proceeding of the INRA-KCW workshop on DSS, Laon, (France), October 1997.

Ritchie, J.T. and S. Otter. (1985). Description and performance of CERES-Wheat: A user-oriented wheat yield model, USDA-ARS, ARS-38, 159-175.

Rumbaugh *et al.* (1991). Object Oriented Modeling and Design. Pentice Hall (U.K.).

Williams J.R., C.A. Jones, P.T. Dyke. (1990). The EPIC model. In: Sharpley, A.N. and J.R. Williams (Eds) EPIC-Erosion/Productivity Impact Calculator: 1 Model Documentation. USDA Technical Bulletin No: 1768 235.

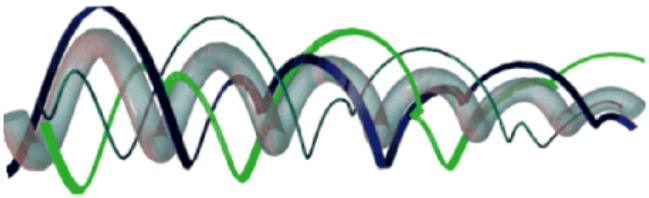
Varga A. (2001). The OMNeT++ Discrete Event Simulation System. In: Proceedings of the European Simulation Multiconference (ESM'2001). June 6-9, 2001. Prague, Czech Republic.

L'unité de Biométrie et Intelligence Artificielle de Toulouse a pour mission de développer et d'appliquer des méthodes de statistiques et d'intelligence artificielle dans le cadre des grands axes de recherche de l'INRA. L'unité travaille en particulier sur l'analyse par modélisation, simulation et optimisation de systèmes en agronomie, écologie, épidémiologie, gestion forestière, et dans le domaine de la biologie sur la localisation et l'identification d'éléments fonctionnels dans les génomes des bactéries, plantes et animaux, aux niveaux génétique, moléculaire et de l'expression de gènes.
Ces recherches s'accompagnent d'une activité de production de logiciels pour leur valorisation et d'une activité de formation pour leur diffusion.

*The Biometry and Artificial Intelligence laboratory at Toulouse is part of INRA (French National Institute for Agricultural Research). Its mission is to develop and apply statistical and artificial intelligence methodologies to the specific research domains of INRA. The laboratory develops methods for modeling, simulating and optimizing systems in agronomy, ecology, epidemiology, forest management. In biology, models and methods are developed that aim at locating and characterizing functional elements inside the genomes of bacteria, plants and animals, at the genetic, sequence and expression data levels. This also includes teaching and software development.*

INRA - UBIA, BP 52627 Auzeville F-31326 Castanet Tolosan cedex
Tél : +33(0)5.61.28.52.75 - Fax : +33(0)5.61.28.53.36 - http://mia.toulouse.inra.fr

Image de synthèse : Benjamin Auriol
Publication : Jérôme Juille (INRA - SIDAR)



INRA
Institut National de la Recherche Agronomique

Siège social : INRA 147, rue de l'Université F-75338 PARIS cedex 07
Tél : +33(0)1 42 75 90 00 - Fax : +33(0)1 47 05 99 66