Guide pour la construction d'un simulateur sous DIESE, support d'introduction à l'utilisation de cet environnement de modélisation

SiBemol : Dynamique et conduite d'une exploitation agricole simple virtuelle

Jean-Pierre Rellier (INRA/MIA/UBIA - Toulouse)
INRA - Centre de Recherche de Toulouse-Auzeville
Unité de Biométrie et Intelligence Artificielle

Décembre 2011

Ce document est un tutoriel, destiné à accompagner pas à pas le développement d'une base de connaissances dans l'environnement de modélisation/simulation DIESE, lequel est la version logicielle et opérationnelle d'une ontologie des systèmes pilotés[1]. L'environnement fournit un ensemble de classes (et les services associés) qui sont autant de modèles des objets rencontrés et manipulés dans l'étude des systèmes pilotés en général, et dans les systèmes de production agricoles en particulier, le domaine dans lequel DIESE a été conçu.

Le système pris comme objet de la modélisation est une exploitation agricole virtuelle. On a sur elle une vue très simplificatrice, avec la seule finalité de rencontrer, lors de sa modélisation, un éventail de situations représentatif de la démarche de modélisation, des difficultés rencontrées et des capacités d'expression des connaissances, notamment sur la conduite du système. L'énoncé du problème de modélisation[2] est en Annexe 4 de ce document. Il contient la description du système de production étudié (aspects structurels, ressources et pilotage) et de son environnement.

Ce développement guidé doit familiariser le lecteur-manipulateur avec les outils offerts par l'environnement pour capturer et organiser ses connaissances. Le pré-requis est donc d'avoir installé DIESE sur la machine de travail. Un guide d'installation est disponible en suivant le lien http://carlit.toulouse.inra.fr/diese/, rubrique 'Télécharger'.

Le présent document est un ensemble de consignes à appliquer, écrit à la manière d'un scénario. Elles portent sur :
- l'exécution de lignes commandes dans un terminal
- l'ouverture de fenêtres à partir de menus et de choix d'items dans ces menus
- l'ouverture de fichiers à fin d'édition, etc.

Au cours du processus de modélisation, le lecteur-manipulateur prendra soin de sauvegarder fréquemment l'état courant de la base développée. Parce que si un problème bloquant intervient à l'étape N, il est sans doute plus facile et sûr, dans une phase d'apprentissage, de ré-appliquer les consignes à partir de l'état de la base à l'étape N-1, que de tenter de réparer le problème.

Avertissements

1) Les copies d'écran illustrant les étapes ont été captées sur le site du rédacteur de cette note. Le nom de login visible ici et là rappelle cette situation. Le scénario est joué dans un environnement Linux, mais le déroulé dans un environnement Windows/Cygwin est totalement identique.

2) Ce tutoriel ne constitue pas une approche systématique de l'ontologie qui fonde le contenu de DIESE. Par exemple, lorsqu'on s'apprêtera à créer des sous-classes d'entités, on ne reprendra pas la définition ontologique du concept d'entité, ni sa place dans le réseau de concepts. Au cours du développement, le lecteur-manipulateur pourra se reporter à ce document, pour des compléments de documentation, d'explication ou de justification.

3) Les choix de modélisation que le lecteur-manipulateur est invité à adopter sont généralement justifiés par la question à laquelle la simulation est censée répondre et par la vision du système étudié nécessaire et suffisante pour cette question. Cependant, certains choix sont aussi justifiés par le rôle assigné à ce tutoriel de démontrer certaines fonctionnalités de l'environnement DIESE.

4) Le but de ce tutoriel n'étant pas d'enseigner la programmation C/C++, les corps de toutes les fonctions de la base de connaissance (qu'elles soient fonction-membre d'une classe ou fonction globale "libre") sont fournis dans un répertoire de fichiers[3] qui accompagne ce tutoriel. Les opérations de "copier/coller" à partir de ces fichiers vers l'espace de développement de la base seront décrites précisément.

5) De même, le but de ce tutoriel n'étant pas d'enseigner le langage par lequel on rédige les fichiers en entrée des simulations, le contenu de ces fichiers est fourni dans un fichier[3] qui accompagne ce tutoriel. Les opérations de "copier/coller" à partir de ce fichier vers l'espace des données seront décrites précisément.

---

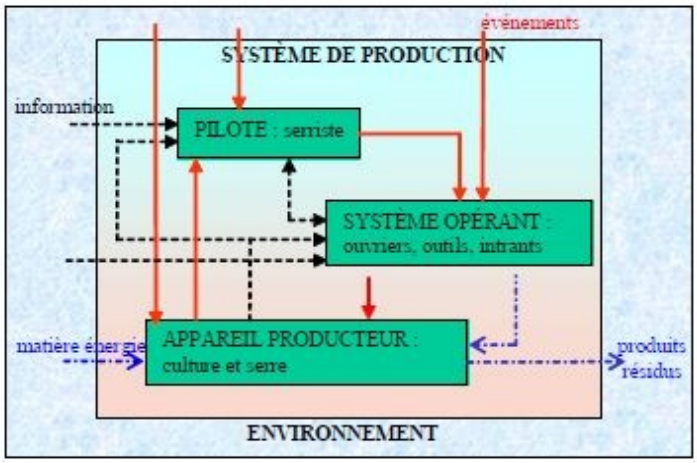
[1] Fondements ontologiques des systèmes pilotés. R. Martin-Clouaire, J.-P. Rellier. 2011. Disponible sur le lien http://carlit.toulouse.inra.fr/diese.

[2] Ce problème a été le support de l'atelier MODELISAD (groupe thématique autour de la modélisation des systèmes biotechniques au département SAD) tenu les 3 et 4 octobre 2011, dans les locaux parisiens (rue Claude-Bernard) d'AgroParisTech.

[3] Ces fichiers pourront contenir des références au système de fichiers du rédacteur de cette note. Le lecteur-manipulateur est chargé d'adapter ces références à son propre système de fichiers.

Préambule : Vue d'ensemble de la démarche de développement

La démarche de développement est liée à la **conception générale du système de production** à la base de l'ontologie. Le schéma ci-dessous, bien que se référant à un type de système, résume cette conception. Il est commenté dans quelques articles sur DIESE ou sur ses applications[4].



Une autre considération générale, structurante sinon pour le scénario de développement, du moins pour le positionnement relatif des modules développés, est qu'un système peut être analysé selon **trois points de vue complémentaires** :
- Du point de vue structurel, on regarde le système tel qu'il est. Sa structure, son aspect.
- Du point de vue fonctionnel, on regarde le système tel qu'il change. Changements de structure, changements d'aspect. Interactions entre composants internes.
- Du point de vue dynamique, on regarde ce qui fait changer le système et quand. Stimulus et commandes externes. Stimulus et commandes entre composants internes.

Enfin, la distinction doit être bien nette entre deux volets du développement :
- **le développement de la base de connaissances** : c'est l'expression des connaissances en jeu dans le domaine étudié et utiles pour le problème posé.
- **la description d'un cas d'étude**, plus ou moins fictif, pour les besoins de la vérification de la qualité du développement.

La base de connaissances est valide dans un espace de cas d'étude. Le développeur doit communiquer ses frontières à l'utilisateur de la base. La base est traitée par le moteur général de simulation de DIESE. Une simulation est l'application, par le moteur, de la base de connaissances sur un cas d'étude.

Scénario du développement

I : Les préparatifs :
- Lancement de l'interface de développement
- Identification d'un nouveau développement et installation dans le système de fichiers
- Sur une base encore vide, initiation de la boucle "génération du code, compilation, exécution, développement"

PHASE 1 : La structure du système, son environnement et les processus biophysiques autonomes

II : Premiers éléments de la base de connaissances :
- Les classes d'entités visibles au premier abord : Exploitation, Parcelle, Troupeau, etc.
- Les premiers descripteurs (aspects) des entités : surface, nombre d'animaux, etc...

III : Premiers éléments de description du cas d'étude
- Installation du cas dans le système de fichiers
- Le fichier des paramètres de la simulation : unité d'horloge, début et fin, etc.
- Le fichier de la structure du système : une exploitation particulière avec ses parcelles à elle, etc...

IV : Un stimulus externe : la croissance de l'herbe
- Examen du fichier disponible, "simulant" cette croissance par ses enregistrements successifs
- Codage de la lecture d'un enregistrement, et de la mémorisation de son contenu
- Dans l'agenda de la simulation, insertion d'événements répétés de lecture du prochain enregistrement

[4] Entre autres : "Représentation et interprétation de plans de production flexibles". R. Martin-Clouaire et J.-P. Rellier (2006) In: *Actes des Journées Francophones sur la Planification, la Décision et l'Apprentissage pour la Conduite de Systèmes (JFPDA'06)*, Toulouse, France, Mai 2006

V : Un autre stimulus externe : l'énergie pour le développement du blé
Examen du fichier disponible, "simulant" cette réception d'énergie par ses enregistrements successifs
Codage de la lecture d'un enregistrement, et de la mémorisation de son contenu
Dans l'agenda, insertion d'événements répétés de lecture du prochain enregistrement

VI : Le processus interne stimulé par l'énergie reçue : le développement du blé
Installation d'un modèle générique de l'âge du blé, repris d'une autre application voisine
Un élément du modèle : une succession de stades
Particularisation de ce modèle pour le blé : ses stades propres, et leurs conditions d'atteinte
Codage du processus de développement : "le stade suivant a-t-il ses conditions d'atteinte satisfaite ?"

PHASE 2 : Activités, Opérations, Ressources

VII : Expression des réquisitions des ressources en jeu dans la gestion du système
Identification des ressources atomiques : unités de main d'œuvre et outils
Expression des réquisitions par les opérations et les activités

VIII : Une première intégration 'Activité, opération, ressources'
Installation de l'opération 'semer un blé', avec l'expression de sa réquisition de ressources propres
Installation de l'activité 'semis d'un blé', avec l'expression de l'objet opéré et de la main d'œuvre

IX : Un embryon de plan d'activités, seulement avec le semis du blé
Installation de l'activité-séquence 'itinéraire technique du blé'
Intégration de l'itinéraire dans un plan global
Attachement du plan global au gestionnaire de l'exploitation
Insertion de l'application du plan global dans l'agenda de la simulation

X : Extension du développement des opérations
Identification sommaire : opérations sur le blé ; couper la luzerne ; apporter du fourrage au troupeau
Expression complémentaire de spécifications de ressources, au besoin

XI : Un aspect du système opérant : les horaires de travail de la main d'œuvre
Ce qui rend une unité de main d'œuvre disponible
Ce qui rend une unité de main d'œuvre indisponible

XII : Extension du développement des activités primitives
Identification sommaire : activités sur le blé ; la coupe de la luzerne ; l'apport de fourrage au troupeau
Expression complémentaire de spécifications de ressources, au besoin
Expression des conditions d'ouverture et de fermeture

XIII : Agrégation des activités primitives en des morceaux de plan
La conduite du blé
L'itération des coupes de luzerne, et intégration au plan global
La répétition régulière de l'affouragement, et intégration au plan global

XIV : Introduction de contraintes vécues pour ne pas produire des conduites impossibles
Une unité de main d'œuvre ne peut faire (dans notre domaine) qu'une chose à la fois
Une unité de main d'œuvre ne peut se trouver qu'en un lieu, en un instant donné

PHASE 3 : Contrôle final du code et empaquetage de la base de connaissances

XV : Vérification du code
Diagnostic et réparation de sources potentielles d'erreurs fatales
Diagnostic et réparation de fuites de mémoire

Annexe 1 : Livraison de la base de connaissances ...
... à un utilisateur qui fut le développeur de la base
... à un utilisateur qui exploite un simulateur qu'il n'a pas développé

PHASE 4 : Utiliser le simulateur et traiter les données

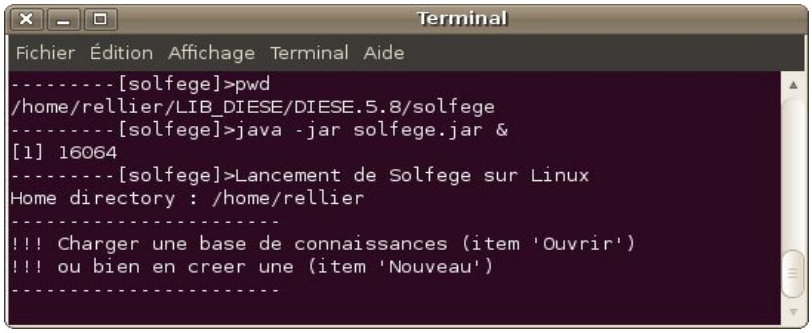
Annexe 2 :
Usage, en entrée, de spécifications de sorties
Internalisation, dans la base, de fonctions de sortie

ACTE I : Les préparatifs

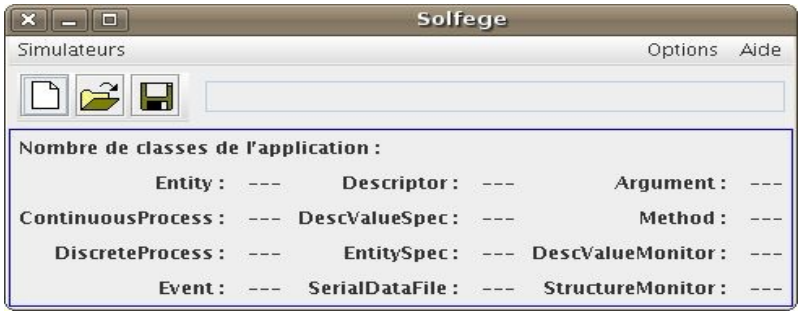
Etape 1 : Lancement de l'environnement de développement.

On rappelle le pré-requis d'avoir installé complètement et correctement l'environnement DIESE, dont (et donc) l'interface de développement Solfege.
Pour lancer l'interface de développement, passer la commande suivante dans un terminal :

```
>java -jar solfege.jar &
```



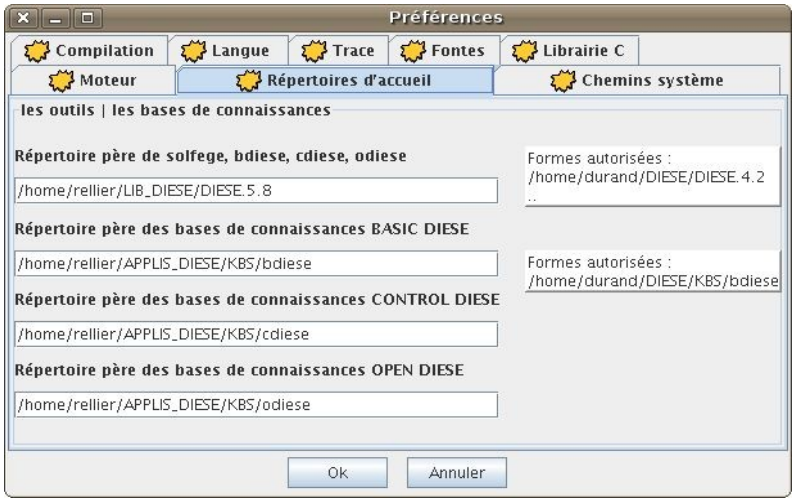
Une fenêtre apparaît :



Etape 2 : Adaptation de l'interface au contexte local

Dans le menu 'Options', choisir l'item 'Préférences'. Une seconde fenêtre s'ouvre : ⏩

Le choix 'CONTROL DIESE' doit être fait dans l'onglet 'Moteur'. Dans l'onglet 'Répertoire d'accueil', indiquer ou vérifier les chemins à renseigner. Les répertoires correspondant doivent exister. ⏩



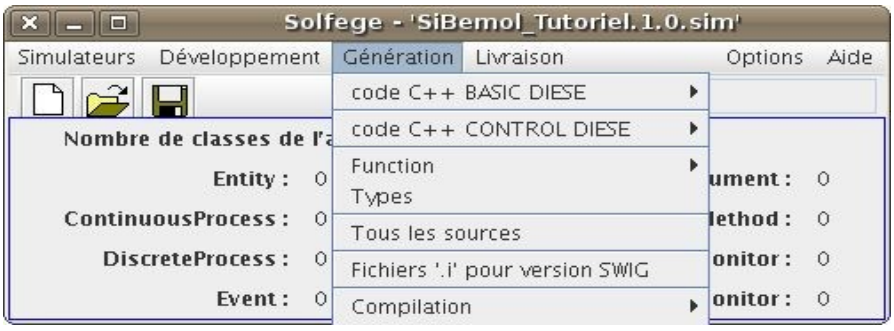
Dans l'onglet 'Chemins système', indiquer la commande locale pour l'éditeur de texte et le navigateur web. Ignorer le champ pour la librairie perl. ⏩

Cliquer sur le bouton 'Ok' pour enregistrer ces préférences locales. La fenêtre se ferme.

Etape 3 : Identification et initialisation d'un développement nouveau

3.1 Le répertoire. Dans le menu 'Simulateur', choisir l'item 'Nouveau' (ou bien cliquer sur l'icône la plus à gauche dans la barre des icônes). Une fenêtre apparaît, qui permet de i) naviguer dans le système de fichiers local jusqu'au répertoire dans lequel on souhaite installer la base, ii) compléter, dans le champ 'Nom de fichier', le chemin de ce répertoire par le nom du répertoire de la base elle-même. Le répertoire de la base de connaissances a le même nom que la base. Ici : SiBemol_Tutoriel, mais le lecteur peut choisir un autre nom. On conseille de suffixer le nom de la base (donc le nom du répertoire) avec un couple de numéros de version, un majeur (pour les évolutions ... majeures) et un mineur. ⏩

Cliquer sur le bouton 'Enregistrer'. La fenêtre suivante apparaît, avec deux nouveaux menus : ⏩
- par le menu 'Développement', on va modéliser le domaine d'application en déclarant et en spécifiant des objets, guidé par l'ontologie sur laquelle est fondé DIESE.
- par le menu 'Génération', on va traduire ces spécifications en code C++, et on va assembler ce code en un simulateur exécutable.



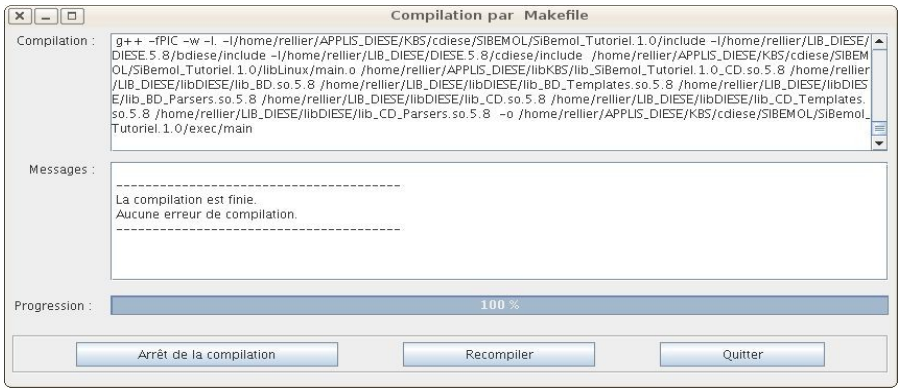
3.2 Le fichier Makefile. Il s'agit d'un fichier de directives de génération du simulateur exécutable à partir du code 'source' et de la librairie DIESE. Dans le menu 'Génération', cliquer sur l'item 'Générer le Makefile'. Une cadre d'avertissement apparaît, avec un texte débutant par "Le fichier des spécifications d'entêtes C++ doit être créé...". Lire

son contenu, cliquer sur le bouton 'Ok'. Ouvrir l'onglet 'Librairie C' de l'item 'Préférences' du menu 'Options'. puis simplement valider par le bouton 'Ok'. Cet onglet permet de compléter (mais c'est inutile ici et à ce stade) la liste des ressources C++ requises par le code de l'application. Se reporter à la documentation de Solfege pour la manipulation de cette fonctionnalité.
Important : dans le menu 'Génération', cliquer une seconde fois sur l'item 'Générer le Makefile', pour prendre en compte les indications (ici encore neutres) de l'onglet 'Librairie C'.
Le fichier Makefile est fabriqué une et une seule fois, tant que l'application repose sur la même version de la librairie DIESE, et tant que les indications dans l'onglet Librairie C' sont inchangées.

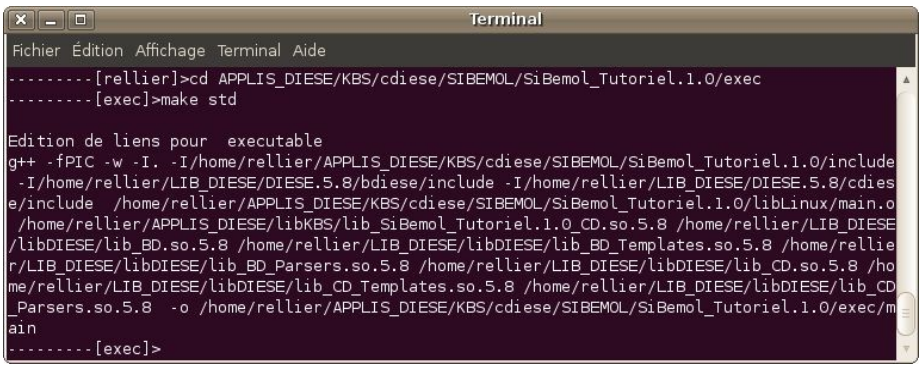
3.3 Une toute première génération du simulateur. Au cours du développement de la base, le simulateur exécutable sera généré souvent, pour vérifier de manière incrémentale l'acceptabilité du code C++ généré. Parce que, le plus souvent, un code C++ incorrect est la trace d'un défaut de spécification des objets (par le menu 'Développement'). La génération du simulateur est un couple de deux actions : la génération du code C++ et l'application des directives du fichier Makefile.
Le développement des objets n'ayant pas encore débuté, les fichiers de code C++ générés seront vides de contenu en termes de spécifications , mais les fichiers seront créés, chacun avec leur entête propre, et l'assemblage du code de l'application (vide) avec la librairie DIESE ne devra pas échouer. La répétition régulière de la génération du simulateur sera alors initiée.

3.3.1 La génération du code C++. Dans le menu 'Génération', cliquer sur l'item 'Tous les sources'. Un message indiquant la fin de la génération peut être visible, furtivement, dans le champ à droite de la barre des icônes.
Important : Si, au cours d'une phase de développement, un ou plusieurs objets nouveaux sont créés, le choix de l'item 'Tous les sources' devra être fait pour la génération du code. Au contraire, si le développement ne consiste qu'à compléter et/ou modifier des objets déjà définis, on pourra choisir, par mesure d'économie, de ne générer que la partie du code incluant ces objets-là. On en rencontrera des applications.

3.3.2 La compilation du code C++. Dans le menu 'Génération', choisir l'item 'Compilation' et cliquer sur le choix 'Compilation pour mode commande'. Une fenêtre apparaît, dans laquelle défile la trace de l'exécution des directives du Makefile. Une fois la compilation achevée, le cadre du bas indique que la compilation a réussi. Cliquer sur le bouton 'Quitter'. ⏩



Remarque : il existe une autre manière de compiler le code C++. Le fichier Makefile ayant été placé, par Solfege, dans un des sous-répertoires de 'SiBemol_Tutoriel.1.0/', en l'occurrence le répertoire 'exec/', et sachant que les directives du Makefile sont mise en œuvre par un utilitaire présent dans le système d'exploitation, en l'occurrence l'utilitaire 'make', la compilation peut être faite en écrivant et exécutant la ligne de commande suivante dans 'SiBemol_Tutoriel.1.0/exec/' :
`>make std`
Lorsque le fichier de directives n'est pas désigné dans la ligne de commande, c'est par défaut le fichier Makefile qui est traité ; c'est bien ce qu'on veut. Le mot `std` est un des "buts" défini dans le fichier Makefile. Ce but est la génération du simulateur qui sera exploité en 'mode commande', c'est-à-dire dans un terminal. ⏩



On remarque que cette seconde génération de l'exécutable ne compile pas chaque fichier 'source', puisqu'aucune modification n'a été faite depuis la précédente réalisée dans Solfege. Il n'y a plus qu'à réaliser l'assemblage des fichiers 'objet' (*.o placés, pour information, dans le répertoire 'SiBemol_Tutoriel.1.0/libLinux') avec la librairie DIESE.

Un autre but est la génération du simulateur qui sera exploité en 'mode interactif', c'est-à-dire par l'interface Mi_Diese. Pour la réalisation de ce but, on écrira la ligne de commande suivante :

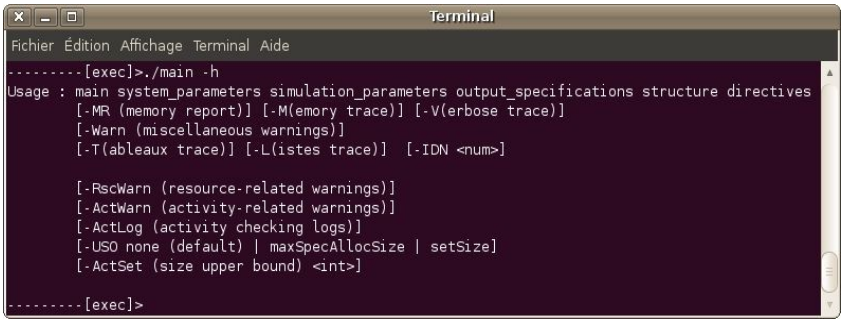
```
>make Ihm
```

Un résultat identique est obtenu dans Solfege en choisissant 'Compilation pour mode interactif'.

3.3.2 Une toute première exécution du simulateur. Dans le terminal ouvert sur '.../exec/', lancer la commande suivante, pour vérifier que l'exécutable est ... exécutable :

```
>./main -h
```

L'argument '`-h`' engage le programme '`main`' à afficher l'écriture standard de la commande de lancement de la simulation, et les options autorisées. On y reviendra plus loin. La réponse du programme est celle-ci (on y reviendra aussi plus loin) : ⏩

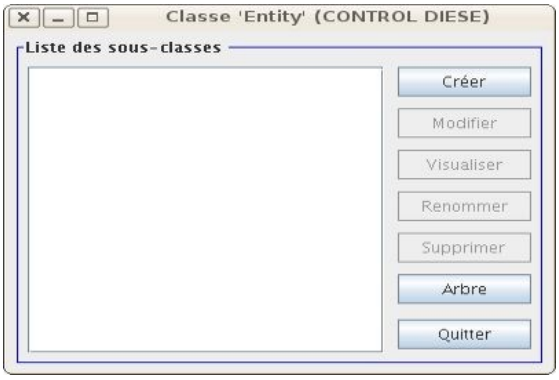


Etape 1: Les classes d'entités visibles au premier abord

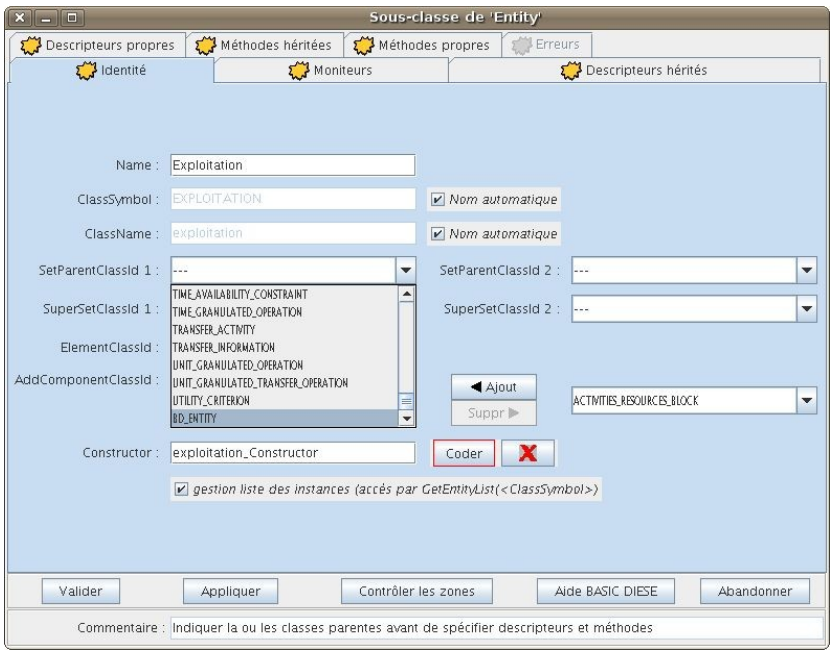
A la lecture de l'énoncé du problème, on juge utile la création d'un premier ensemble de sous-classes d'entités : exploitation, blé, luzerne, prairie, vache. Par choix anticipatoire, ou dicté par l'expérience, on décide d'installer les classes C++ suivantes : Exploitation, Parcellaire, Parcelle (un élément du parcellaire), Troupeau, Animal (un élément du troupeau), Vegetation, Culture, Ble, Luzerne, HerbeNaturelle.

Important : On note d'ores et déjà une convention de nomenclature : Les noms de classes ne contiennent ni espaces ni accents, et sont composés d'un ou plusieurs termes débutant par une majuscule et continuant par des minuscules.

1.1 Pour créer la classe Exploitation, dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity'. Apparaît ce qu'on appelle une fenêtre de choix, parce qu'elle nous permettra ultérieurement de choisir la classe à manipuler. ⏩

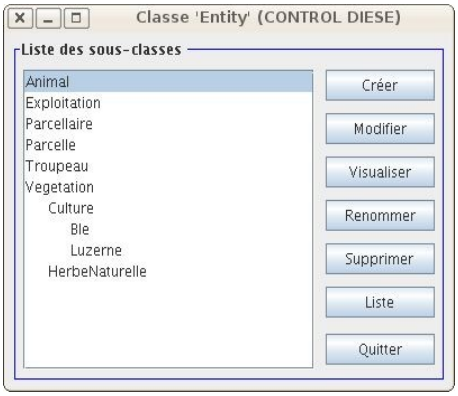


Puisqu'aucune classe n'a encore été créée, cliquer sur le seul bouton actif et utile : 'Créer' Une fenêtre dite de modification apparaît, avec des champs encore vides.
Taper le nom C++ de la classe : Exploitation dans le champ 'Name'. Les deux champs situés juste au-dessous se remplissent automatiquement, parce que, et c'est la situation par défaut, les deux cases 'Nom automatique' sont cochées. Le champ 'ClassSymbol' contient ce qu'on appelle le "symbole de classe" de la classe. Le champ 'ClassName' contient ce qu'on appelle le "nom de classe" de la classe. Ces deux attributs de la classe seront souvent exploités au cours du développement.
Il est nécessaire de désigner la classe dont Exploitation est une sous-classe directe. A ce stade, le seul choix possible est la classe générique prédéfinie dans DIESE : Entity. Dans la liste soulevée en cliquant le bouton '▾' sur la droite du champ 'SetParentClassId 1', descendre pour surligner et cliquer le mot BD_ENTITY (c'est le symbole de classe de la classe Entity). BD_ENTITY apparaît comme valeur du champ.



Valider cette spécification sommaire de l'exploitation en cliquant le bouton 'Valider'. La classe Exploitation apparaît dans la fenêtre de choix des entités.

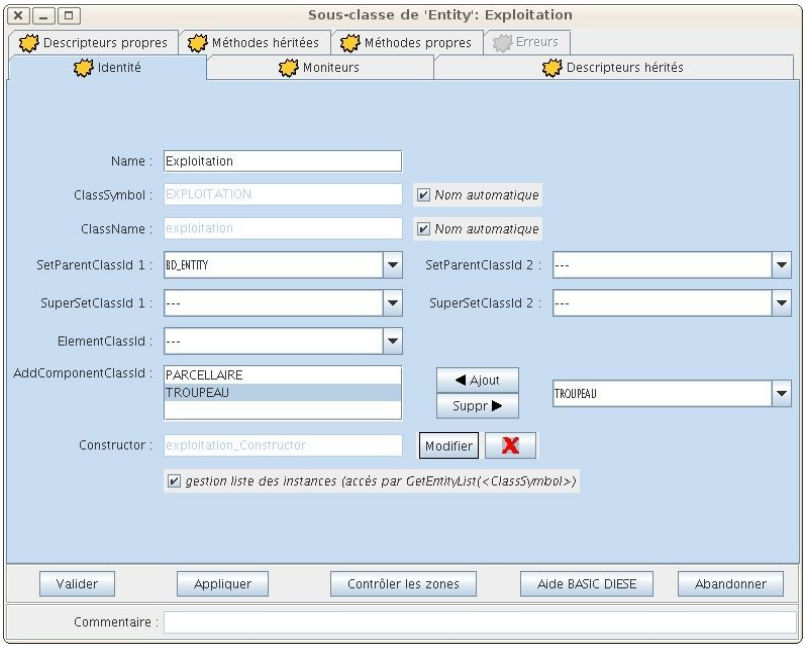
1.2 Pour créer les autres classes, procéder de la même manière jusqu'à la classe Vegetation de la liste en tête de cette étape 1.
Pour les classes en fin de liste (Culture, Ble, Luzerne, HerbeNaturelle), on comprend qu'on choisisse de les faire hériter de la classe Vegetation.
Quant aux classes Ble et Luzerne (mais pas la classe HerbeNaturelle), on choisit Culture comme leur classe-mère directe (et par conséquent Vegetation comme leur classe-ancêtre). En conséquence :
- au moment de valuer le champ 'SetParentClassId 1', de Culture et HerbeNaturelle, descendre dans la liste pour surligner et cliquer le mot VEGETATION (c'est le symbole de classe de la classe Vegetation).
- au moment de valuer le champ 'SetParentClassId 1', de Ble et Luzerne, descendre dans la liste pour surligner et cliquer le mot CULTURE (c'est le symbole de classe de la classe Culture).
Cliquer sur le bouton 'Arbre' pour faire apparaître, dans la fenêtre de choix, une indentation imageant la hiérarchie des classes. On peut revenir à une vue "à plat" en cliquant alternativement sur le bouton 'Liste'. ⏩



1.3 Installer les premières relations structurelles, celles qui existent ... :
- entre les classes Parcellaire et Parcelle : une parcelle est un élément (au sens de l'ontologie) du parcellaire ;
- entre la classe Exploitation d'une part et les classes Parcellaire et Troupeau d'autre part : les deux dernières sont des composants (au sens de l'ontologie) de la première.

Pour installer le premier lien (Parcellaire/Parcelle), cliquer Parcellaire dans la fenêtre de choix, puis sur 'Modifier'. Dans la liste soulevée en cliquant le bouton ' ▾ ' sur la droite du champ 'ElementClassId', descendre pour surligner et cliquer le mot PARCELLE, lequel apparaît comme valeur du champ.
Valider ce complément de spécification du parcellaire en cliquant le bouton 'Valider'.

Pour installer le second lien (Exploitation/{Parcellaire Troupeau}), cliquer Exploitation dans la fenêtre de choix, puis sur 'Modifier'. Dans la liste soulevée en cliquant le bouton ' ▾ ' tout à fait à droite du champ 'AddComponentClassId', descendre pour surligner et cliquer le mot PARCELLAIRE, puis cliquer le bouton 'Ajout'. Procéder de même pour ajouter le mot TROUPEAU dans le cadre des classes de composants de l'exploitation. ⏩



Valider ce complément de spécification de l'exploitation en cliquant le bouton 'Valider'.

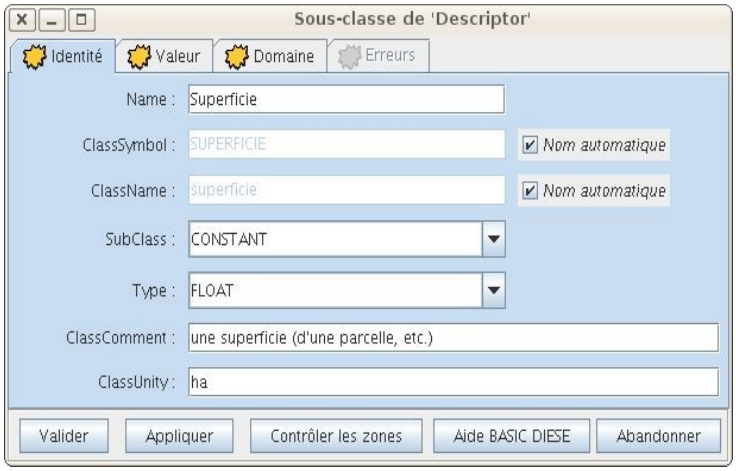
1.4 Sauvegarder ! Il est maintenant utile de sauvegarder le travail déjà réalisé en tant que version courante de la base dans le système de fichiers. A cette fin cliquer simplement sur l'icône figurant une disquette dans la barre des icônes. Attendre (très peu de temps à ce stade) l'affichage du message de fin de l'opération dans le champ des messages.

Etape 2 : Doter les classes d'entités de descripteurs.

Les classes véhiculent la définition d'un concept. Par exemple, une parcelle est définie (entre autres choses) par sa superficie et comme le support d'un couvert végétal, ou la localisation d'un troupeau. Notre troupeau, implicitement homogène, peut être défini comme un certain nombre d'exemplaires d'un animal représentatif. Un blé est caractérisé par une quantité acquise d'unités de développement. L'herbe est décrite par sa croissance nette.
Ainsi décide-t-on de créer les descripteurs suivants :
- en "constants" : Superficie, AnimalRepresentatif, ParcelleSupport
- en "variables" : NombreAnimaux, QuantiteUnitesDev, CroissanceNette, CouvertVegetal

Si les trois premiers descripteurs "variables" semblent mériter ce qualificatif, ce n'est pas le cas du quatrième. En fait, on envisage d'établir une relation réciproque entre une parcelle et sa culture, parce que c'est souvent utile. Une manière de faire cela commodément est d'installer un démon sur le descripteur CouvertVegetal de la parcelle, et de confier à ce démon de valuer le descripteur ParcelleSupport de la culture. Comme, on le rappelle ici, seuls les descripteurs variables peuvent être surveillés par un démon, CouvertVegetal doit être variable. L'installation de ce démon est faite plus loin.

2.1 Pour créer le descripteur Superficie, dans le menu 'Développement', item 'BASIC DIESE', choisir 'Descriptor', puis 'Constant'. Apparaît la fenêtre de choix des descripteurs constants, encore vide.
Cliquer sur 'Créer'.Taper Superficie dans le champ 'Name'. Choisir FLOAT pour le champ 'Type' (pour pouvoir exprimer des décimales).Ajouter un commentaire et une unité (optionnels, mais recommandés). ⏩



Cliquer sur 'Valider'.

2.2 Pour les autres descripteurs constants et variables, procéder de la même manière, avec les consignes suivantes :
- le type de AnimalRepresentatif (C), ParcelleSupport (C), CouvertVegetal (V) est P_ENTITY (la valeur est une entité)
- le type de NombreAnimaux (V) est INT
- le type de QuantiteUnitesDev (V), CroissanceNette (V) est FLOAT

Ajouter librement des commentaires, et des unités le cas échéant, en sachant que ces informations pourront être complétées ou modifiées par la suite. Noter qu'une unité ne peut être spécifiée que pour les types confirmés numériques.

Pour le descripteur AnimalRepresentatif, exprimer que la valeur ne peut être qu'une instance de la classe Animal. Pour ce faire, dans la fenêtre de modification de la classe Animal, onglet 'Domaine', sélectionner ANIMAL dans le menu à droite sur la ligne 'Classes', puis cliquer sur 'Ajout'. ⏩

2.3 Attachement des descripteurs aux entités. Les descripteurs créés vont désormais apparaître dans l'onglet 'Descripteurs propres' de la fenêtre de modification des classes d'entités. L'attachement pourra être accompagné de la désignation d'une valeur, laquelle sera

affectée au descripteur lors de chaque création d'une instance de la classe.

2.3.1 Le descripteur Superficie pour les parcelles. Ouvrir la fenêtre de modification pour la classe Parcelle. Sélectionner l'onglet 'Descripteurs propres'. Et non l'onglet 'Descripteurs hérités' : ceux-là sont prédéfinis au niveau des classes mère et ancêtres, et peuvent recevoir dans cet onglet une valeur caractéristique de la classe.
Puisque Superficie est un descripteur constant, c'est le bouton ' ▾ ' sur la droite de la partie supérieure qu'il faut cliquer, pour faire apparaître la liste des descripteurs candidats. Cliquer sur Superficie, puis sur le bouton 'Ajout'.
Profiter de l'ouverture de cette fenêtre pour ajouter de la même manière le descripteur variable CouvertVegetal.

2.3.1 Les autres descripteurs pour les autres classes.

Ouvrir la fenêtre de modification pour la classe Troupeau. Ajouter les descripteurs AnimalRepresentatif, NombreAnimaux et ParcelleSupport.
Ouvrir la fenêtre de modification pour la classe Ble. Ajouter le descripteur QuantiteUnitesDev.
Ouvrir la fenêtre de modification pour la classe HerbeNaturelle. Ajouter le descripteur CroissanceNette.

2.4 Sauvegarder !

Etape 3 : Nouvelle génération du code C++ : dernier rappel de la procédure

3.3.1 La génération du code C++. Dans le menu 'Génération', cliquer sur l'item 'Tous les sources' (et non pour une classe particulière, parce que des objets nouveaux sont créés : des entités, par exemple).

3.3.2 La compilation du code C++. Dans le menu 'Génération', choisir l'item 'Compilation' et cliquer sur le choix 'Compilation pour mode commande', ou bien exécuter la ligne de commande suivante dans 'SiBemol_Tutoriel.1.0/exec/' :

```
>make std
```

ACTE III : De la base de connaissances générales à la description d'un cas particulier

Le système étudié est vu comme l'assemblage d'objets dont la définition et la constitution sont encapsulés dans les classes. A ce stade, ce sont les classes Exploitation, Parcellaire, Parcelle, Troupeau, Animal, Vegetation, Culture, Ble, Luzerne, HerbeNaturelle. Pour manipuler ces objets conceptuels via un programme, il faut en avoir une matérialisation informatique : ces structures de données prennent le nom général d' « instance », parce que chacune est le résultat de l' « instanciation » d'une classe. Par exemple, la parcelle de blé de l'exploitation qu'on va simuler est une instance de la classe conceptuelle Parcelle.

Par extension, la description de l'exploitation à simuler est un ensemble de directives d'instanciation, organisé en une sorte de texte. Pour rédiger ce texte, on utilise bien entendu un langage, en respectant ses règles lexicales et grammaticales. Par exemple, voici le texte par lequel on déclarerait l'exploitation, sans précision sur sa structure :

```
+   I   exploitation   monEA ;
```

Voici le texte par lequel on déclarerait l'exploitation, en précisant un peu sa structure :

```
+   I   exploitation   monEA
      +   C  parcellaire   monParcellaire ;
      +   C  troupeau   monTroupeau ;
;
```

On comprend qu'on peut augmenter le texte pour compléter la description. Le lexique et la grammaire du langage sont complètement présentés dans la documentation de DIESE. Ici, le lecteur-manipulateur est invité à copier/coller les paragraphes utiles à partir d'un des fichiers accompagnant ce texte, ou bien à partir de ce texte lui-même.

Dans cet Acte, on va donc installer une première version des fichiers contenant la description de l'exploitation à simuler, puis vérifier que cette description est correctement interprétée par notre programme.

Etape 1 : Aménager l'espace des données des simulations

Le lecteur qui a placé la base de connaissances dans …/APPLIS_DIESE/KBS/cdiese/SIBEMOL (revoir l'étape 3.1 de l'Acte I) pourra placer les données des simulations sous le répertoire …/APPLIS_DIESE/SIM/cdiese/SIBEMOL.

1.1 Création du répertoire des données. Créer un répertoire SIBEMOL_TUTORIEL_1.0, par les commandes suivantes dans un terminal (ou dans la fenêtre de navigation graphique) :

```
>cd /home/…/APPLIS_DIESE/SIM/cdiese/SIBEMOL
>mkdir SIBEMOL_TUTORIEL_1.0
>cd  SIBEMOL_TUTORIEL_1.0
```

Dans ce répertoire, créer les cinq fichiers suivants : sim1.par, sim1.sim, sim1.osp, sim1.str, sim1.dir. Dans un terminal, la commande est la suivante, sur le seul exemple du premier de ces fichiers :

```
>touch sim1.par
```

1.2 Lier le répertoire des exécutions et le répertoire des données. La commande d'exécution (main) fait référence aux fichiers de données, naturellement. Pour faciliter l'écriture des fichiers dans la ligne de commande, il est commode d'installer, dans le répertoire d'exécution ce qu'on appelle un « lien symbolique » avec le répertoire des données. Cela s'opère par la commande suivante … :

```
>ln  -s  /home/…/APPLIS_DIESE/SIM/cdiese/SIBEMOL/ SIBEMOL_TUTORIEL_1.0  SIM
```

… en remplaçant «`/home/…/APPLIS_DIESE/`» par le chemin local choisi.

Etape 2 : Ecrire le fichier des paramètres de la simulation

Il s'agit du fichier sim1.sim.
- Editer ce fichier avec l'éditeur de texte préféré
- Ouvrir le fichier ressources_SIM.txt, et sélectionner le § AIII_E2 (éviter de récupérer les deux lignes d'encadrement contenant la chaîne « § AIII_E2 »)
- Coller ce § dans la fenêtre d'édition de sim1.sim
- Sauvegarder le fichier et quitter l'éditeur

On établit dans ce fichier l'unité de progression de l'horloge de la simulation (ici 1 heure), la période réelle simulée (réduite aux cinq premiers jours, suffisants pour vérifier le démarrage correct de la simulation), et l'emplacement du répertoire de sortie par défaut des résultats.

Noter aussi l'initialisation de la "graine" du générateur de nombres aléatoires présent dans DIESE. Dans la présente forme (`INITRAND;`, sans argument), la graine change d'une exécution à l'autre.

Etape 3 : Ecrire une première version du fichier de la structure du système

3.1 Puisqu'il s'agit de lui, écriture du fichier sim1.str

- Editer ce fichier avec l'éditeur de texte préféré
- Ouvrir le fichier ressources_SIM.txt, et sélectionner le § AIII_E3.1
- Coller ce § dans la fenêtre d'édition de sim1.str
- Sauvegarder le fichier et quitter l'éditeur

Dans ce fichier, on commence par inclure la lecture d'un fichier « tiers » (systemePilote.inc) dans lequel on va décrire un des composants principaux du système simulé (en référence à l'ontologie) : le système piloté lui-même (Les deux autres seront le système opérant et le système de décision. Ce système d'inclusion permet de compartimenter la description globale. Chaque compartiment est plus facilement manipulable, et le système global est lisible de manière synoptique.

3.2 Ecriture du fichier inclus systemePilote.inc

- Editer ce fichier avec l'éditeur de texte préféré
- Ouvrir le fichier ressources_SIM.txt, et sélectionner le § AIII_E3.2
- Coller ce § dans la fenêtre d'édition de systemePilote.inc
- Sauvegarder le fichier et quitter l'éditeur

Dans ce fichier, on complète la description de l'exploitation à simuler commencée dans l'introduction de cet Acte. On note la manière d'affecter une valeur à un descripteur (par exemple : « superficie = 7. ; », et la manière de faire référence à une entité créée préalablement lors de l'interprétation « top-down » du fichier (par exemple « couvertVegetal = <I><, monBle> »). Le dernier bloc fait créer une instance de la classe prédéfinie ControlledSystem, laquelle possède le descripteur prédéfini AttachedEntity. Ce dernier reçoit naturellement comme valeur l'instance préalablement créée de la classe Exploitation.
Noter la nécessité de clore l'interprétation de ce fichier par une directive « END INCLUDE ; ». La « main » est ainsi redonnée à l'interprétation du fichier « appelant », ici sim1.str.

Etape 4 : Vérification de la bonne interprétation de ces fichiers

Dans le terminal d'exécution, lancer la commande suivante :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir
```

Ce qui doit provoquer l'affichage suivant dans le même terminal : ⏭

Le message d'avertissement ".. composant ... MANAGER non présent..." est normal, puisqu'on n'a encore installé, parmi les trois composants du système de production, que le composant "système piloté". Idem pour le composant OPERATING_SYSTEM.

Si cette trace n'est affichée, et si apparaît un message semblant décrire une erreur, il s'agit vraisemblablement, à ce stade, d'une mauvaise désignation de fichier. Bien vérifier les chaînes de caractères des chemins des répertoires et des noms de fichiers.

Le lecteur pourra introduire volontairement dans les fichiers des fautes lexicales (nom de classe inconnu, par exemple, "+ I" remplacé par "+ J", etc...), ou des fautes grammaticales ("+ I blé monBle" remplacé par "+ blé monBle", par exemple) et constater que la lecture du fichier s'arrête avec un diagnostic d'erreur. Dé-commenter la ligne "TRACE PARSING;" pour suivre l'interprétation. pas à pas.

Remarques :

- les fichiers sim1.par, sim1.osp et sim1.dir, encore vides, ne seront complétés que dans les étapes suivantes, lorsque le besoin sera là.

- notamment pour sim1.dir, mais c'est aussi le cas pour sim1.sim, Solfège permet d'en générer une version standard, que le développeur n'a plus qu'à ajuster à la marge.

L'énoncé du problème externalise dans un fichier séquentiel les valeurs de la croissance nette de l'herbe, par opposition à une "internalisation" du calcul sous la forme de processus DIESE. De manière générale, cette externalisation est adoptée pour les connaissances qu'on a décidé de ne pas inclure dans le fonctionnement du système. Elles sont alors considérées comme partie de l'environnement, par construction incontrôlable, du système.

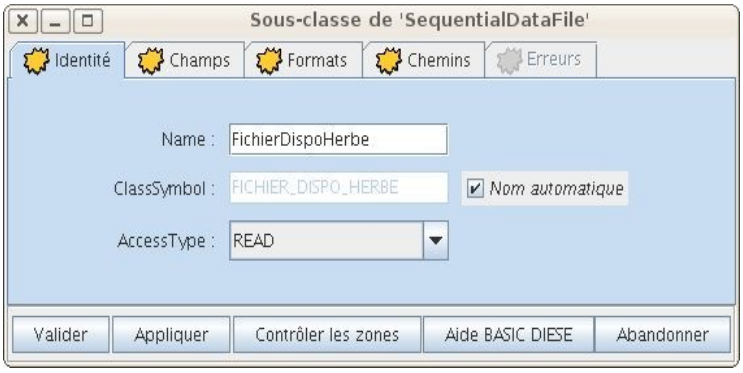
DIESE propose **une classe de processus dédiée** à la lecture de fichiers constitués d'une suite d'enregistrements, chacun correspondant à une date postérieure à celle de l'enregistrement précédent, et tels que la différence entre deux dates successives soit constante. C'est typiquement le cas des fichiers de séries météorologiques.

Cette classe de processus est indissociable d'une autre classe prédéfinie dans DIESE, **la classe des fichiers séquentiels** (SequentialDataFile). Cette classe est le modèle en mémoire, manipulable par un programme, du fichier externe résidant dans le système de fichiers de l'utilisateur.

Etape 1 : Le modèle du fichier externe de la croissance nette de l'herbe

Fermer, si besoin, les fenêtres de choix des entités et des descripteurs (par le bouton 'Quitter').

Dans le menu 'Développement', item 'BASIC DIESE', choisir 'SerialDataFile', puis 'SequentialDataFile'. Apparaît la fenêtre de choix des fichiers séquentiels, encore vide.
Cliquer le bouton 'Créer'. Apparaît une fenêtre de modification : ⏩



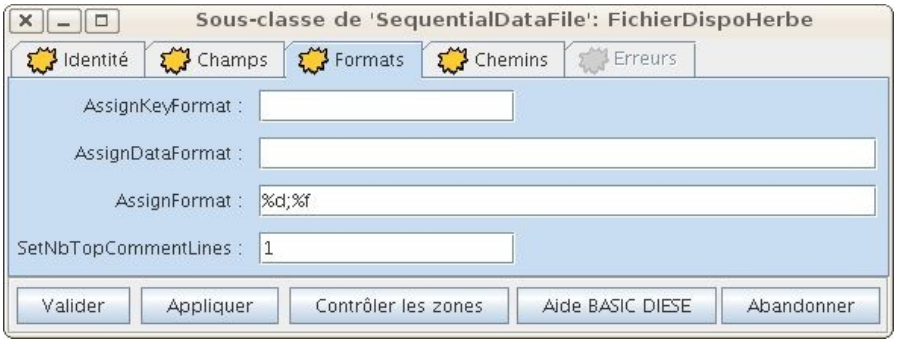
On a indiqué le nom de classe FichierDispoHerbe, et on a maintenu que le fichier est en mode "lecture".

Dans l'onglet 'Champs', on va décrire la structure d'un enregistrement comme une succession de champs. On sait qu'un enregistrement du fichier commence par le numéro du jour (depuis le 1er janvier) et continue par une valeur de croissance nette.

Cliquer sur le bouton '▼' sur la droite du champ dont la valeur est encore '---' exhibe une liste de symboles candidats pour les champs. Solfege les a générés à partir des descripteurs qu'il connaît (prédéfinis et applicatifs), en les préfixant par le mot FIELD. Ignorer cette liste, et taper NDAY_FROM_ORIG (ou autre chose, mais il faudra s'en souvenir) en lieu et place des '---'. Puisque ce champ contient un nombre entier, un clic sur le bouton 'Ajout' prend en compte le symbole INT proposé par défaut dans le champ du dessous.

En lieu et place de NDAY_FROM_ORIG, taper NET_GROWTH comme symbole pour le second champ, choisir le type FLOAT juste en-dessous, puis cliquer sur 'Ajout'. ⏩

Dans l'onglet 'Formats', enfin, taper la chaîne "%d;%f" (attention ! sans les guillemets) dans le champ 'AssignFormat'. Puis taper 1 dans le champ SetNbTopCommentLines, puisque ce type de fichiers comprend une première ligne, et une seule, de commentaires. On pourrait fixer, au moins partiellement, l'emplacement du fichier dans l'onglet 'Chemins', mais on décide de ne pas le faire ici. Cette information sera externalisée, donc modifiable par l'utilisateur. ⏩

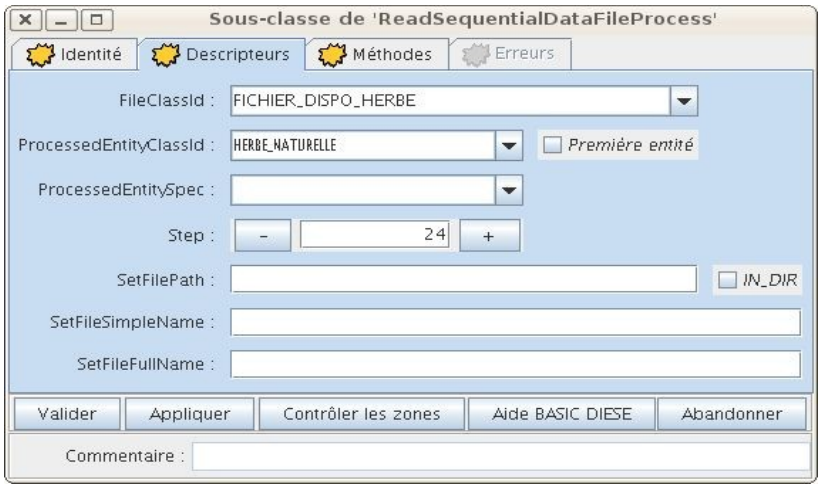


Sauvegarder !

Etape 2 : Le processus de lecture du fichier de la croissance nette

Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Process', puis 'ReadSequentialDataFileProcess'. Apparaît la fenêtre de choix des processus de ce type, encore vide.
Cliquer le bouton 'Créer'. Dans la fenêtre de modification, on indique le nom de classe LectureCroissanceHerbe : ⏩

Dans l'onglet 'Descripteurs', on indique que le processus est spécialisé dans la lecture d'un fichier de symbole de classe FICHIER_DISPO_HERBE, et que ce processus va modifier l'état d'une entité de classe HERBE_NATURELLE (on ne sait bien entendu pas laquelle au stade de la définition du processus). Enfin, on tient compte de la périodicité des enregistrements du fichier (journalière) en donnant la valeur 24 au 'pas' du processus (puisque l'unité d'horloge est de une heure (cf. Acte III, Etape 2). ⏩



2.1 Initialisation de la lecture du fichier. Dans l'onglet 'Méthodes', en cochant la case 'Nom automatique' dans la ligne du champ 'SetInitialize', puis en cliquant sur le bouton 'Créer', on ouvre une fenêtre dans laquelle on programme l'initialisation de la lecture du fichier. Solfege propose un contenu standard, qu'il suffit très généralement de modifier à la marge, notamment dans la partie qui positionne le lecteur juste après les lignes inutilisées.

- Ouvrir le fichier ressources_FCT.txt, et sélectionner le § AIV_E2.1

- Remplacer le contenu de la fenêtre d'édition par ce §. On remarque que seule la partie finale est en fait modifiée : celle qui récupère la date de début de la simulation, fixée dans le fichier sim1.sim (cf. Acte III Etape 2), qui calcule le

nombre de jours, donc de lignes, depuis la date du premier enregistrement, et qui enfin les saute après avoir sauté le nombre de lignes d'entête déclaré pour ce type de fichiers (*cf.* Etape 1 ci-dessus).

- Sauvegarder le fichier et quitter l'éditeur

2.2 Changement de l'état du système. Dans l'onglet 'Méthodes', en cochant la case 'Nom automatique' dans la ligne du champ 'SetPostRead', puis en cliquant sur le bouton 'Créer', on ouvre une fenêtre dans laquelle on programme, non pas la manière de lire chacun de ses enregistrement (les champs et le format de lecture sont en effet des connaissances attachées au fichier, exploitées par le moteur de simulation de DIESE), mais la manière de modifier l'état du système en conséquence de la lecture.

- Dans le fichier ressources_FCT.txt, sélectionner le § AIV_E2.2

- Remplacer le contenu de la fenêtre d'édition par ce §. On remarque que l'instance de la classe de symbole HERBE_NATURELLE (annoncé dans le champ ProcessedEntityClassId de l'onglet 'Descripteurs') est obtenu par un couple de lignes généré par Solfege (on a juste choisi l'identifiant pHerbe pour clarifier son sens). Ce couple de lignes renverra, au moment de l'exécution, l'instance qu'on aura désignée lors de la création du processus avant son insertion dans l'agenda de l'événement qui le gère (dans cette application, on envisage de le faire dans le fichier sim1.str). Pour le reste, on ne fait que récupérer la valeur du champ utile (la croissance nette) pour l'affecter au descripteur CroissanceNette de l'instance d'herbe.

- Sauvegarder le fichier et quitter l'éditeur

- Terminer la spécification de ce processus en cliquant sur le bouton 'Valider'

Sauvegarder ! Et générer le code C++ par 'Tout le code'.

Etape 3 : Vérification du codage correct de la lecture de la croissance nette

3.1 Installation du fichier. Un fichier DISPO_HERBE.txt accompagne ce document. Dans le système de fichiers local, on suggère de le placer à côté du répertoire SIBEMOL_TUTORIEL_1.0, en référence à l'Etape 1.1 de l'Acte III :

```
>cd /home/…/APPLIS_DIESE/SIM/cdiese/SIBEMOL
>mkdir IN_FILES
>cd  IN_FILES
```

Dans ce répertoire IN_FILES, copier DISPO_HERBE.txt

3.2 Dans l'agenda des choses à faire, placement de cette lecture de fichier.

- Dans le répertoire des données (SIBEMOL_TUTORIEL_1.0), ouvrir le fichier sim1.str si ce n'est déjà fait.

- Coller en fin de fichier le § AIV_E3.2 extrait de ressources_SIM.txt. On y désigne le fichier à lire, et on précise que ce processus doit être lancé à 6 heures du matin du premier jour de la période simulée (t=6, ce qui donne à l'événement une priorité suffisante -on le constatera plus loin-). Enfin, on désigne l'instance d'herbe qui sera touchée par le processus.

- Dans ce même fichier, inhiber (par deux slashs) la directive d'affichage de la structure du système de production :

```
//AFFICHER STUCTURE <I><, pSP>;
```

3.3 Installation d'un moyen de trace de la lecture

- Dans le menu 'Génération' de Solfege, choisir 'Edition des fichiers source C++'. Apparaît une fenêtre avec la liste des fichiers C++ générés par la commande de compilation. Choisir le fichier UserContinuousProcess.cc et cliquer sur le bouton 'Editer'. En fin de la fonction 'lectureCroissanceHerbe_postReadProcess_Body', mettre en jeu les deux lignes de trace suivantes (de la bonne lecture d'un enregistrement), en enlevant les deux slashs initiaux :

```
int now = ...;
printf(...);...
```

- Sauvegarder cette modification, mais ne pas quitter l'éditeur.

3.4 Excécution de la simulation.

- Dans le répertoire d'exécution (cf. Acte III, Etape 4), lancer les commandes classiques de compilation (si le code a bien été généré à l'Etape 2) et d'exécution :

```
>make std
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir
```

Ce qui doit provoquer l'affichage suivant dans le même terminal :⏭.

On ne constate aucune trace de la lecture des enregistrements. Pour tenter un diagnostic sur cette anomalie, on relance la commande d'exécution en ajoutant l'option -V :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -V
```

Des messages nouveaux sont émis dans le terminal, mais toujours pas de trace de la lecture du fichier.

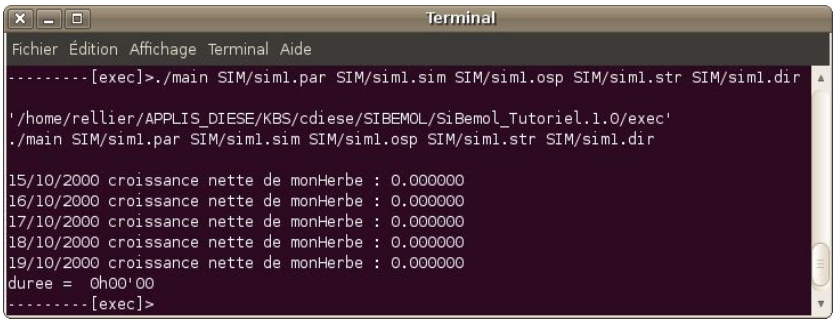
Analyse : La trace est émise par le processus de lecture, lequel est bien lancé par un événement qu'on a bien placé dans l'agenda, lequel est traité par le moteur de simulation, une fois celui-ci lancé. Tout est en principe en place ... sauf le lancement du moteur de simulation ! La documentation de BASIC DIESE nous dit en effet, dans sa page sur "les fichiers en entrée de type déclaratif" qu'une directive RUN doit être émise dans le fichier des directives de la simulation. Or, dans l'Acte III, Etape 4, on a repoussé à plus tard la rédaction du fichier sim1.dir.

En conséquence :

- dans le répertoire des données (SIBEMOL_TUTORIEL_1.0), ouvrir le fichier sim1.dir

- y coller le paragraphe § AIV_E3.4, copié dans le fichier ressources_SIM.txt. Il s'agit, à peu de choses près, du contenu généré par Solfège lorsqu'on choisit, dans le menu 'Livraison', l'item 'Génération des fichiers d'entrée, et l'option 'directives de simulation'. Sans détailler ici son contenu, on y note la présence de la directive RUN.

Désormais, la commande `./main ...` (sans l'option -V) doit générer la trace ci-dessous : ⏩



3.5 Désinstallation de la trace de la lecture

- Dans la fenêtre d'édition, encore ouverte, du fichier UserContinuousProcess.cc, inhiber les deux lignes de trace précédemment mises en jeu, en remettant les deux slashs initiaux:

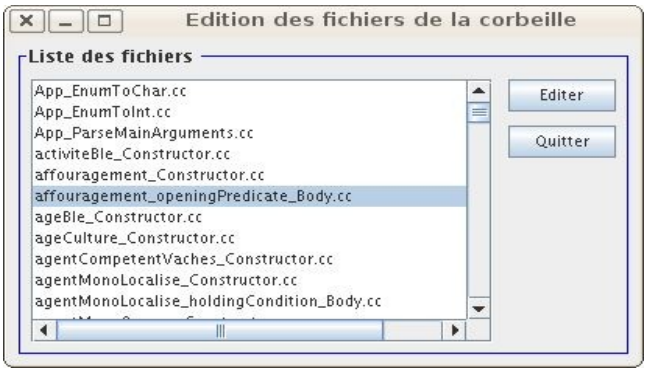
```
//int now = ...;
//printf(...);...
```

- Sauvegarder cette modification, et quitter l'éditeur.

Remarque : Accès aux versions antérieures du code

Solfege conserve, dans un répertoire de la base de connaissances (de nom .Trash/), le code des fonctions, corps de méthodes et constructeurs dans leur dernier état lors de journées de développement dans le passé. Chaque date fait l'objet d'un sous-répertoire dont le nom est de la forme aammdd/, par exemple 981207/ pour le 12 juillet 1998. Le sous-répertoire crée le plus récemment est accessible directement dans l'interface Solfege, et les autres indirectement.

- Dans le menu 'Simulateurs', choisir 'Corbeille', puis 'Contenu de la corbeille'. Apparaît une fenêtre de choix des fichiers enregistrés à cette date : ⏩



La sélection d'un fichier, suivie d'un clic sur 'Editer' ouvre une fenêtre d'édition pour ce fichier. A partir de là, on peut si besoin est, accéder aux versions enregistrées à des dates antérieures.

ACTE V : Les unités de développement du blé, vues comme un aspect de l'environnement

L'énoncé du problème stipule que certaines interventions culturales sur le blé sont "calées" sur l'atteinte d'un stade phénologique. Il stipule aussi que certains stades sont considérés atteints lorsque le blé a accumulé une certaine quantité d'"unités de développement" (UDev). Par exemple, le stade "gonflement" est atteint dès que 2900 UDev sont acquises.

La donnée de cette accumulation d'UDev a été externalisée dans un fichier d'enregistrements séquentiels, qui donne la valeur accumulée par le blé pour chaque jour de la simulation..

On va donc installer un processus de lecture de ce fichier, exactement similaire au processus de lecture de la croissance nette de l'herbe à l'Acte IV. Ce processus va modifier l'état du blé, plus précisément son descripteur QuantiteUnitesDev.

Selon l'énoncé, cette variable est suffisante comme indicateur pour déclencher les interventions culturales en question. Et la simulation de son évolution semble être une étape nécessaire mais suffisante. On va cependant introduire un développement complémentaire. Au lieu d'assurer le lien informationnel suivant :

lecture du fichier UDev –> descripteur QuantiteUnitesDev –> décision d'actions sur ce descripteur

on va installer le lien suivant :

lecture UDev –> desc. QuantiteUnitesDev –> suite de stades –> décision d'actions sur atteinte stade

Avec une double justification. Présenter un "pattern de modélisation" récurrent dans les systèmes de production végétale. Donner à notre développement la flexibilité de définir la transition des stades sur une autre base agro-climatique. C'est dans l'Acte VI que sera développé ce pattern.

Etape 1 : Le modèle du fichier externe des unités de développement

Fermer, si besoin, les fenêtres de choix ouvertes.

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'SerialDataFile', puis 'SequentialDataFile'. Apparaît la fenêtre de choix des fichiers séquentiels, avec le fichier FichierDispoHerbe déjà créé.
- Cliquer le bouton 'Créer'. Apparaît une fenêtre de modification , dans laquelle on indique le nom de classe FichierUdevBle, et on maintient que le fichier est en mode "lecture".

- Dans l'onglet 'Champs', décrire la structure d'un enregistrement comme une succession de champs. On sait qu'un enregistrement du fichier commence par le numéro du jour (depuis le 1er janvier) et continue par une valeur d'unités de développement accumulées.

- Taper NDAY_FROM_ORIG2 en lieu et place des '---'. Puisque ce champ contient un nombre entier, un clic sur le bouton 'Ajout' prend en compte le symbole INT proposé par défaut dans le champ du dessous.

- En lieu et place de NDAY_FROM_ORIG2, taper DEV_UNITS comme symbole pour le second champ, choisir le type FLOAT juste en-dessous, puis cliquer sur 'Ajout'.

- Dans l'onglet 'Formats' et son champ 'AssignFormat', taper la chaîne "%d;%f" (sans les guillemets). Puis taper 1 dans le champ SetNbTopCommentLines (ce type de fichiers comprend une première ligne, et une seule, de commentaires). On ne fixe pas l'emplacement du fichier dans l'onglet 'Chemins' : cette information sera externalisée.

Sauvegarder !

Etape 2 : Le processus de lecture du fichier des unités de développement

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Process', puis 'ReadSequentialDataFileProcess'. Apparaît la fenêtre de choix des processus de ce type, avec le processus LectureCroissanceHerbe déjà créé..
- Cliquer le bouton 'Créer'. Dans la fenêtre de modification, indiquer le nom de classe LectureUnitesDeveloppementBle

- Dans l'onglet 'Descripteurs', on indique que le processus est spécialisé dans la lecture d'un fichier de symbole de classe FICHIER_UDEV_BLE, et que ce processus va modifier l'état d'une entité de classe BLE Enfin, on tient compte de la périodicité des enregistrements du fichier (journalière) en donnant la valeur 24 au 'pas' du processus.

2.1 Initialisation de la lecture du fichier. Dans l'onglet 'Méthodes', en cochant la case 'Nom automatique' dans la ligne du champ 'SetInitialize', puis en cliquant sur le bouton 'Créer', on ouvre une fenêtre dans laquelle on programme l'initialisation de la lecture du fichier.

- Ouvrir le fichier ressources_FCT.txt, et sélectionner le § AV_E2.1
- Remplacer le contenu de la fenêtre d'édition par ce §. On remarque que seule la partie finale est en fait modifiée : celle qui calcule le nombre de lignes à sauter depuis la date du premier enregistrement.
- Sauvegarder le fichier et quitter l'éditeur

2.2 Changement de l'état du système. Dans l'onglet 'Méthodes', en cochant la case 'Nom automatique' dans la ligne du champ 'SetPostRead', puis en cliquant sur le bouton 'Créer', on ouvre une fenêtre dans laquelle on programme la manière de modifier l'état du système en conséquence de la lecture.

- Dans le fichier ressources_FCT.txt, sélectionner le § AV_E2.2

- Remplacer le contenu de la fenêtre d'édition par ce §. L'instance de la classe de symbole BLE (annoncé dans le champ ProcessedEntityClassId de l'onglet 'Descripteurs') est obtenu par un couple de lignes généré par Solfege qui renverra, au moment de l'exécution, l'instance qu'on aura désignée lors de la création du processus avant son insertion dans l'agenda de l'événement qui le gère (dans le fichier sim1.str). Pour le reste, on ne fait que récupérer la valeur du champ utile (les unités de développement accumulées) pour l'affecter au descripteur QuantiteUnitesDev de l'instance de blé.

- Sauvegarder le fichier et quitter l'éditeur

- Terminer la spécification de ce processus en cliquant sur le bouton 'Valider'

Sauvegarder ! Et générer le code C++ par 'Tout le code'.

Etape 3 : Vérification du codage correct de la lecture des unités de développement

3.1 Installation du fichier. Un fichier UDEV_BLE.txt accompagne ce document. Le copier dans le même répertoire IN_FILES que le fichier de la croissance nette de l'herbe.

3.2 Dans l'agenda des choses à faire, placement de cette lecture de fichier.

- Dans le répertoire des données (SIBEMOL_TUTORIEL_1.0), ouvrir le fichier sim1.str si ce n'est déjà fait.

- Coller en fin de fichier le § AV_E3.2 extrait de ressources_SIM.txt. On y désigne le fichier à lire, et on précise que ce processus doit être lancé à 6 heures du matin du premier jour de la période simulée. Enfin, on désigne l'instance de blé qui sera touchée par le processus.

3.3 Installation d'un moyen de trace de la lecture

- Dans le menu 'Génération' de Solfege, choisir 'Edition des fichiers source C++'. Editer le fichier UserContinuousProcess.cc. En fin de la fonction 'lectureUnitesDeveloppementBle_postReadProcess_Body', mettre en jeu les deux lignes de trace suivantes en enlevant les deux slashs initiaux :

```
int now = ...;
printf(...);...
```

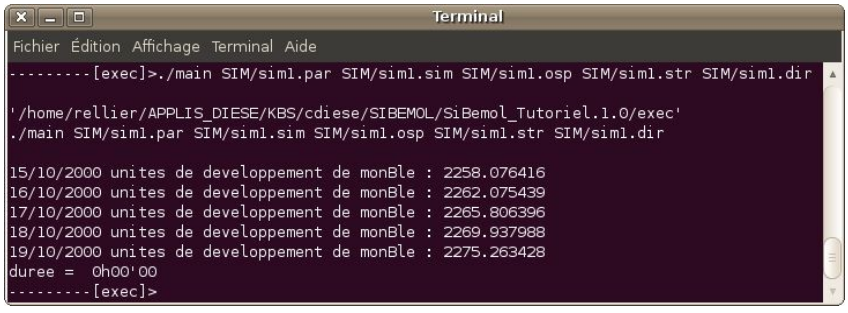
- Sauvegarder cette modification, mais ne pas quitter l'éditeur.

3.4 Excécution de la simulation.

- Dans le répertoire d'exécution (cf. Acte III, Etape 4), lancer les commandes classiques de compilation (si le code a bien été généré à l'Etape 2) et d'exécution :

```
>make std
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir
```

Ce qui doit provoquer l'affichage suivant dans le même terminal :⏭.



3.5 Désinstallation de la trace de la lecture

- Dans la fenêtre d'édition, encore ouverte, du fichier UserContinuousProcess.cc, inhiber les deux lignes de trace précédemment mises en jeu, en remettant les deux slashs initiaux:

```
//int now = ...;
//printf(...);...
```

- Sauvegarder cette modification, et quitter l'éditeur.

ACTE VI : Le développement du blé, vu comme un processus interne au système

Pour un large ensemble de cultures, le développement phénologique peut être vu comme une composante de leur âge. D'autres composantes parfois rencontrées sont le nombre de jours, ou encore le nombre de degrés*jour, depuis le semis ou depuis la levée.

Considérant que cette vision peut être développée sans référence à une culture particulière., d'une part, et que, d'autre part, chaque étude sur ces cultures peut requérir une vision particulière de l'âge, on propose que ces descripteurs ne soient pas attachés aux différentes classes de cultures, ni même à une classe-mère de laquelle les sous-classes hériteraient les descripteurs communs. Au contraire, une culture possède un unique descripteur AttributAge, dont la valeur est une instance de la classe AgeCulture. L'entité AgeCulture détient les informations considérées partagées par toutes les cultures, et différentes sous-classes, par exemple AgeBle, peuvent particulariser le modèle de l'âge.

Pour modéliser le développement, plus précisément le cheminement du premier stade vers le dernier, on codera un processus. On désignera une instance de Ble comme l'entité touchée par ce processus, mais ce sera l'instance d'AgeBle, valeur de son descripteur AttributAge qui sera en fait modifié au cours du temps. On indique brièvement ici que ce processus, à chacun de ses 'pas', se demandera si le prochain stade du blé est ou non atteint, en testant un prédicat propre à chaque stade et qui le définit.

Etape 1 : Installation de la connaissance partagée

L'âge d'une culture est ici réduit au stade phénonogique atteint. Avec deux formes pour cette connaissance : un symbole tel que APRES_SEMIS, FLORAISON, etc, et un nombre entier, indice du stade dans la suite des stades. Les descripteurs à créer sont donc TableauStades (dont la valeur sera un ensemble de stades) et un descripteur à valeur entière et à sémantique d'indice. On sait qu'un tel descripteur est prédéfini dans DIESE et déjà attaché à la classe-mère de toutes les entités : CurrentElementIndex.

1.1 Un descripteur de l'âge : le tableau des stades

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Descriptor', puis 'Constant', cliquer sur 'Créer'.

- Taper TableauStades dans le champ 'Name'. Choisir ENTITY_TAB pour le champ 'Type'. Ajouter un commentaire en références à la classe StadePheno qu'on envisage de créer (par exemple : tableau des entites 'stade pheno').

- Dans l'onglet 'Domaine', on voudrait spécifier qu'un élément de valeur ne peut être qu'une instance de StadePheno. Or un clic sur le bouton '▾' fait apparaître une liste sans le symbole STADE_PHENO, puisque la classe n'a pas encore été créée. Dans cette situation, taper STADE_PHENO en lieu et place de la valeur proposée par défaut. Attention ! Une compilation du code C++ échouerait, le mot 'STADE_PHENO' n'étant pas encore été défini. Cliquer sur 'Ajout'. ⏩

- Cliquer sur 'Valider'.

1.2 Un descripteur complémentaire de l'âge : la culture "agée"

Créer le descripteur suivant, avec les propriétés mentionnées (les commentaires sont entre guillemets). Ainsi, à partir d'une instance d'âge, on pourra pointer facilement sur la culture qui en est dotée.

- AttributCulture ; Constant ; P_ENTITY ; "attribut d'un age : la culture qui a cet age" ; domaine : CULTURE

1.3 Une entité, donc : l'âge d'une culture

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity', cliquer sur 'Créer'.

- Taper AgeCulture dans le champ 'Name'. Faire hériter de BD_ENTITY.

- Ajouter les descripteurs propres constants AttributCulture et TableauStades, précédemment créés.

- Valider

1.4 Quelques descripteurs des stades en général

Créer les descripteurs suivants, avec les propriétés mentionnées (les commentaires sont entre guillemets) :

- AttributNom ; Constant ; STRING ; "nom d'une entité (stade, etc.)"

- AttributAge ; Constant ; P_ENTITY ; "structure encapsulant les ages d'un individu" ; domaine : AGE_CULTURE

- JourJulienQuandStadeAtteint ; Constant ; INT ; "memoire du jour julien d'atteinte du stade" ; domaine : [0 :366]

Pour attribuer un domaine à un descripteur numérique (ici entier), il faut d'abord choisir explicitement le type (ici, ne pas se contenter de laisser la valeur par défaut proposée : INT, mais la confirmer par le bouton de choix). Ensuite, dans l'onglet 'Domaine', écrire les deux bornes (incluses par convention) dans les deux champs dédiés, avant de cliquer sur 'Ajout'. ⏩

1.5 La classe-mère des entités 'stades phénoloqiques'

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity', cliquer sur 'Créer'.

- Taper StadePheno dans le champ 'Name'. Faire hériter de BD_ENTITY.

- Ajouter les descripteurs propres constants AttributNom, AttributAge et JourJulienQuandStadeAtteint, déjà créés.

- Valider

Etape 2 : Installation de la connaissance sur le blé

Les stades phénologiques retenus pour le blé sont : AVANT_SEMIS, APRES_SEMIS, EPI_1CM, GONFLEMENT, FLORAISON (en utilisant d'ores et déjà les symboles des classes qu'on va créer). On décide de les définir à l'aide de deux descripteurs constants : la quantité minimale d'unités de développement à accumuler pour les atteindre, et l'écart minimal sur la quantité d'unités de développement avec le stade précédent. Une autre caractéristique sera la quantité réelle d'unités accumulées lors de l'atteinte du stade.

On a présenté en introduction de cet Acte le processus d'avancée de l'âge, en évoquant un prédicat attaché aux stades. Le rôle de ce prédicat, pour un stade donné, est de renvoyer la valeur booléenne 'vrai' si et seulement si, au moment de son invocation, l'atteinte de ce stade est avérée. On comprend que le contenu de ce prédicat est propre à chaque stade. Par contre, l'existence de ce prédicat est commune à tous les stades. Dans Solfege, on lui donnera le statut de 'méthode' (booléenne), et on attachera cette méthode au stade, comme on lui attache aussi des descripteurs.

2.1 Création des descripteurs propres aux stades du blé

Créer les descripteurs suivants, avec les propriétés mentionnées (les commentaires sont entre guillemets) :

- SeuilUnitDevPourAtteinte ; Constant ; FLOAT ; "qte unites dev pour atteindre le stade"

- DureeStadeEnUnitDev ; Constant ; FLOAT ; "unites dev. a accumuler depuis stade precedent"

- UnitDevReelAuStade ; Variable ; FLOAT ; "unites dev. accumulees quand stade atteint"

2.2 Création du prédicat de test d'atteinte des stades

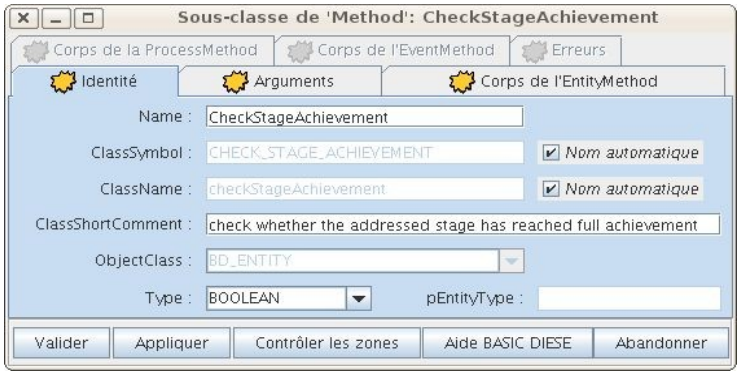
Dans le menu 'Développement', cliquer sur l'item 'Méthodes non prédéfinies'. Apparaît la fenêtre de choix des méthodes, encore vide. Cliquer le bouton 'Créer'. Apparaît une fenêtre de modification.

- Dans le champ 'Name', taper CheckStageAchievement.

- Ecrire un commentaire, tel que "check wether the addressed stage has reached full achievement".

- Dans la liste de la ligne 'ObjectClass', choisir BD_ENTITY (la méthode sera attachée à une entité).

- Dans la liste de la ligne 'Type', choisir BOOLEAN ⏩



- Valider

2.3 Création de l'entité 'stade phéno du blé'

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity', cliquer sur 'Créer'.

- Taper StadePhenoBle dans le champ 'Name'. Faire hériter de STADE_PHENO.

- Ajouter les descripteurs propres constants SeuilUnitDevPourAtteinte et DureeStadeEnUnitDev, et le descripteur variable UnitDevReelAuStade, déjà créés.

- Dans l'onglet 'Méthodes propres', lequel propose une liste de méthodes candidates réduite à la seule encore créée, cliquer sur 'Ajout'. On note qu'aucun contenu n'est donné à ce prédicat à ce niveau de la hiérarchie des classes : ce sont les classes-filles, correspondant à des stades particuliers, qui porteront cette connaissance.

- Valider

2.4 Création de l'entité 'âge du blé'

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity', cliquer sur 'Créer'.

- Taper AgeBle dans le champ 'Name'. Faire hériter de AGE_CULTURE

Cette entité n'a pas de descripteurs particuliers, hormis ceux hérités de la classe-mère. Sa constitution spécifique doit donc apparaître ailleurs. Cela va être dans une propriété fondamentale de cette classe (mais de toutes les classes aussi) : son constructeur. Le constructeur d'une classe est une procédure exécutée en réponse à l'invocation , dans le code C++, de l'opérateur 'new' (par exemple dans l'instruction 'AgeBle* pAge = new AgeBle();'). C'est dans ce constructeur qu'on va exprimer quels stades figurent dans la série des stades du blé.

- Toujours dan l'onglet 'Identité', cliquer sur le bouton, 'Coder' dans la ligne du 'Constructor'. Une fenêtre d'édition apparaît, dans laquelle il faut remplacer les lignes générées par Solfege par le § AVI_E2.4 copié du fichier ressources_FCT.txt. On y repère 5 instruction 'push_back' qui ajoutent 5 stades au tableau qui est la valeur du descripteur de symbole TABLEAU_STADES. Les classes pour ces 5 stades (StadeAvantSemis, etc.) ne sont pas encore créées : une compilation du code lancée maintenant échouerait.

- Sauvegarder le contenu du constructeur, quitter l'éditeur, valider la création de AgeBle (bouton 'Valider').

Sauvegarder la base !

2.5 Création des cinq classes de stade du blé

2.5.1 Le stade "avant semis"

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity', cliquer sur 'Créer'.

- Taper StadeAvantSemis dans le champ 'Name'. Faire hériter de STADE_PHENO_BLE. Puis cliquer sur 'Valider'.

Note : contrairement à l'annonce faite en 2.3, on ne donne pas un corps à la méthode CheckStageAchievement pour cette classe-fille de StadePhenoBle. Simplement parce que c'est le premier de la série, qu'à la création d'un blé, son âge sera ce stade-là, et que la première atteinte de stade testée le sera pour le stade suivant.

2.5.2 Le stade "après semis"

- Cliquer à nouveau sur 'Créer'. Taper StadeApresSemis dans le champ 'Name'. Faire hériter de STADE_PHENO_BLE.

- Dans l'onglet 'Méthodes héritées' (puisque la méthode a été définie au niveau supérieur), cocher la case 'Nom automatique, puis cliquer sur le bouton 'Coder'. Apparaît une fenêtre d'édition. En remplacement du code proposé, y coller le § AVI_E2.5.2 copié du fichier ressources_FCT.txt. On y lit, en commentaire, pourquoi ce prédicat doit toujours renvoyer 'vrai'.

- Sauvegarder le contenu de la méthode, quitter l'éditeur, puis cliquer sur 'Valider'.

2.5.3 Le stade "épi 1 cm"

- Cliquer à nouveau sur 'Créer'. Taper StadeEpi1cm dans le champ 'Name'. Faire hériter de STADE_PHENO_BLE.

- Dans l'onglet 'Méthodes héritées', cocher la case 'Nom automatique, puis cliquer sur le bouton 'Coder'. Apparaît une fenêtre d'édition. En remplacement du code proposé, y coller le § AVI_E2.5.3 copié du fichier ressources_FCT.txt. Brièvement expliqué, on renvoie 'vrai' si la quantité d'UDev accumulée par le blé est au moins égale à ce qu'il avait accumulé au stade précédent augmenté de la "durée" de ce stade en UDev.

- Sauvegarder le contenu de la méthode, quitter l'éditeur, puis cliquer sur 'Valider'.

2.5.4 Le stade "gonflement"

- Cliquer à nouveau sur 'Créer'. Taper Gonflement (*sic*) dans le champ 'Name'. Faire hériter de STADE_PHENO_BLE.

- Dans l'onglet 'Méthodes héritées', cocher la case 'Nom automatique, puis cliquer sur le bouton 'Coder'. Apparaît une fenêtre d'édition. En remplacement du code proposé, y coller le § AVI_E2.5.4 copié du fichier ressources_FCT.txt. Le code est identique à celui pour le stade Epi1cm.

- Sauvegarder le contenu de la méthode, quitter l'éditeur, puis cliquer sur 'Valider'.

Attention ! Le terme Gonflement est en rupture par rapport à notre nomenclature. Mieux vaudrait StadeGonflement. Il faut renommer cette entité. Pour ce faire :

- Dans la fenêtre de choix des entités, sélectionner Gonflement, puis cliquer sur 'Renommer'. Une fenêtre apparaît. ⏩

Dans son champ 'Name', remplacer Gonflement par StadeGonflement, puis cliquer sur le bouton 'Renommer'. Attendre la fin de l'opération, qui peut prendre quelques secondes dans une basse assez grande, parce que tout le code est revisité. Noter que le nom de la classe a changé dans la fenêtre de choix.

2.5.5 Le stade "floraison"

- Cliquer à nouveau sur 'Créer'. Taper StadeFloraison dans le champ 'Name'. Faire hériter de STADE_PHENO_BLE.

- Dans l'onglet 'Méthodes héritées', cocher la case 'Nom automatique, puis cliquer sur le bouton 'Coder'. Apparaît une fenêtre d'édition. En remplacement du code proposé, y coller le § AVI_E2.5.5 copié du fichier ressources_FCT.txt. Le code est identique à celui pour les stades Epi1cm et Gonflement.

- Sauvegarder le contenu de la méthode, quitter l'éditeur, puis cliquer sur 'Valider'.

Sauvegarder la base !

2.6 Installer les valeurs numériques caractéristiques des stades

On a remarqué que les prédicats de trois stades (StadeEpi1cm, StadeGonflement, StadeFloraison) exploitaient une des caractéristiques des stades, la "durée" exprimée en nombre d'UDev depuis l'atteinte du stade précédent. Dans l'état actuel de la base, les prédicats en question ne trouveront comme valeur que la valeur par défaut des descripteurs flottants : 0.0. Pour attribuer une juste valeur aux stades, on va augmenter le contenu des constructeurs, puisque ladite valeur devra être utilisée lors de chaque création d'instance.

2.6.1 Le stade "épi 1 cm"

- Dans la fenêtre de choix des entités, choisir StadeEpi1cm, puis cliquer sur 'Modifier'

- Cliquer sur le bouton 'Modifier' en regard du mot 'Constructor', dans la fenêtre de modification. S'ouvre une fenêtre d'édition du constructeur. On peut y voir l'assignation, spécifiée plus haut (2.5.3) d'un corps spécifique à la méthode CheckStageAchievement. Parce que cette ligne a été générée par Solfege à partir de la spécification, elle apparaît au-dessus d'une ligne séparatrice d'avec un espace réservé à une extension du code non générable automatiquement. C'est dans cet espace qu'on va librement affecter une valeur aux descripteurs qui définissent le stade.
A cette fin, copier juste au-dessous de cette ligne débutant par "// *** DEBUT DU CODE SPECIFIQUE" le § AVI_E2.6.1 trouvé dans ressources_FCT.txt.

- Sauvegarder le contenu de ce constructeur, quitter l'éditeur, puis cliquer sur 'Valider'.

2.6.2 Le stade "gonflement"

- Dans la fenêtre de choix des entités, choisir StadeGonflement, puis cliquer sur 'Modifier'

- Cliquer sur le bouton 'Modifier' en regard du mot 'Constructor', dans la fenêtre de modification. S'ouvre une fenêtre d'édition du constructeur.
Copier juste au-dessous de la ligne débutant par "// *** DEBUT DU CODE SPECIFIQUE" le § AVI_E2.6.2 trouvé dans ressources_FCT.txt.

- Sauvegarder le contenu de ce constructeur, quitter l'éditeur, puis cliquer sur 'Valider'.

2.6.3 Le stade "floraison"

- Dans la fenêtre de choix des entités, choisir StadeFloraison, puis cliquer sur 'Modifier'

- Cliquer sur le bouton 'Modifier' en regard du mot 'Constructor', dans la fenêtre de modification. S'ouvre une fenêtre d'édition du constructeur.
Copier juste au-dessous de la ligne débutant par "// *** DEBUT DU CODE SPECIFIQUE" le § AVI_E2.6.3 trouvé dans ressources_FCT.txt.

- Sauvegarder le contenu de ce constructeur, quitter l'éditeur, puis cliquer sur 'Valider'.

Sauvegarder la base !

2.6.4 Les valeurs numériques ... enfin !

On a remarque que les constructeurs affectaient des valeurs avec des instructions telles que :

float seuilAtteinte = **GetRealParameterValue**("stadesBle", "seuilUnitDevPourEpi1cm");

Le service prédéfini GetRealParameterValue exploite une déclaration externalisée dans un des fichiers d'entrée de la simulation, en termes de "paramètres du système" : le fichier sim1.par encore vide (lire une remarque dans l'Etape 4 de l'Acte III). Il faut donc commencer à le remplir.

- Dans le répertoire des données (SIBEMOL_TUTORIEL_1.0), ouvrir le fichier sim1.par.

- Coller dans fichier le § AVI_E2.6.4 extrait de ressources_SIM.txt. On y retrouve les lignes qui définissent les valeurs des paramètres. Remarquer, pour chacun des paramètres, la stricte égalité entre le couple d'identifiants dans sim1.par et le couple d'arguments du service GetRealParameterValue.

2.7 Dire enfin que le blé a un âge

- Dans la fenêtre de choix des entités, choisir Ble, puis cliquer sur 'Modifier'.

- Dans l'onglet 'Descripteurs propres' de la fenêtre de modification ainsi ouverte, choisir AttributAge parmi les descripteurs constants, puis cliquer sur 'Ajout'.

- Dans l'onglet 'Identité', cliquer sur le bouton 'Modifier' du constructeur. Dans la fenêtre d'édition ainsi ouverte, et juste en dessous de la ligne indiquant le début du code spécifique, ajouter le § AVI_E2.7 du fichier ressources_FCT.txt. On remarque que, en plus de valuer le nouveau descripteur AttributAge avec une instance de AgeBle, on installe la relation réciproque en disant que le blé en construction est la culture dont AttributAge est de descripteur.

- Sauvegarder le contenu de ce constructeur, quitter l'éditeur, puis cliquer sur 'Valider' pour valider ce complément de spécification de la classe Ble.

Etape 3 : Le processus de développement, proprement dit

Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Process', puis 'ContinuousProcess'. Apparaît la fenêtre de choix des processus continus, encore vide. Cliquer sur 'Créer'.

- Dans la fenêtre de modification, onglet 'Identité', taper ProcessusDeveloppementBle dans le champ 'Name'.

- Dans l'onglet 'Descripteurs', taper BLE dans le champ 'ProcessedEntityClassId' (voir l'introduction de cet Acte VI). Donner la valeur 24 (c'est-à-dire en réalité un jour) au 'pas (champ 'Step').

- Dans l'onglet 'Méthodes', cocher la case 'Nom automatique' dans les lignes 'SetInitialize' et 'SetGoOneStepForward'.

- Cliquer le bouton 'Coder' de la ligne 'SetInitialize' pour ouvrir la fenêtre d'édition de cette méthode. Y coller le § AVI_E3.1 du fichier ressources_FCT.txt. On constate que le stade de symbole AVANT_SEMIS est bien donné comme valeur (initiale, donc) à l'âge de l'entité touchée : le blé.

Attention ! L'instruction qui établit la valeur initiale est :

```
pAgeBle = SetIntVarValue(CURRENT_ELEMENT_INDEX, AVANT_SEMIS);
```

Cette écriture provoquerait une erreur de compilation, sans une précaution préalable. L'identifiant AVANT_SEMIS (ne pas confondre avec le symbole de classe STADE_AVANT_SEMIS), très "parlant" ici, n'est pas une variable locale au

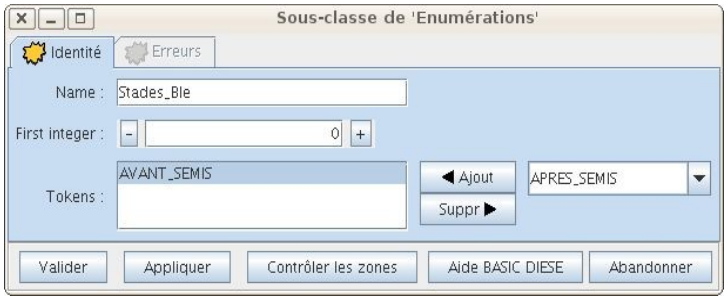
bloc, et on n'a pas encore créé de variable globale de ce nom (par le menu 'Développement', item 'Variables globales'[5]). Il est donc inconnu, et le compilateur émettrait un message du type "AVANT_SEMIS was not declared in this scope". En réalité, le codage de cette fonction d'initialisation du développement aurait dû être précédé par la déclaration des identifiants tels que AVANT_SEMIS, APRES_SEMIS, etc. Et parce que la méthode SetIntVarValue réclame un second argument de type entier, il nous aurait fallu associer un nombre entier à ces mots. L'outil C++ des "énumérations" est fait pour ça, et on va l'utiliser maintenant, avant toute compilation de cette initialisation :

- Dans le menu 'Développement', choisir l'item 'Enumérations'.

- Dans la fenêtre de choix encore vide, cliquer sur le bouton 'Créer'.

- Dans le champ 'Name', taper Stades_Ble, et répéter l'action suivante pour chaque mot dans AVANT_SEMIS, APRES_SEMIS, EPI_1CM, GONFLEMENT, FLORAISON, en respectant cet ordre et l'orthographe :

- Dans le champ à droite de la ligne 'Tokens' taper le mot, puis cliquer sur 'Ajout'. L'écran ci-après est capté juste avant l'ajout du mot APRES_SEMIS. ⏩



- Une fois les cinq mot ajoutés, cliquer sur 'Valider.

- Sauvegarder le contenu de la méthode d'initialisation, quitter l'éditeur de cette méthode.

- Toujours dans la fenêtre de modification du processus, onglet 'Méthodes', cliquer le bouton 'Coder' de la ligne 'SetGoOneStepForward' pour ouvrir une fenêtre d'édition. Y coller le § AVI_E3.2 du fichier ressources_FCT.txt. Brièvement commentée, cette fonction tente de repérer (chaque jour, donc) un éventuel changement de stade, en invoquant la méthode CheckStageAchievement (son symbole est CHECK_STAGE_ACHIEVEMENT). On comprend ici qu'en fonction du stade courant, ce n'est pas le même corps de méthode qui sera invoqué : cette unique procédure d'avancée de l'âge du blé est donc générale. Ensuite, et si le prédicat a renvoyé 'vrai', le reste de la procédure met à jour les variables d'état appropriées, notamment l'indice du stade courant, la valeur réelle des UDev accumulées jusu'au passage de stade, et le jour auquel ce stade a été atteint.

Attention encore ! Cette méthode contient une instruction de trace optionnelle :

```
if(gTraceStadesBle) { ...; TraceDateFromClock(...); printf(...);};
```

TraceDateFromClock est une fonction de la bibliothèque et printf est dans la librairie C++. Seul l'identifiant gTraceStadesBle est inconnu, et provoquerait une erreur de compilation.
Ce qu'on souhaite programmer ici, c'est l'activation/désactivation de cette trace par la présence/absence d'une option dans la ligne de commande. Pratiquement, on voudrait que la ligne suivante provoque la trace de l'atteinte des stades :

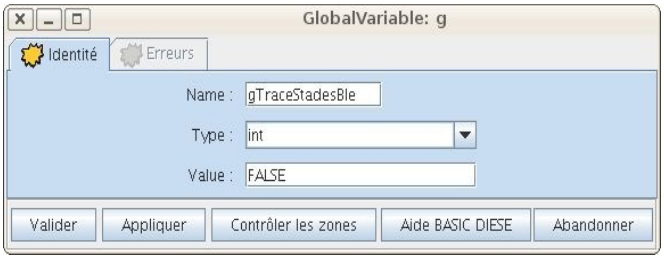
```
./.main sim1.par sim1.sim sim1.osp sim1.str.sim1.dir -sb
```

Pour que la présence de l'option '-sb' ait un effet, il faut que la fonction qui interprète la ligne de commande (celle qui déjà reconnaît les noms des fichiers d'entrée pour les lire) soit aussi informée de la présence possible du '-sb'. Cette fonction (ParseMainArgument) est prédéfinie dans DIESE, et il suffit maintenant de la compléter ainsi :

- Dans le menu 'Développement', item 'Function', choisir 'Interprétation ligne de commande (main)'.

- Dans la fenêtre d'édition qui s'est ouverte, remplacer le schéma de code proposé par Solfege, par le § AVI_E3.3 du fichier ressources_FCT.txt. On remarque qu'on prévoit d'ores et déjà de disposer d'une trace optionnelle de la réalisation des opérations culturales -avec l'option '-o').

- Dans le menu 'Développement', choisir l'item 'Variables globales', et cliquer sur 'Créer'. Dans la fenêtre ainsi ouverte, et dans son champ 'Name', compléter pour obtenir gTraceStadesBle, l'identifiant utilisé dans la fonction d'interprétation. Choisir 'int' dans le menu 'Type', et taper FALSE dans le champ 'Value' pour spécifier qu'en l'absence de l'option '-sb', la trace ne se fera pas. Cliquer sur le bouton 'Valider'. ⏩



[5] Une variable globale est installée dans une partie de la mémoire accessible de n'importe quel module du code de l'application.

- Dès maintenant, réaliser la déclaration de la variable globale gTraceOperations, de type 'int', et de valeur par défaut FALSE, également.

- Sauvegarder la fonction d'interprétation de la ligne de commande et quitter l'éditeur.

- Sauvegarder le contenu de la méthode d'avancée du développement, quitter l'éditeur, puis cliquer sur 'Valider'. pour valider l'ensemble de la définition du processus.

Sauvegarder la base !

Générer le code C++ par le menu 'Génération', item 'Tous les sources'.

Etape 4 : Le test de la modélisation du processus de développement

Avant de réaliser cette vérification du bon développement de cette partie de la base de connaissances, on résume ici le principe du modèle de développement :

- un **processus** assure, sur un blé donné, un enchaînement de stades, à partir du stade initial "avant semis".

- ce processus accède à et manipule un **tableau de stades**, valeur d'un descripteur (TableauStades) d'une entité AgeBle, elle-même valeur de l'**âge** (descripteur AgeAttribut) du blé :

Ble –> AgeBle –> TableauStades –> {AvantSemis ApresSemis Epi1cm Gonflement Floraison}

- une transition de stade intervient lorsque le stade suivant le stade courant est atteint, ce qui est statué par un prédicat propre au stade.

- le prédicat propre au stade est librement écrit par le développeur, mais dans notre cas, il est instruit par la durée du stade à atteindre, en termes d'unités de développement depuis le précédent stade.

Puisque la clé de la simulation du développement est un processus, il reste à l'encapsuler dans un événement, puis à placer cet événement dans l'agenda de la simulation. Pour cela :

- Dans le répertoire des données (SIBEMOL_TUTORIEL_1.0), ouvrir le fichier sim1.str si ce n'est déjà fait.

- Coller en fin de fichier le § AVI_E4 extrait de ressources_SIM.txt. On y désigne le fichier à lire, et on précise que ce processus doit être lancé à 6 heures du matin du premier jour de la période simulée (t=6, ce qui donne à l'événement une priorité suffisante -on le constatera plus loin-). Enfin, on désigne l'instance d'herbe qui sera touchée par le processus.

- Toujours dans SIBEMOL_TUTORIEL_1.0), ouvrir le fichier sim1.sim si ce n'est déjà fait. Modifier la date de fin de la simulation, pour espérer constater l'atteinte de tous les stades du blé :

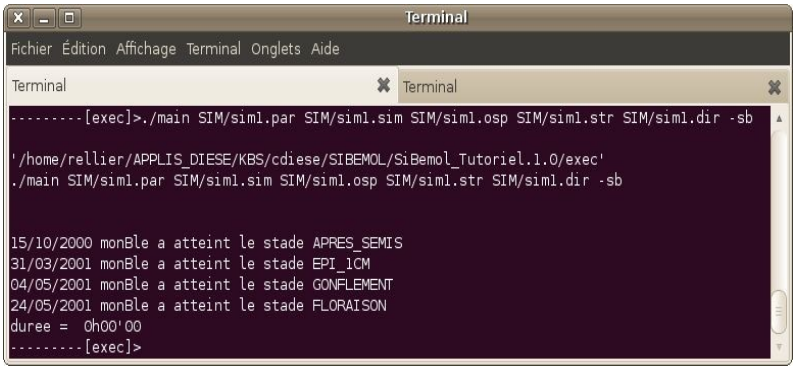
```
//DATE_FIN 20 OCT 2001;
DATE_FIN 2 SEP 2001;
```

- **Compiler la base de connaissances** par le moyen choisi (la commande "make std" dans le terminal d'exécution, ou bien le menu 'Génération', item 'Compilation', choix ' ... pour mode commande').

- Lancer l'exécution par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb
```

Ce qui doit provoquer l'affichage suivant dans le même terminal :⏭



ACTE VII : Les ressources en jeu dans la conduite du système

Les ressources explicitement citées dans l'énoncé du problème sont les unités de main d'œuvre (MO). On va leur donner naturellement le rôle du sujet (le "performer" selon l'ontologie) des activités agricoles modélisées.

Au-delà de ça, nous introduisons dans le problème des "ressources propres" pour certaines opérations culturales. Une telle ressource est spécifiée pour l'opération indépendamment de sa future insertion dans une activité primitive. Cette indépendance est assurée par une convention : la ressource propre d'une opération (impliquée dans une activité) est requise (pour réaliser son effet) pour chaque unité de puissance du "performer" de l'activité. Pour illustration, où que ce soit et quel que soit l'acteur, semer un blé requiert un semoir tracté comme ressource propre. Si deux unités de MO (supposées de puissance 1) sont affectées à l'activité, deux semoirs tractés seront requis. Si un seul semoir est disponible, une seule unité de MO pourra intervenir.

Nous posons les réquisitions suivantes pour les **ressources propres d'opérations** :

- opération culturales du blé, hormis la récolte : un outil tracté (OT ; on ne précise pas quel type d'outil)
- récolte du blé : une moissonneuse-batteuse (MB) et une remorque tractée (RT). Puisque ces deux outils sont pilotés, on comprend qu'un couple de deux unités de MO sera nécessaire à la récolte. Puisqu'on va décider de laisser à chaque unité de MO sa puissance par défaut de 1, ce "performer" couplé aurait une puissance de 2. Il faudra donc surcharger cette valeur avec la valeur 1 pour qu'un seul couple {MB RT} soit affecté au couple de deux unités de MO.
- coupe de la luzerne : un outil tracté (on ne précise pas quel type d'outil)
- complémentation troupeau : un outil tracté (on ne précise pas quel type d'outil)

Pour **la main d'œuvre**, l'énoncé stipule la présence de deux agents. On étend ici à "au moins deux agents" :

- au moins un agent non compétent(s) sur l'atelier "vaches", qui ne peu(ven)t donc intervenir que sur le blé et la luzerne
- un autre agent qui peut intervenir sur le blé, la luzerne et l'atelier "vache"

Une activité primitive réalise une opération par un **"performer"**. Les spécifications en sont les suivantes :

- complémentation troupeau : une unité de MO 'compétente vaches'
- semis du blé : deux unités de MO si possible, une seule si l'autre est allouée à la complémentation, ou si un seul outil
- récolte du blé : un couple de deux unités de MO si disponibles, personne si moins de deux unités sont disponibles
- autres opérations sur blé : une unité de MO quelconque
- coupe luzerne : deux unités de MO si possible, une seule si l'autre est allouée à la complémentation, ou si un seul outil

On rappelle que ces chiffres sont aussi contraints par la réelle présence au travail de la MO (connaissance développée dans une étape ultérieure), et par la disponibilité des ressources propres d'opérations que doit utiliser la MO.

Etape 1 : Les classes de main d'œuvre

1.1 Pour développer les classes d'agents, on constate d'abord que tous les agents sont compétents sur les cultures, et que un parmi eux a la capacité d'intervenir sur les vaches. On pourrait donc créer une classe d'**agents** 'compétents cultures', dont toutes les unités seraient des instances ; et une autre classe qui en serait une spécialisation : la classe des **agents** 'compétents vaches'. 'Agent compétent cultures' ne signifie pas 'agent compétent *seulement en* cultures', et que 'agent compétent vaches' signifie, à cause de la relation d'héritage, 'agent compétent *aussi en* vaches',

Cependant, et en anticipation d'une manipulation justifiée plus loin dans cet Acte, on va créer deux sous-classes de la classe-mère Agent : une pour les agents 'compétents vaches' et une autre pour ceux qui ne le sont pas. La classe-mère n'aura donc pas d'instances directes. Dans la manipulation visée, on aura en effet besoin d'avoir réparti les unités de MO dans des "classes-sœurs".

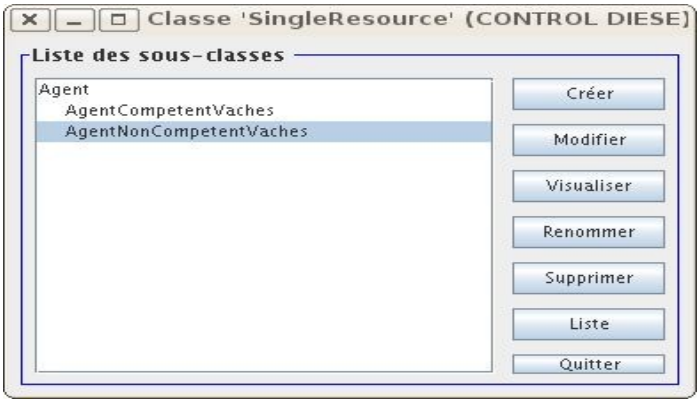
- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Resource', puis 'SingleResource'. Apparaît la fenêtre de choix des ressources simples, encore vide. Cliquer le bouton 'Créer'

- Dans la fenêtre de modification, indiquer le nom de classe Agent. On précise la classe-mère, SinglePerformer, de symbole SINGLE_PERFORMER (prédéfinie dans DIESE).

- Dans la même fenêtre de choix, cliquer à nouveau sur 'Créer'. Dans la fenêtre de modification, indiquer le nom de classe AgentCompetentVaches. On précise la classe-mère, de symbole AGENT.
Attention ! Lors de la frappe du mot AgentCompetentVaches (précisément, après la frappe du premier 't'), une alerte apparaît : le nom de classe (Agent) est déjà pris. Dépasser cette alerte par le bouton 'Ok', puis continuer la frappe.

- Dans la fenêtre de choix, cliquer à nouveau sur 'Créer'. puis indiquer le nom de classe AgentNonCompetentVaches, avec la même classe-mère, de symbole AGENT.

- Dans la fenêtre de choix, cliquer sur le bouton 'Arbre', pour marquer visuellement la hiérarchie entre les deux classes. Le bouton change d'étiquette : cliquer sur 'Liste' remet les classes "à plat". ⏩



1.2 Pour créer **le couple de MO de récolte**, dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Resource', puis 'AggregatedResource'. Apparaît la fenêtre de choix des ressources agrégées, encore vide.

- Dans cette fenêtre, cliquer sur 'Créer'. Dans la fenêtre de modification, indiquer UnitePersonnelRecolte comme nom de classe. On précise la classe-mère, de symbole HOMOGENEOUS_LABOR_TEAM (prédéfinie dans DIESE).

- Une ressource homogène ayant des éléments de la même classe (et non des composants hétérogènes), dans le menu à droite dans la ligne 'ElementClassId', choisir AGENT, puis cliquer sur le bouton 'Ajout'. Ainsi toutes les instances, directes ou indirectes (*via* la classe-fille) pourront intégrer l'équipe de récolte.

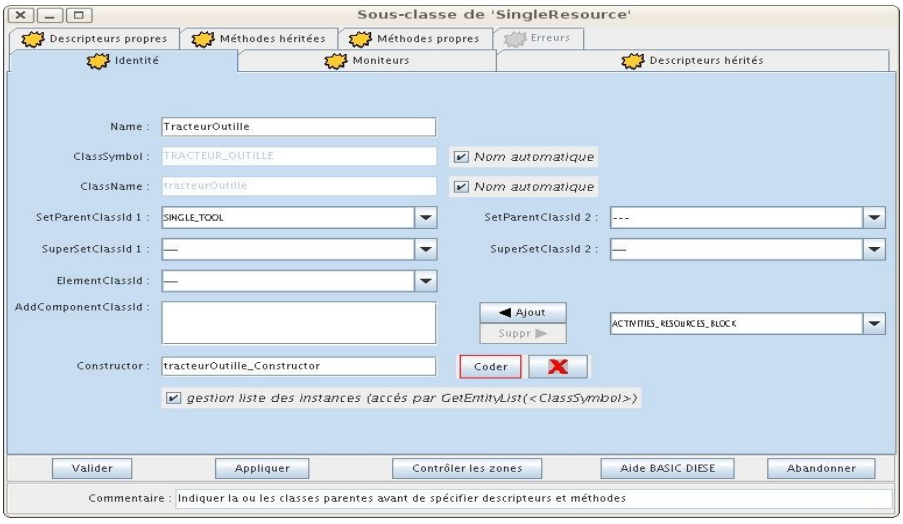
- Dans l'onglet 'Descripteurs hérités', donner la valeur FALSE au descripteur constant IsPermanent. Et donner la valeur 1.0 au descripteur variable Power, conformément à l'intention exprimée dans l'introduction de cet Acte.

Etape 2 : Les classes de ressources propres d'opération

Il y a des **ressources simples** (atomiques) : un tracteur (implicitement muni d'un outil), une remorque (implicitement attachée à un tracteur non muni d'un outil), une moissonneuse-batteuse, et enfin les unités de MO.

Et il y a des **ressources agrégées** : une unité de matériel de récolte (composé d'une moissonneuse-batteuse et d'une remorque tractée), un couple de MO de récolte (dont les éléments seront deux agents quelconques).

2.1 Pour créer la classe des **outils tractés**, dans la fenêtre de choix des ressources simples, cliquer le bouton 'Créer'. Dans la fenêtre de modification , indiquer le nom de classe TracteurOutille, et préciser que la classe-mère a le symbole SINGLE_TOOL (prédéfinie dans DIESE). ⏩



- Valider cette spécification par le bouton 'Valider'.

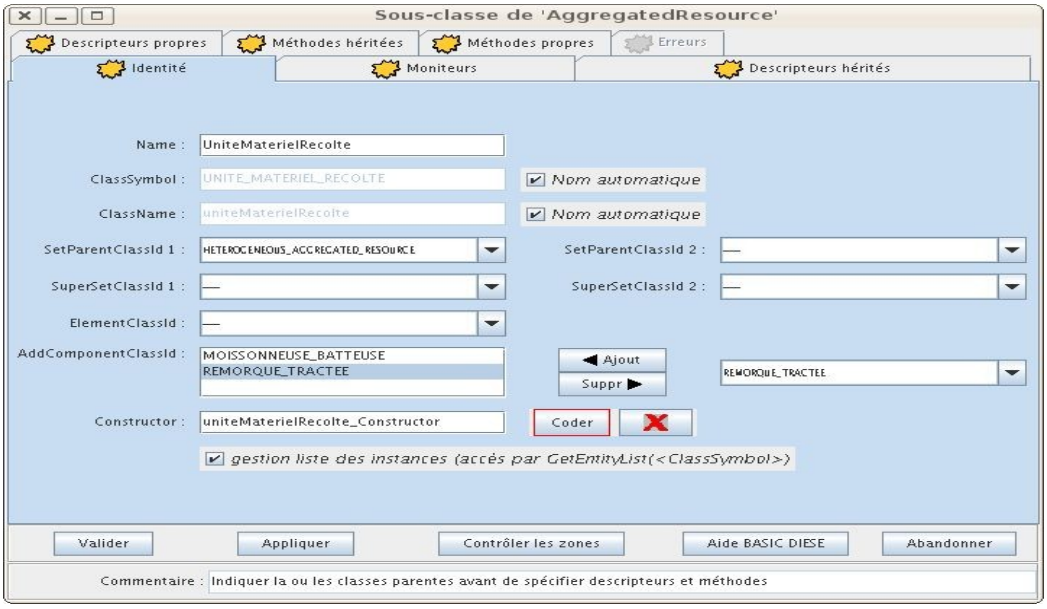
- Procéder de la même manière pour les classes RemorqueTractee et MoissonneuseBatteuse (SINGLE_TOOL, aussi).

2.2 Pour créer **l'unité de matériel de récolte**, dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Resource', puis 'AggregatedResource'. Cliquer le bouton 'Créer' de la fenêtre de choix.

- Dans la fenêtre de modification , indiquer le nom de classe UniteMaterielRecolte. On précise la classe-mère, de symbole HETEROGENEOUS_AGGREGATED_RESOURCE (prédéfinie dans DIESE).

- Pour déclarer le composant 'moissonneuse batteuse', dans le menu à droite dans la ligne 'AddComponentClassId', choisir MOISSONNEUSE_BATTEUSE, puis cliquer sur le bouton 'Ajout'.

- Pour le composant 'remorque', choisir le symbole REMORQUE_TRACTEE, puis cliquer sur le bouton 'Ajout'. ⏩



Etape 3 : L'expression des réquisitions de ressources par les opérations et les activités

C'est dans la définition-même des opérations et activités que sont exprimées ces réquisitions. L'expression utilise les classes de ressources précédemment développées, et ajoute des précisions sur les modalités de leur usage.

On rappelle seulement ici que, de manière générale, une spécification de ressource, c'est (i) la donnée, explicite ou fonctionnelle, d'une liste d'instances de la ressource, (ii) une manière d'en choisir un sous-ensemble (éventuellement total), et (iii) une indication, si besoin, de la taille du sous-ensemble. Si la liste d'instances est simplement spécifiée par un symbole de classe, ce sont ses instances qui constituent la liste. Si on utilise plutôt une 'spécification d'ensemble d'entités' (au sens de l'ontologie), la liste résulte de l'"expansion" de cette spécification.

En reformulant les données de l'introduction de cet Acte, on obtient les tableaux suivants, pour les ressources propres et le 'performer', successivement :

ressources propres d'opération	ensemble d'entités	mode de choix	taille
opération culturales du blé, hormis la récolte	TRACTEUR_OUTILLE	ANYONE	1, PROPORTIONAL
récolter le blé	spécification : "les associations MB+RT"	ANYONE	1, PROPORTIONAL
couper la luzerne	TRACTEUR_OUTILLE	ANYONE	1, PROPORTIONAL
apporter le fourrage	TRACTEUR_OUTILLE	ANYONE	1, PROPORTIONAL

'performer' d'activité	ensemble d'entités	mode de choix	taille
semis du blé	AGENT	ANYONE ou MAX	2 (si ANYONE)
apport azote, désherbage, traitements	AGENT	SETS	1
récolte du blé	spécification : "les couples d'AGENT"	DISJ	
couper la luzerne	AGENT	ANYONE ou MAX	2 (si ANYONE)
apporter le fourrage	AGENT_COMPETENT_VACHES	ANYONE	1

L'alternative ANYONE vs. MAX pour le semis du blé et la coupe de luzerne anticipe qu'on va observer la différence entre les deux options sur le déroulé des actions. Avec ANYONE, les deux unités devront être disponibles. Avec MAX, on mobilisera autant d'unités que disponibles (éventuellement moins de deux).

Par la modalité DISJ, pour la récolte du blé, tous les couples possibles sont des alternatives que la procédure d'allocation va essayer successivement. S'il y a 3 agents (leur compétence n'importe pas ici) 3 couples seront candidats, et la procédure gardera le premier couple disponible (et n'examinera pas les autres).

Par la modalité SETS, pour d'autres activités sur le blé, on tentera l'allocation d'une unité de MO de chaque sous-classe terminale d'Agent : un agent 'compétent vaches' (n'importe lequel s'il y en a plusieurs) et un agent 'non compétent vaches'. Remarquer qu'avec la modalité DISJ, dans la situation avec 3 agents, il y aurait eu 3 candidats. Noter qu'on trouve ici la justification à la création des deux classes-sœurs dans l'étape 1.1.

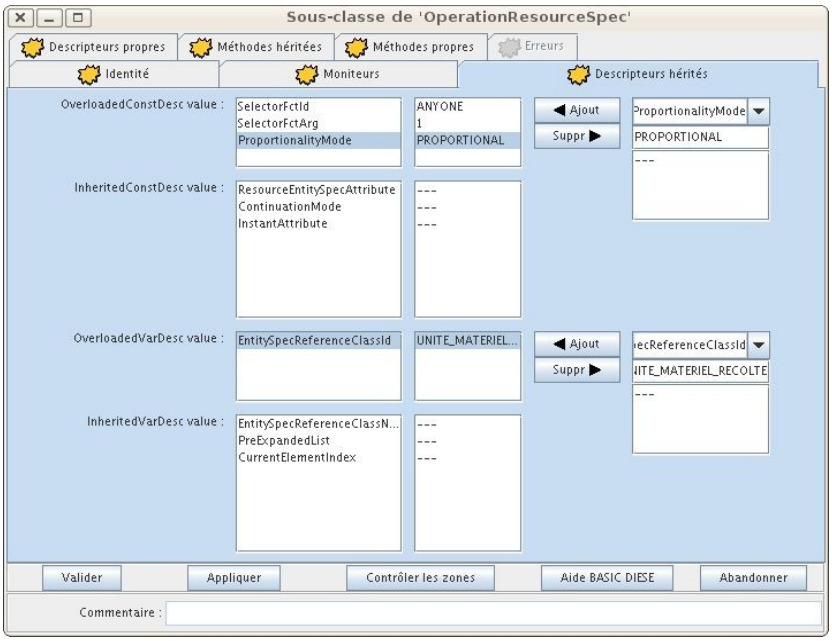
3.1 Spécification de la **réquisition du couple {MB, RT}** pour la récolte du blé

-Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Specification', puis 'OperationResourceSpec'. Cliquer le bouton 'Créer' de la fenêtre de choix, encore vide.

- Dans la fenêtre de modification , indiquer le nom de classe RequisitionUnitaireMaterielRecolte. On précise la seule classe-mère possible à ce stade, de symbole OPERATION_RESOURCE_SPECIFICATION (prédéfinie dans DIESE).

- Dans l'onglet 'Descripteurs hérités' (implicitement : de la classe-mère), on repère deux zones : les descripteurs hérités constants en haut, et les variables en bas. Dans le menu à droite de la ligne 'OverloadedVarDesc value', confirmer le choix par défaut 'EntitySpecReferenceClassId'. Puis, dans le champ de valeur juste au-dessous, remplacer '---' par UNITE_MATERIEL_RECOLTE (c'est le symbole de classe de la ressource agrégée développée à l'étape 2.2. Cliquer ensuite sur le bouton 'Ajout'.

- Dans le même onglet, dans le menu à droite de la ligne 'OverloadedConstDesc value', choisir 'SelectorFctId'. Puis, dans le champ de valeur juste au-dessous, remplacer '---' par ANYONE. Cliquer ensuite sur le bouton 'Ajout'. Faire de même pour le descripteur 'SelectorFctArg', avec la valeur 1. Puis pour le descripteur 'ProportionalityMode', avec la valeur PROPORTIONAL. ⏭



On comprend qu'on requiert ainsi une seule (SelectorFctArg), et n'importe quelle (SelectotFctId), instance de UNITE_MATERIEL_RECOLTE (EntitySpecReferenceClassId), mais autant de fois que la valeur entière de la puissance du 'performer' qui sera alloué (PropotionalityMode).

- Valider la définition de cette classe en cliquant sur le bouton 'Valider'.

3.2 Spécification de la **réquisition du tracteur outillé TO**

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Specification', puis 'OperationResourceSpec'. Cliquer le bouton 'Créer' de la fenêtre de choix.

- Dans la fenêtre de modification , indiquer le nom de classe RequisitionTracteurOutille. On précise la classe-mère, de symbole OPERATION_RESOURCE_SPECIFICATION.

- Dans l'onglet 'Descripteurs hérités', et à droite de la ligne 'OverloadedVarDesc value', confirmer le choix pardéfaut 'EntitySpecReferenceClassId'. Puis, dans le champ de valeur juste au-dessous, remplacer '---' par TRACTEUR_OUTILLE (ressource agrégée développée à l'étape 2.1). Cliquer ensuite sur le bouton 'Ajout'.

- Dans le même onglet, à droite de la ligne 'OverloadedConstDesc value', choisir 'SelectorFctId'. Puis remplacer '---' par ANYONE. Cliquer ensuite sur le bouton 'Ajout'. Faire de même pour le descripteur 'SelectorFctArg', avec la valeur 1. Puis pour le descripteur 'ProportionalityMode', avec la valeur PROPORTIONAL.

3.3 Spécification de la **réquisition du personnel de récolte**

Pour cette spécification, on ne va pas désigner les ressources candidates directement par un symbole de classe, mais plutôt par une 'spécification d'ensemble d'entités', une des classes de base de DIESE (EntitySpec), définie dans l'ontologie. Ainsi, on pourra déterminer les couples d'agents de manière dynamique en cours de simulation, et non de manière statique dans le fichier *.str, comme on le fera pour le couple {MB, RT}. Et cela grâce à la fonction d'instanciation qui est une des propriétés de l'EntitySpec.

On va donc en premier lieu développer cette sous-classe d'EntitySpec, puis y faire référence lors du développement de

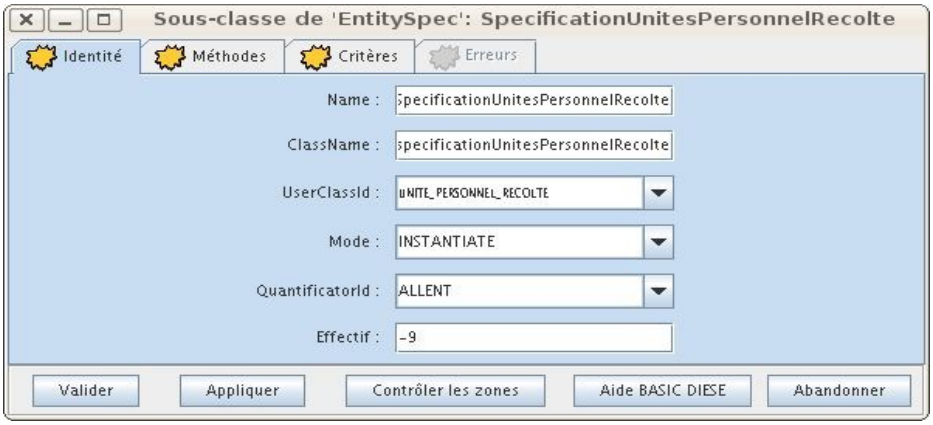
la spécification de la réquisition du personnel de récolte.

3.3.1 Définition de **l'ensemble des équipes de récolte**

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'EntitySpec'. Cliquer sur 'Créer' de la fenêtre de choix, encore vide.

- Dans la fenêtre de modification , indiquer le nom de classe SpecificationUnitesPersonnelRecolte. Bien respecter cette orthographe, en notant le 's' de Unites : on est en train de définir un outil pour créer une liste de couples d'unités de MO.

- Dans ce même onglet 'Identité', et dans le champ 'UserClassId', choisir UNITE_PERSONNEL_RECOLTE. Dans le champ 'Mode', choisir INSTANTIATE (parce qu'on va instancier des entités, dynamiquement). Dans le champ 'QuantificatorId', choisir ALLENT (parce qu'on ne veut éliminer aucune entité ainsi créée), et enfin dans le champ 'Effectif', taper -9 (parce que le nombre d'entités retenues ne peut pas être fixé à l'avance). ⏩



- Dans l'onglet 'Méthodes', cocher le bouton 'Nom automatique' de la ligne 'AssignEntityListInstantiator', puis cliquer sur le bouton 'Créer'. Une fenêtre d'édition apparaît, dont le contenu doit être remplacé par le § AVII_E3.3.1 du fichier ressources_FCT.txt. On remarque l'usage de la fonction prédéfinie SplitToSubsets, qui renvoie une liste de sous-ensembles de son second argument (ici la liste des instances de la classe de symbole AGENT), de la taille de son premier argument (ici 2). On crée ensuite une équipe de récolte (se rappeler que le constructeur lui donne une puissance de 1, voir Etape 1 de cet Acte). On n'oublie pas de désallouer cette double liste en profondeur par depth_delete.

- Sauvegarder cette fonction, puis quitter l'éditeur.

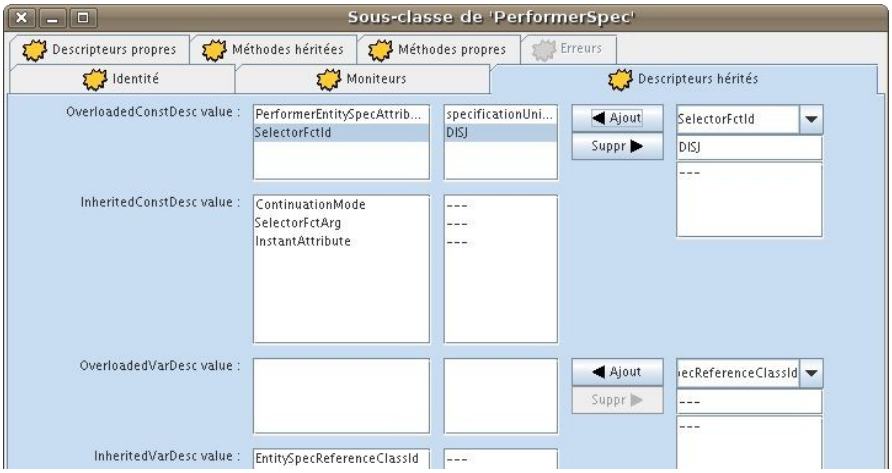
- Valider la définition de cette EntitySpec par le bouton 'Valider'.

3.3.2 Spécification de la **réquisition**, proprement dit

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Specification', puis 'PerformerSpec'. Cliquer le bouton 'Créer' de la fenêtre de choix qui apparaît vide.

- Dans la fenêtre de modification , indiquer le nom de classe RequisitionUnitairePersonnelRecolte. On précise la classe-mère, de symbole PERFORMER_SPECIFICATION.

- Dans l'onglet 'Descripteurs hérités', et dans la partie haute relative aux descripteurs constants, le descripteur proposé pour une éventuelle affectation de valeur "par construction" est PerformerEntitySpecAttribute. Ce descripteur a pour valeur une instance de la classe EntitySpec. Dans notre cas, ça doit être une instance de la sous-classe SpecificationUnitesPersonnelRecolte. Pour prendre en charge cette instanciation et l'affectation de valeur, Solfege requiert du développeur qu'il inscrive dans le champ de la valeur (à droite de la fenêtre, et juste en dessous du nom du descripteur) le "nom de classe" de la classe, ici specificationUnitesPersonnelRecolte (noter le 's' minuscule en tête de chaîne). Taper, donc, cette chaîne puis cliquer sur 'Ajout'. ⏩



- Dans la même zone du même onglet, choisir le descripteur SelectorFctId puis taper DISJ dans le champ de la valeur. Cliquer sur 'Ajout'. On rappelle que ce mot-clé engagera la procédure d'allocation à examiner alternativement toutes les équipes de récolte candidates issues de l'expansion de SpecificationUnitesPersonnelRecolte.

Sauvegarder la base !

Etape 4 : Vérification du code et exécution

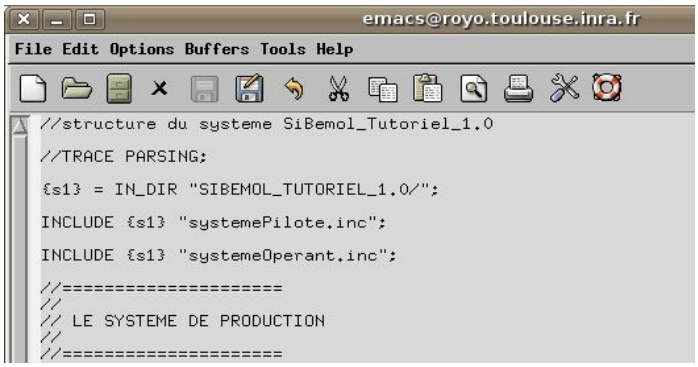
4.1 Générer tout le code source, puis compiler la base de connaissances (se référer au besoin à l'Etape 3 de l'Acte II).

4.2 Ecriture du jeu de données sur les ressources

Une fois développées les classes de ressources, il s'agit ici de décrire le pool de ressources particulier de l'exploitation dont on veut simuler la conduite et le fonctionnement. Cette description est déjà entamée dans le fichier sim1.str (voir les étapes 3.1 ActeIII, 3.2 ActeIV, 3.2 ActeV, 4 ActeVI).

- Dans le répertoire des données (SIBEMOL_TUTORIEL_1.0), ouvrir le fichier sim1.str si ce n'est déjà fait.

- Coller en début de ce fichier, juste après la ligne d'inclusion du fichier systemePilote.inc, le § AVII_E4a extrait de ressources_SIM.txt. Il s'agit de la ligne d'inclusion d'un autre fichier : systemeOperant.inc, dans lequel on va décrire les ressources en jeu dans le cas d'étude. On obtient : ⏩



- Dans le répertoire des données (SIBEMOL_TUTORIEL_1.0), ouvrir une fenêtre d'édition, puis sauvegarder le contenu (encore vide !) dans le fichier systemeOperant.inc. Dans cette fenêtre coller le § AVII_E4b extrait de ressources_SIM.txt. De (presque le) bas en haut, y est déclaré le système opérant composé (c'est l'ontologie qui le dicte) d'un pool multiple de ressources. Dans notre cas, ce pool a deux éléments : un pool de matériel et un pool de personnel. Dans le personnel, il y a deux unités de MO (une troisième, luc, est en stand-by). Par la valeur 1 du 'availabilityStatus', on rend l'unité disponible dès sa création. Le matériel est composé d'un tracteur outillé (un second est en stand-by), d'une remorque et d'une moissonneuse.

Remarque : on instancie, en fin de fichier systemeOperant.inc, la ressource agrégée {MB, RT}. On a en effet spécifié la réquisition du couple {MB, RT} (pour la récolte du blé) à l'aide d'un nom de classe : UNITE_MATERIEL_RECOLTE. Cela implique, pour la procédure d'allocation, de rechercher les instances existantes de la classe (et non d'en instancier de nouvelles dynamiquement, comme pour l'équipe de récolte - voir l'Etape 3.3.1 de cet Acte). Il faut donc, et on le fait ici, instancier au moins une fois la classe UniteMaterielRecolte.

4.3 Lancer l'exécution par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb
```

Ce qui doit provoquer, dans le même terminal, le même affichage que celui obtenu à l'Etape 4 de l'Acte VI. Simplement parce que les ressources créées par la lecture du fichier systemeOperant.inc n'interviennent aucunement dans les processus programmés à ce stade dans l'agenda de la simulation.

Les ressources vont intervenir dès qu'elles seront partie prenante du système de conduite de l'exploitation, c'est-à-dire dès que :
-elles seront attachées aux opérations et aux activités,
-on aura installé dans l'agenda au événement qui provoque la mise en œuvre de ces dernières.

C'est l'objectif des Actes suivants du développement de la base de connaissances et des fichiers de données.

ACTE VIII : Amorce du développement des opérations et les activités primitives

Les opérations, au sens de l'ontologie, sont les opérations culturales sur le blé, les coupes de la luzerne, et l'apport de fourrage au troupeau (on a parlé de complémentation). Les opérations étant en position de composant des activités, et adoptant une démarche d'assemblage, nous allons d'abord développer les classes d'opérations, pour pouvoir y faire référence lors du développement ultérieur des classes d'activités. Dans une autre démarche possible, on développe incomplètement les activités, pour disposer du cadre des concepts les plus englobant, puis on les complète au fur et à mesure du développement des concepts sous-jacents.

Notre liste de noms pour les sous-classes 'application' de la classe prédéfinie Operation est la suivante (en notant que nous allons utiliser des verbes pour nommer les opérations, et des substantifs pour les activités, MoissonBle, par exemple) : SemerBle, DesherberBle, FertiliserBle, TraiterBle, RecolterBle, CouperLuzerne, ApporterFourrage,

On se fixe l'objectif d'installer un embryon du plan de conduite de l'exploitation, suffisant pour rencontrer les principaux outils et les attitudes fréquentes dans la confection de ce plan. On va se focaliser sur le semis du blé, parce qu'il va nous permettre de simuler le processus de développement, non plus par un processus dédié et externalisé (voir les Etapes 3 et 4 de l'Acte VI), mais en réponse à une opération décidée par le plan de conduite.

Dans cet Acte, on définit un élément du plan, l'activité de semis du blé, avec en préalable, la définition de l'opération mise en œuvre : SemerBle. Dans l'Acte suivant, on passera de l'activité primitive à l'activité agrégée qu'est le plan de conduite.

Etape 1 : Un premier développement des opérations : semer le blé

- Pour **SemerBle**, dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Operation'. Dans la fenêtre de choix des opérations encore vide, cliquer sur 'Créer'. Dans la fenêtre de modification, taper SemerBle dans le champ 'Name'.

- Dans le champ SetParentClassId, on s'apprête à choisir TIME_GRANULATED_OPERATION. Selon l'ontologie, la vitesse d'extension de l'effet de l'opération serait ainsi, sans unité, un pourcentage, et l'effet de l'opération serait étendu, à chaque 'pas', de cette valeur de vitesse[6]. Cependant, en anticipant que cette remarque va s'appliquer, par choix supposé bon pour notre problème, à l'ensemble des opérations sur le blé, on projette de créer une autre classe OperationBle, de type TIME_GRANULATED_OPERATION, dont SemerBle, et les autres opérations sur le blé, seront héritières, notamment de ce type. Donc, c'est le mot OPERATION_BLE qu'on tape (puisqu'il le figure pas encore dans la liste ouvrable par le bouton '▼') dans le champ SetParentClassId.

- Valider la définition de SemerBle par le bouton 'Valider'.

- Pour **OperationBle**, cliquer à nouveau sur 'Créer'. Puis taper OperationBle dans le champ 'Name'.

- Dans le champ SetParentClassId, on s'apprête donc à choisir TIME_GRANULATED_OPERATION, comme annoncé ci-dessus. On se rappelle à ce stade qu'on a prévu de disposer d'une trace de la réalisation des opérations culturales, avec l'option '-o' de la ligne de commande (voir l'Etape 3 de l'Act VI). Pour que cette fonctionnalité s'applique à toutes les opérations, on va l'installer (on verra comment dans l'étape suivante) sur une classe qui sera la mère de toutes les autres, dont OperationBle : cette classe, on va l'appeler OperationTracee (qui sera du type TIME_GRANULATED_OPERATION). Donc, c'est le mot OPERATION_TRACEE qu'on tape dans le champ SetParentClassId.

- Dans l'onglet 'Descripteurs hérités', on voudrait stipuler que l'objet opéré par toute opération sur le blé est de la classe Parcelle, *via* une valeur par "construction" du descripteur prédéfini OperatedEntityId de la classe Operation. On constate cependant que ce descripteur ne figure pas dans les menus des zones 'constants' et 'variables'. C'est normal, puisqu'aucun lien d'héritage n'existe encore entre OperationBle et Operation.

- Donc, valider telle quelle la définition de OperationBle par le bouton 'Valider'.

- Puis on définit la classe **OperationTracee** en cliquant à nouveau sur 'Créer'. Puis en tapant OperationTracee dans le champ Name. Dans le champ SetParentClassId, on choisit TIME_GRANULATED_OPERATION, symbole d'une sous-classe de Operation.

- Dans l'onglet 'Descripteurs hérités', donner au descripteur constant Step la valeur 8[7] (la rédaction de sim1.sim, dans le répertoire des données, indique que ce 'pas' est exprimé en heures).

---

[6]Pour les QuantityGranulatedOperation's, l'effet s'étend à chaque 'pas' sur une nouvelle portion d'une grandeur numérique (e.g. surface, effectif) décrivant l'objet opéré. Pour les UnitGranulatedOperation's, l'effet touche à chaque 'pas' de nouveaux éléments de l'objet opéré (e.g. différentes bandes identifiées dans une parcelle).

[7]La valeur 8 a une justification purement technique : elle permet de situer le changement du degré de progression dans le même jour (parce qu'on sait qu'il n'intervient qu'à la fin du 'pas'), alors qu'une valeur de 24, autant naturelle que 8, aurait situé le changement le lendemain. Cette justification est connue par expérience, mais elle peut naître aussi le l'observation attentive des résultats de la simulation.

Etape 2 : Installation de la trace de la progression de l'effet des opérations

Le descripteur prédéfini AchievementDegree de la classe Operation est mis à jour par le moteur de simulation, en fonction du 'pas' et de la vitesse de l'opération, et bien entendu de l'avancée de l'horloge de la simulation. Cette mise à jour n'est accompagnée, toujours dans le moteur, d'aucune trace. Il revient au développeur d'une base de connaissance d'installer cette trace, au besoin, et sous une forme qu'il décide.

Parce que le développeur n'a pas accès au code du moteur de simulation de DIESE, pour y insérer des instructions de trace, la procédure de trace fait nécessairement partie de la base de connaissance. C'est, en principe, une procédure qu'on doit déclencher lors de tout ou partie des changements de valeur du descripteur AchievementDegree. On sait que DIESE propose le schéma du "démon" (ou "moniteur") pour coder ce principe. Un démon est posé sur un descripteur, et possède deux méthodes : l'une (appelons-la ici WhenGet) se déclenche automatiquement lorsqu'on accède à la valeur du descripteur (pour l'exploiter), l'autre (appelons-la ici WhenSet) se déclenche automatiquement lorsqu'on affecte une nouvelle valeur au descripteur (après l'avoir calculée, lue, apprise, etc.). Un démon placé sur AchievementDegree et muni d'une méthode WhenSet fait donc l'affaire pour notre trace du degré de progression. C'est le moteur de simulation qui reconnaît qu'un démon a été placé sur le descripteur, et qui exécute WhenSet lors d'un changement de valeur.

Un écueil à la pose, dans le code de la base de connaissances, d'un démon sur AchievementDegree, c'est qu'un démon est connu pour être déjà posé sur ce descripteur dans DIESE (pour les besoins propres du moteur). Et surtout, qu'il est impossible d'en poser deux ! Autrement dit, nous sommes dans le schéma suivant :

Operation : *desc. natif AchievementDegree : démon A : WhenSet_A*
OperationTracee : *desc. hérité AchievementDegree : démon A (par héritage) + démon B: WhenSet_B* impossible !

La solution est dans le schéma suivant :

Operation : *desc. natif AchievementDegree : démon A : WhenSet_A*
OperationTracee : *desc. hérité AchievementDegree : démon B: WhenSet_B'* possible !

... avec *démon_B* venant remplacer (on dit "surcharger") *démon_A* dans OperationTracee, et *W **5.2.1** henSet_B' cumulant l'effet* de *WhenSet_A* (exigé par le moteur) et l'effet de *WhenSet_B* (la trace désirée).

- Ainsi, dans l'onglet 'Moniteurs' de la fenêtre de modification de OperationTracee, et précisément dans la zone 'SetMonitorToDescriptor', allons-nous ajouter un démon (moniteur) au descripteur AchievementDegree. On constate que ce descripteur figure dans la liste ouverte par le bouton '▾' à droite du bouton 'Ajout', puisqu'hérité de Operation. Par contre aucun moniteur ne figure encore dans le menu situé juste en dessous. Il faut donc préalablement créer ce qui s'appelle plus haut le *démon_B*.

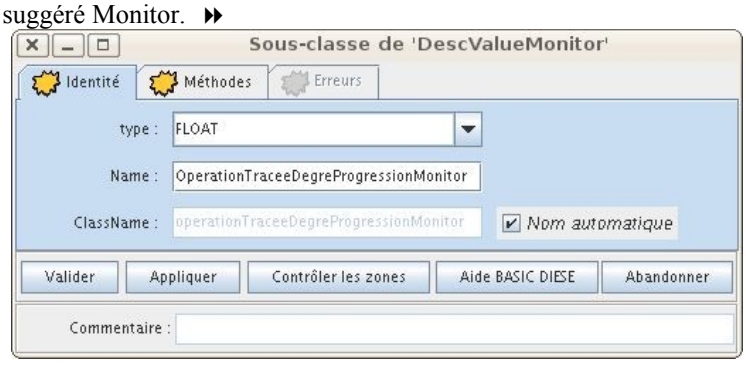
- Valider la description incomplète de OperationTracee, par le bouton 'Valider'.

2.1 Création du moniteur pour le descripteur AchievementDegree

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Monitor', puis 'DescValueMonitor'. Dans la fenêtre de choix des démons, encore vide, cliquer sur 'Créer'.

- Dans la fenêtre de modification, choisir FLOAT dans le menu du champ 'type', parce qu'on sait que le descripteur visé est du type FLOAT.

- Dans le champ 'Name', taper la chaîne "OperationTraceeDegreProgression" (sans les guillemets) avant le suffixe suggéré Monitor. ⏩



- Dans l'onglet 'Méthodes', cocher la case 'Nom automatique' à droite de la ligne 'AssignWhenSetFloatMethod'. Puis cliquer le bouton 'Coder'. Dans la fenêtre d'édition ainsi ouverte, coller le § AVIII_E2.1 du fichier ressources_FCT.txt, en remplacement de l'amorce de code généré par Solfege.

La première tâche de ce code consiste à appeler la fonction operationPercentAchieved_WhenSetFloat (documentée dans CONTROL DIESE). C'est celle qu'on a appelée plus haut *WhenSet_A*, attachée à ce qu'on a appelé *démon_A*. Le reste est la programmation de la trace désirée. On y remarque l'utilisation de services prédéfinis dans DIESE : ExtractOOSet, ExtractPerfSet, qui récupèrent des informations sur l'objet opéré et le 'performer'. On remarque aussi l'usage de la

variable globale gTraceOperations, créée à l'Etape 3 de l'Acte VI, pour permettre l'optionalité de la trace.

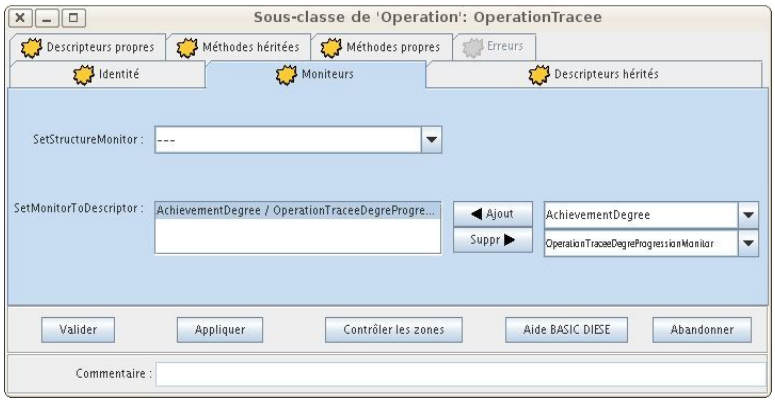
-Sauvegarder ce contenu, puis quitter l'éditeur.

- Valider la définition de ce moniteur par le bouton 'Valider'.

2.2 Pose du moniteur sur le descripteur AchievementDegree

-Rouvrir la fenêtre de modification de OperationTracee (en la sélectionnant dans la fenêtre de choix des classes d'opérations, puis en cliquant 'Modifier').

- Dans l'onglet 'Moniteurs', et dans la zone 'SetMonitorToDescriptor', choisir AchievementDegree dans la liste ouverte par le bouton '▾' à droite du bouton 'Ajout'. Dans le menu situé juste en dessous, choisir le moniteur créé à l'Etape 2.1, puis cliquer sur 'Ajout'. ⏩



Etape 3 : Toujours sur OperationBle : attachement des ressources propres

A l'Etape 3 de l'Acte VII, on a envisagé de généraliser la réquisition d'un tracteur outillé en la déclarant au niveau de la classe-mère OperationBle. Donc :

- Dans la fenêtre de modification de OperationBle, et dans l'onglet 'Descripteurs hérités', zone des descripteurs variables, choisir RequiredResources, et taper requisitionTracteurOutille (noter la minuscule en première lettre), avant de cliquer sur 'Ajout' .

- Saisir la visite de cet onglet pour donner la valeur PARCELLE au descripteur OperatedEntityId, ce qui n'avait pas pu être fait à l'Etape 1de l'Acte VIII. Dans la zone des descripteurs constants, choisir OperatedEntityId, et taper PARCELLE, avant de cliquer sur 'Ajout'

- Valider la définition complétée de l'opération par le bouton 'Valider'.

Etape 4 : Retour sur SemerBle : codage de l'effet de cette opération

- Rouvrir la fenêtre de modification de SemerBle. Dans l'onglet 'Méthodes héritées', cocher la case 'Nom automatique' à droite de la ligne 'StateTransitionProcedure'. Puis cliquer le bouton 'Coder'. Dans la fenêtre d'édition ainsi ouverte, coller le § AVIII_E4 du fichier ressources_FCT.txt, en remplacement de l'amorce de code généré par Solfege.

La StateTransitionProcedure est automatiquement lancée par le moteur de simulation au moment opportun. Pour les opérations de type TIME_GRANULATED_OPERATION, elle code le changement d'état observable lorsque l'opération a eu son plein effet (contrairement à d'autres types, pour lesquels elle code le changement réalisé à chaque 'pas'). Toujours pour le type choisi, cette procédure est exécuter lors du premier 'pas' de progression de l'opération. Lors des pas suivants, ce n'est que le degré d'atteinte du plein effet qui est augmenté.

4.1 Dans notre codage, **le lancement du développement du blé** est réalisé en seconde partie. On instancie la classe ProcessusDeveloppementBle (créée à l'Etape 3 de l'Acte VI), puis on exploite le service prédéfini PostInitAndProceedEvent pour insérer cet événement autogénérable dans l'agenda.

Important ! Il convient dès lors de supprimer du fichier de données sim1.str le paragraphe qui a le même effet. Pour cela, lui donnant le statut de "commentaire" en l'encadrant par les chaînes '/*' et '*/'. ⏩

```
/*  pour test hors init/proceed par operation SemerBle
+ P processusDeveloppementBle pDevBle
     PAS = 24;
     ENTITE_CIBLE = <I><, monBle>;
     INIT_PRCD_EVT
        DATE_OCCUR = 6;
        PRIORITE = 50;
    ;
;
*/
```

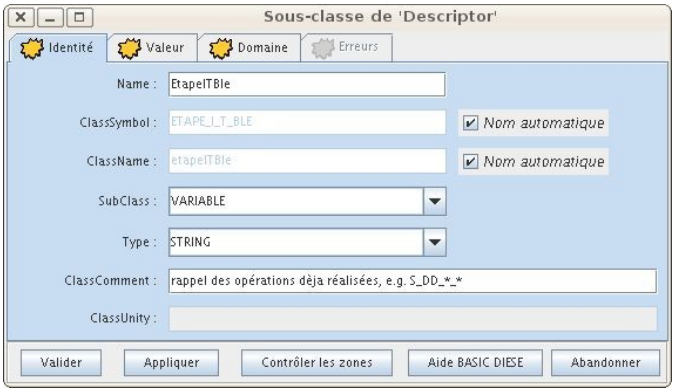
Noter la priorité donnée à l'événement dans le code de StateTransitionProcedure : 10, priorité relativement forte. On participe ici à l'ordonnancement global des priorités (synthétisé dans l'Annexe 3) en décidant que la connaissance de l'état de développement sera connu avant de décider les actions culturales du jour, dont on sait (par l'ontologie) qu'elles ont par défaut une priorité dans le milieu de la gamme [0 99].

4.2 Dans la première partie de StateTransitionProcedure, on a ajouté un élément de trace spécifique des opérations sur le blé : l'affichage d'une chaîne de caractères dont le contenu va progresser de "S_*_*_*_*", dès que le semis sera fait, jusqu'à "S_D_FFF_TT_R" quand on aura fait, par exemple, un désherbage, trois apports d'azote, deux traitements et la récolte. Cet appendice ne sert qu'à illustrer l'usage (relativement fréquent, mais un peu délicat) de la fonction C strtok.

Le code fait usage d'un descripteur de symbole ETAPE_I_T_BLE, sur une instance de la clas **5.2.1** se Culture, dont on se rappelle ne pas l'avoir encore créé, ni bien sûr attaché à la classe. Pour faire cela maintenant, et ne pas risquer une erreur de compilation plus tard :

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Descriptor', puis 'Variable'. Dans la fenêtre de choix, cliquer sur 'Créer'.

- Dans la fenêtre de modification, taper EtapeITBle dans le champ Name, confirmer éventuellement VARIABLE dans le menu 'SubClass', choisir STRING dans le menu 'Type', et taper un commentaire tel que "rappel des opérations dèjà réalisées, e.g. S_DD_*_*". ⏩



- Valider la définition de ce descripteur par le bouton 'Valider'

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity". Dans la fenêtre de choix, sélectionner Culture, puis cliquer sur 'Modifier'.

- Dans la fenêtre de modification, et dans l'onglet 'Descripteurs propres', dans la zone des descripteurs variables, choisir EtapeITBle, puis cliquer sur 'Ajout'.

- Valider la définition complétée de cette entité par le bouton 'Valider'.

4.3 Revenir sur la fenêtre d'édition encore ouverte de la StateTransitionProcedure, pour **enregistrer** son contenu, et quitter l'éditeur.

4.4 Attribution d'une vitesse à la progression de l'effet

- Dans la fenêtre de modification encore ouverte de SemerBle, et dans l'onglet 'Identité', cliquer sur le bouton 'Coder' du champ 'Constructor' (ça peut être devenu le bouton 'Modifier' si la fenêtre a été, pour une raison quelconque, fermée puis rouverte).

- Dans la fenêtre d'édition ainsi ouverte, coller le § AVIII_E4.4 du fichier ressources_FCT.txt, et cela juste après la ligne contenant la chaîne "DEBUT DU CODE SPECIFIQUE" (et juste avant le '};' final). On y constate l'utilisation du service prédéfini GetRealParameterValue, argumenté par deux chaînes de caractères : 'vitesseOperation", et "semerBle". La valeur renvoyée est affectée au descripteur prédéfini TimeSpeed de la classe TimeGranulatedOperation.

En fait, GetRealParameterValue ne renverra une valeur que si le fichier sim1.par contient une ligne contenant en bonne place ces deux chaînes de caractères. Donc, dans le répertoire des données, ouvrir le fichier sim1.par (dont la rédaction a été entamée à l'Etape 2.6.4 de l'Acte VI). Y coller, en sa fin, le § AVIII_E4.4 du fichier ressources_SIM.txt. La valeur écrite 1:10 (égale à 0,1 ou 10%) implique que 10 'pas' de progression seront nécessaires pour réaliser le plein effet du semis, avec un 'performer' de puissance 1. Le plein effet sera atteint en seulement 5 'pas' si deux unités de MO sont affectées au semis.

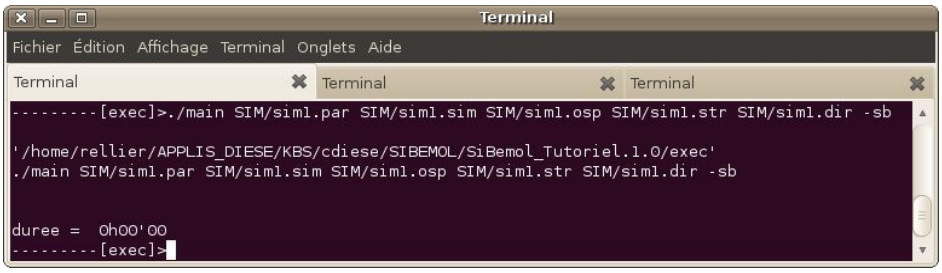
- Valider la définition de l'opération SemerBle par le bouton 'Valider'

Sauvegarder la base ! Générer tout le code source, puis compiler la base de connaissances.

- Vérifier la bonne lecture du fichier des paramètres modifié, en relançant l'exécution :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb
```

Ce qui doit provoquer l'affichage suivant dans le même terminal :



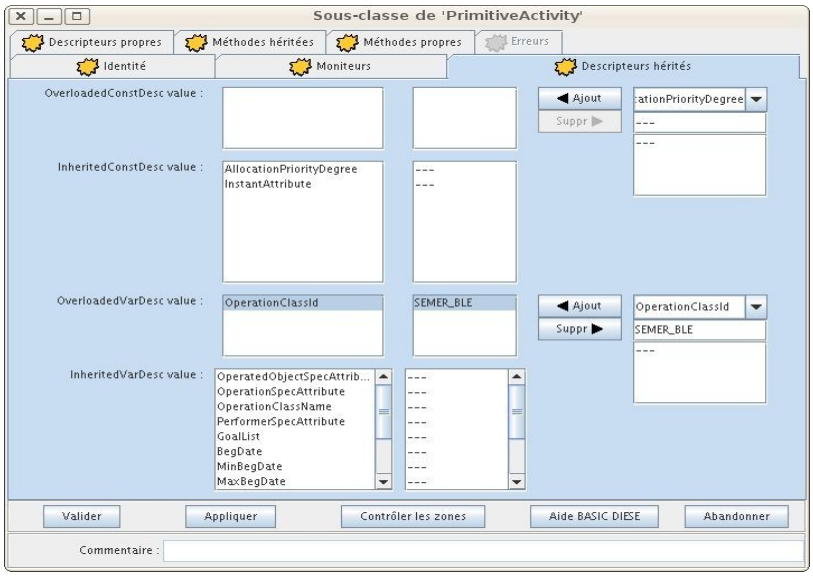
On remarque que la trace de l'apparition des stades du blé a disparu, et que la trace du semis n'apparaît pas. Cela est dû à la suppression, dans sim1.str, de la programmation de l'événement qui lance le développement, et au fait que notre plan de conduite ne contient rien, même pas l'activité de semis du blé. C'est la tâche réalisée dans l'étape suivante.

Etape 5 : Un premier développement des activités : le semis du blé

5.1 L'activité **SemisBle**

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Activity' puis '...Primitive'. Dans la fenêtre de choix des activités primitives encore vide, cliquer sur 'Créer'. Dans la fenêtre de modification, taper SemisBle dans le champ 'Name'. Et choisir PRIMITIVE_ACTIVITY dans le champ SetParentClassId.

- Dans l'onglet 'descripteur hérités', et la zone des descripteurs variables, choisir OperationClassId, puis taper SEMER_BLE comme valeur (dans le champ juste en dessous). C'est donc une instance de la classe SemerBle qui sera l'opération mise en œuvre par cette activité. Cliquer sur 'Ajout'.



Il reste à exprimer ce que sera le 'performer' de l'activité, et son 'objet opéré'.

- Par un choix de modélisation, les activités primitives du blé seront organisées, de manière essentiellement séquentielle, en ce qu'on appelle un "itinéraire technique". **L'objet opéré** par toutes les activités primitives sera le même : **la parcelle de blé**. Un schéma de codage classique consiste à attacher la parcelle de blé à l'itinéraire (plutôt qu'à chacune des primitives, ce qui serait peu commode en cas de changement de parcelle), puis à installer un mécanisme de propagation automatique de cette entité en tant qu'objet opéré des primitives.

- **Quant au 'performer', on choisit d'en déclarer un par défaut**, commun à toutes les activités sur le blé, puis de surcharger cette valeur par une autre pour les activités qui dérogent à cette règle commune. Ce 'performer' par défaut sera **un couple de deux agents quelconques**. On va donc définir une activité virtuelle qui véhiculera cette spécification commune de 'performer', dont les activités réelles (SemisBle, etc.) hériteront, avant de surcharger éventuellement la valeur héritée du 'performer'. Cette activité(primitive) virtuelle, on va l'appeler ActiviteBle.

- Donc, dans l'onglet 'Identité' de SemisBle, taper le symbole ACTIVITE_BLE en remplacement de PRIMITIVE_ACTIVITY. Le taper, parce que l'item ACTIVITE_BLE ne figure pas encore dans le menu des symboles connus dans 'SetParentClassId'.

- Valider cette définition d'activité primitive par 'Valider'. Une compilation du code généré conduirait à une erreur, puisque le symbole ACTIVITE_BLE n'est pas encore connu du compilateur.

5.2 L'activité **ActiviteBle**

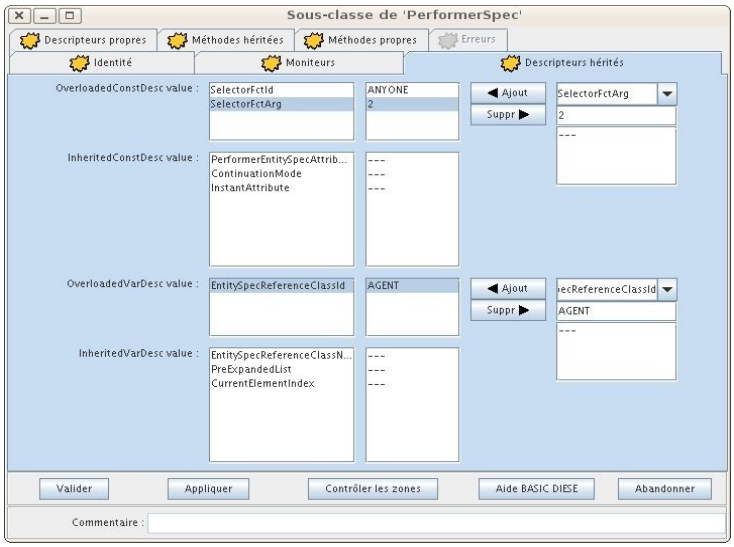
- Dans la fenêtre de choix des activités primitives, cliquer sur 'Créer'. Dans la fenêtre de modification, taper ActiviteBle dans le champ 'Name'. Et choisir PRIMITIVE_ACTIVITY dans le champ 'SetParentClassId'.

- Valider cette première définition de l'activité primitive par 'Valider'.

5.2.1 La définition du **'couple de deux agents'**

-Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Specification', puis 'PerformerSpec'. Dans la fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper DeuxAgents dans le champ 'Name'. Puis choisir PERFORMER_SPECIFICATION dans le champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', et dans la zone des descripteurs variables, donner la valeur AGENT au descripteur EntitySpecReferenceClassId. Dans la zone descripteurs constants, donner la valeur ANYONE au descripteur SelectorFctId, et la valeur 2 au descripteur SelectorFctArg. ⏭



- Valider la définition de cette spécification de 'performer' par le bouton 'Valider'.

5.2.2 L'attachement du 'performer' **'couple de deux agents'** à **ActiviteBle**

- Rouvrir la fenêtre de modification de ActiviteBle. Dans l'onglet 'Descripteur hérités', donner au descripteur PerformerSpecAttribute la valeur "deuxAgents" (sans taper les guillemets) : il s'agit du nom de classe de la classe DeuxAgents, à partir duquel le moteur instanciera cette classe, par une fonction automatiquement générée par Solfege). Cliquer donc sur 'Ajout', après avoir sélectionné le bon descripteur variable, et tapé la chaîne-valeur dans le champ juste en dessous.

- Un affinage du modèle de conduite consiste à installer une variante de cette spécification de 'performer'. On a dit que, par construction, on prendrait exactement deux agents (ANYONE 2). La variante consiste, comme évoqué à l'Etape 3 de l'Acte VII) à prendre 2 agents si possible, mais un seul si 2 est impossible. On décide d'externaliser le choix de cette variante, pour permettre de comparer facilement les déroulés d'actions culturales dans les deux options, sans avoir à modifier la base de connaissances. A cette fin, on va doter le gestionnaire de l'exploitation, instance de la classe prédéfinie Manager, d'un descripteur propre ModaliteDeuxAgents auquel on donnera, dans le fichier externe qui instanciera le gestionnaire, la valeur ANYONE ou alternativement la valeur MAX.

- Valider cette définition complétée de l'activité primitive par 'Valider'.

5.2.3 Qualifier le gestionnaire par la modalité d'allocation de deux agents

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Descriptor', puis 'Constant'. Dans la fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper ModaliteDeuxAgents dans le champ 'Name'. Confirmer les valeurs CONSTANT et INT. Ajouter un commentaire tel que "ANYONE=1, MAX, ALL, DISJ, SETS" (sans les guillemets), pour rappeler que les valeurs possibles sont les symboles de l'énumération (prédéfinie dans CONTROL DIESE) des modes de sélection des ressources candidates lors de l'allocation.

- Valider la définition de ce descripteur par le bouton 'Valider'.

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'System', puis 'Manager'. Dans la fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper MonManager dans le champ 'Name'. Puis choisir MANAGER dans le champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs propres', et dans la zone des descripteurs constants, choisir ModaliteDeuxAgents, puis taper la valeur par construction ANYONE dans le champ juste en dessous, avant de cliquer sur 'Ajout'.

- Valider la définition de ce gestionnaire par le bouton 'Valider'.

5.2.4 Installer le mécanisme de changement de modalité d'allocation de deux agents

- Rouvrir la fenêtre de modification de ActiviteBle. Dans l'onglet 'Identité' alors ouvert, cliquer sur le bouton 'Modifier' de la ligne 'Constructor'. Une fenêtre d'édition s'ouvre, avec le code que Solfege a pu générer à partir des spécifications déjà fournies.

- Coller le § AVIII_E5.2.4 du fichier ressources_FCT.txt, et cela juste après la ligne contenant la chaîne "DEBUT DU CODE SPECIFIQUE" (et juste avant le '};' final). On y remarque le surchargement de la valeur du SelectorFctId de la spécification du 'performer', seulement si, dans le fichier externe, on a donné au descripteur ModalitéDeuxAgents une valeur différente de celle par défaut.

- Remarquer aussi la dernière partie du code ajouté. On crée une instance de OperatedObjectSpecification : c'est une spécification dont le développement identifiera ce qu'on a appelé l'objet opéré' de l'activité, dans l'Etape 5.1 de cet Acte. A ce stade, on ne donne pas d'indication qui permette au développement de désigner un quelconque objet opéré. C'est le mécanisme de propagation automatique posé sur ActivitéBle et évoqué à l'Etape 5.1 qui donnera cette indication.

- Sauvegarder le contenu du constructeur de ActiviteBle, quitter l'éditeur, puis cliquer sur 'Valider'.

Sauvegarder la base !

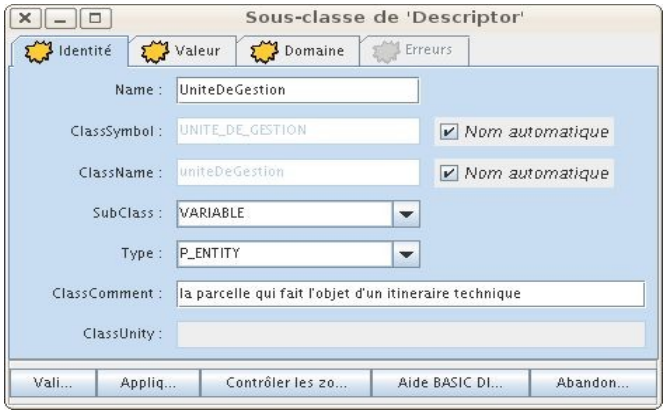
ACTE IX : A partir d'une activité primitive, un embryon de plan de conduite

L'activité de semis est pour nous la première de l'itinéraire technique du blé. Elle sera suivie des activités de désherbage, fertilisation, traitement et récolte. Cela conduit à placer ActivitéBle comme premier élément d'une instance de la classe ActivityBefore, prédéfinie dans CONTROL DIESE pour représenter, manipuler et raisonner sur une séquence de tâches non recouvrantes dans le temps.

Etape 1 : La classe pour l'itinéraire technique du blé

- Dans le menu 'Développement', item 'CONTROL DIESE, choisir 'Acvtivity', puis '... NonPrimitive'. Dans la fenêtre de choix encore vide, cliquer sur 'Créer'. Dans la fenêtre de modification ainsi ouverte, taper ItineraireBle dans le champ 'Name', et choisir ACTIVITY_BEFORE dans le menu de la ligne 'SetParentClassId'.

-Dans l'onglet 'Descripteurs propres', on voudrait ajouter un descripteur UniteDeGestion dont la valeur serait l'objet opéré par toutes les activités primitives de l'itinéraire (voir Etape 4.1 de l'Acte précédent). Puisque ce descripteur n'existe pas encore, dans la zone des descripteurs variables, cliquer sur 'Créer'. Dans la fenêtre de modification ainsi ouverte, taper UniteDeGestion dans le champ 'Name', choisir VARIABLE dans le menu '', choisir P_ENTITY dans le menu 'Type', puis ajouter un commentaire tel que "la parcelle qui fait l'objet d'un itinéraire technique". ⏩



- Valider cette définition de descripteur par 'Valider'.

-Maintenant, dans l'onglet 'Descripteurs propres' de ItineraireBle, ajouter le descripteur UniteDeGestion qui est apparu dans la liste, puis valider cette définition de cette activité agrégée par 'Valider'.

Etape 2 : La déclaration de la séquence d'activités sur blé

Externaliser cette déclaration dans le fichier sim1.str par un paragraphe tel que ci-dessous ?

```
+   I   itineraireBle   monITBle
      +   E  semisBle   monSemisBle ;
      +   E  desherbageBle   monDeherbageBle;
      // etc.
;
```

Internaliser le codage de la séquence dans la procédure qui construira toute instance d'itinéraire ? C'est cette option qui est choisie, considérant que l'étude de variantes dans la constitution de la séquence n'est pas un point à faciliter pour l'utilisateur du simulateur.

- Aussi, dans l'onglet 'Identité', cliquer sur le bouton dont l'étiquette est 'Créer' ou 'Modifier' dans la ligne 'Constructor'. Juste en dessous de la ligne contenant "DEBUT CODE SPECIFIQUE", coller le § AIX_E2 du fichier ressources_FCT.txt. Une partie du code est déactivé, par un encadrement avec '/*' et '*/' : il s'agit des éléments de la séquence non encore connus dans la base de connaissance, et donc la compilation serait en échec. On activera ce code dès que possible.

On constate que, par construction donc, l'itinéraire de conduite de notre blé comprend quatre éléments :
. le semis, un désherbage d'automne, et la récolte : ce sont des activités primitives ;
. en 3ème position, une activité non primitive encapsulant la fertilisation et les traitements : tout cela commencera après la fin du désherbage et avant le début de la récolte. L'organisation de ces travaux sera déclarée ultérieurement.

- Valider la définition de l'itinéraire du blé par le bouton 'Valider'.

Etape 3 : La séquence d'activités sur blé dans le plan d'action global du gestionnaire

Le gestionnaire de l'exploitation fait l'objet de la classe MonManager dans cette base de connaissances (voir l'Etape 5.2.3 de l'Acte VIII). Cette classe hérite de la classe prédéfinie Manager, dont le composant unique est une instance de la classe prédéfinie Strategy. N'ayant à ce stade pas de motif de créer une sous-classe de Strategy spécialisée sur notre domaine, c'est une instance directe de cette classe que nous attacherons à notre instance de MonManager.

La manière de déclarer la stratégie en conformité avec l'ontologie est peu variable d'une application à l'autre. On y retrouve la structuration en "blocs activités-ressources" (un seul dans la situation fréquente où il existe au moins une ressource exploitable par toutes les activités). Un bloc activités-ressources est la mise en service d'un ensemble de ressources pour un plan d'activités. Notre ensemble de ressources a été décrit à l'Etape 4.2 de l'Acte VII. Il reste à décrire le plan d'activités.

Le plan (global) d'activités est la mise en œuvre, en parallèle, de trois ensembles d'opérations, celles sur le blé, celles sur la luzerne et celles pour le troupeau. En suivant l'ontologie, il s'agit d'une conjonction d'activités non primitives. Une est déjà définie : l'itinéraire sur le blé. On définira ultérieurement les autres, et notre conjonction est à ce stade réduite au cas limite, mais admissible, d'un seul élément. On va dédier un fichier séparé à la description de la stratégie de conduite.

- Dans le répertoire ds données, ouvrir une fenêtre d'édition, naturellement vide. Provoquer la création du fichier visé en sauvegardant le contenu dans le fichier systemeDecisionnel.inc.

- Dans le même répertoire, éditer le fichier sim1.str, juste après l'inclusion du fichier systèmeOperant.inc, dans le haut de la fenêtre, coller le § AIX_E3a de ressources_SIM.txt.

- Dans la fenêtre d'édition de systemeDecisionnel.inc, coller le § AIX_E3b de ressources_SIM.txt. On note :
. l'instanciation du gestionnaire, laissant le descripteur ModaliteDeuxAgents avec sa valeur ANYONE par défaut. Remarquer qu'on donne une valeur au descripteur UniteDeGestion, déclenchant ainsi la transmission automatique de la parcelle de blé comme objet opéré des activités primitives (voir les Etapes 5.1de l'Acte VIII et 1 de celui-ci).
. l'instanciation de l'itinéraire sur le blé, puis son attachement comme élément du plan global d'activités, créé à l'étape suivante. C'est la conjonction évoquée plus haut dans cette étape, mais non encore intégrée dans la base de connaissances. Remarquer qu'on donne la valeur 1 au descripteur ReadyToRun. C'est un descripteur prédéfini de la classe-mère Activity. On sait qu'un démon est posé sur ce descripteur, dont la réaction à l'affectation de la valeur 1 provoque le passage de l'activité porteuse du statut SLEEPING au statut WAITING, avec propagation appropriée aux activités-filles. L'absence de ce détail de spécification dans les données de la simulation provoque l'absence totale de mise en œuvre d'opérations pendant la simulation.
. la définition de l'unique bloc 'activités-ressources', incluant le pool de ressources décrit dans l'autre fichier systemeOperant.inc (voir Etape 4.2 de l'Acte VII).
. le dernier paragraphe (avant l'instruction `END INCLUDE;` de retour à la lecture de sim1.str) est l'attachement de la stratégie au gestionnaire crée en haut du fichier. On note alors la syntaxe pour faire référence à une instance déjà créée (`<I><...> ...;`).

Etape 4 : La définition du plan d'action global du gestionnaire

- Dans la fenêtre de chois des activités non primitives, cliquer sur 'Créer'. Dans la fenêtre de modification, taper ConduiteExploitation dans le champ 'Name', puis choisir ACTIVITY_CONJUNCTION dans le menu 'SetParentClassId'.

Aucune autre précision n'est utile à ce stade. La lecture, dans le fichier systemeDecisionnel.inc, d'une directive d'instanciation de cette classe (voir ci-dessus) est maintenant possible.

Sauvegarder la base !

Etape 5 : Insertion du plan d'action dans le système de production

Le plan d'action est dans la stratégie, qui est dans l'instance de MonManager, et ce gestionnaire est un des trois composants du système de production, avec le système opérant et le système piloté. Ce dernier composant a déjà été installé à l'Etape 3.1 de l'Acte III. Il suffit alors de compléter la structure du système de production :

- Dans le fichier sim1.str du répertoire de données, insérer le § AIX_E5 de ressources_SIM.txt dans le paragraphe d'instanciation de la classe ProductionSystem, qui devient : ⏩

```
//=====================
//
// LE SYSTEME DE PRODUCTION
//
//=====================

+ I productionSystem pSP
    <- C <I><, pCS>;
    <- C <I><, pOS>;
    <- C <I><, pSD>;
;
```

Etape 6 : Insertion de la mise en œuvre du plan d'action dans l'agenda de la simulation

Le plan d'action (même partiel à ce stade) est désormais défini (dans la base de connaissances) et instancié dans les fichiers de données externalisées. Dans l'état, il s'agit d'une structure de données statique qui ne peut évoluer (notamment sur les transitions d'état des activités, de WAITING à OPEN -provoquant la mise en œuvre des opérations-, puis CLOSED) si un processus ne lui est pas appliqué. Et si ce processus n'est pas placé sous la gestion d'un événement inséré dans l'agenda.

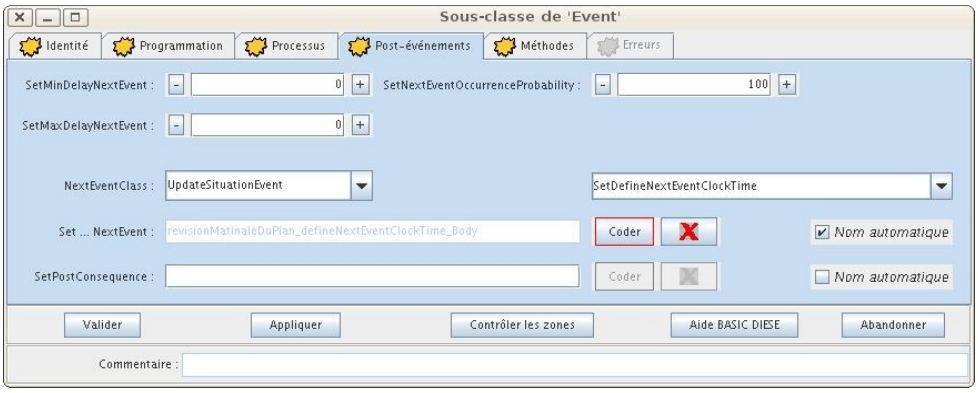
Un tel processus est prédéfini dans CONTROL DIESE (UpdateSituationProcess), et son contenu ontologique ne semble pas devoir être surchargé dans notre application. Quant à l'événement qui gère ce processus, il est aussi prédéfini (UpdateSituationEvent). Sa méthode d'autogénération reprogramme l'événement à un instant futur spécifié par les attributs MinDelayNextEvent et MaxDelayNextEvent. Choisissant que le gestionnaire révisera l'état du plan tous les matins, on pourrait fixer ces deux valeurs à 24 dans le fichier externe. Si le premier événement était positionné, par exemple, à 8h du matin le 15 octobre, les suivants seraient aussi positionnés à 8h jusqu'au passage à l'heure d'hiver, puis à 7h jusqu'au retour de l'heure d'été, puis à 8h, et ainsi de suite. Pour maintenir constante l'heure de révision du plan, on doit donc surcharger la méthode en charge de la reprogrammation, et par conséquent ... :

6.1 Créer une spécialisation de l'événement prédéfini :

- Dans le menu 'Développement', choisir l'item 'Event'. Dans la fenêtre de choix ainsi ouverte, cliquer sur 'Créer'. Dans le fenêtre de modification, taper RevisionMatinaleDuPlan dans le champ 'Name', puis choisir UpdateSituationEvent dans le champ 'ParentClass'.

- Dans l'onglet 'Post-événements', repérer la ligne étiquetée 'NextEventClass'. Dans le menu de gauche, ne rien changer, ou confirmer la valeur UpdateSituationEvent, puisque c'est un événement descendant de cette classe qu'on veut (re)générer automatiquement. Dans le menu de droite, choisir la valeur SetDefineNextEventClockTime, puisque ce qu'on veut surcharger, c'est la manière de définir l'instant d'occurrence de l'événement.

- Dans ce même onglet, et à la ligne 'Set...NextEvent', cocher le bouton 'Nom automatique', puis cliquer sur Créer'. ⏩



- Dans la fenêtre d'édition ainsi ouverte, remplacer l'amorce de code générée par Solfege par le § AIX_E6 du fichier ressources_FCT.txt. On y remarque l'usage des services prédéfinis ClockValueToTm et, réciproquement de StructTmToClockValue : partant la valeur courante de l'horloge, on crée une structure tm de C qui permet, par l'instruction C mktime, de commodément changer le jour sans changer l'heure : on revient ensuite à une valeur d'horloge pour renvoyer la date du nouvel événement. Le prologue et l'épilogue de cette fonction ont été suggérés par Solfege.

- Sauvegarder le contenu de cette méthode, puis valider la définition de l'événement par 'Valider'.

Sauvegarder la base !

6.2 Instancier cet événement dans le jeu de données

- A la fin du fichier sim1.str du répertoire de données, insérer le § AIX_E6.2a de ressources_SIM.txt. On y remarque la ligne suivante :

```
DATE_OCCUR = <pi><"revisionDuPlan" "heureMatin"">;
```

La forme <pi> fait référence à un paramètre entier, lequel doit donc avoir été préalablement déclaré. Il faut donc ...

- ... dans le fichier sim1.par du répertoire de données, insérer le § AIX_E6.2b de ressources_SIM.txt au début de la section 'Conduite'. La valeur de l'heure est fixée à 8.

Etape 7 : Le test de la modélisation du plan de conduite

- **Compiler la base de connaissances** par le moyen choisi (la commande 'make std' dans le terminal d'exécution, ou bien le menu 'Génération', item 'Compilation', choix ' ... pour mode commande').

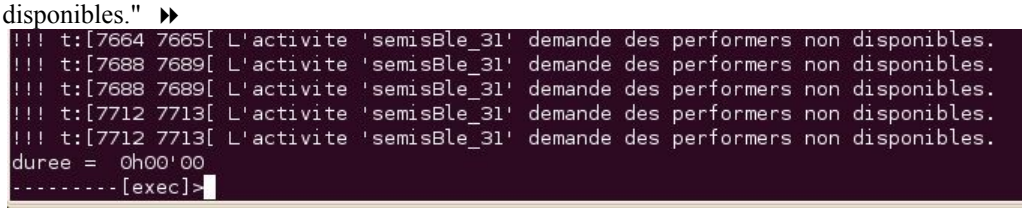
- **Lancer l'exécution** par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb
```

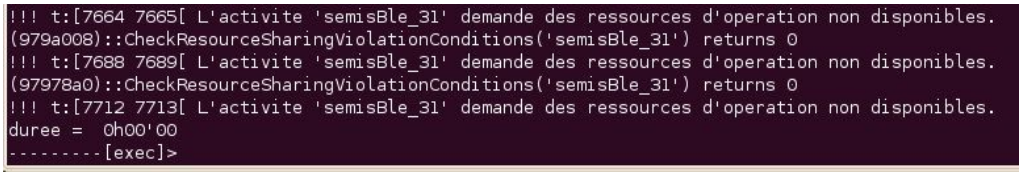
Aucune trace n'apparaît, comme après l'exécution de l'Etape 4 de l'Acte VIII. Parce que c'est le comportement du moteur dans la gestion des ressources qui nous intéresse à ce stade, on relance l'exécution avec une option complémentaire prédéfinie : -RscWarn

```
>./main SIM/sim1.par ........ SIM/sim1.dir -sb -RscWarn
```

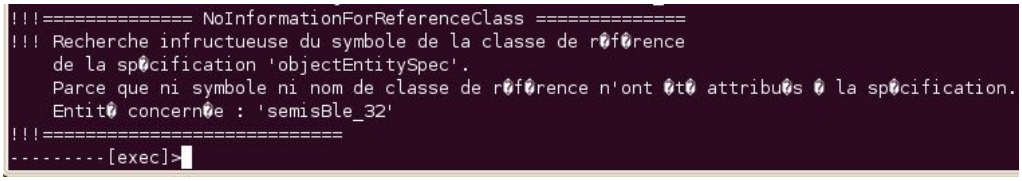
On constate alors une série de messages sur la sortie standard : "L'activité SemisBle demande des performers non disponibles." ⏩



On vérifie donc la spécification de 'performers' pour l'activité SemisBle, héritée de ActivitéBle. On a attaché DeuxAgents (Etape 5.2.4 de l'Acte VIII), et, dans les données (systemeDecisionnel.inc, Etape 3 de l'Acte IX), on n'a pas modifié la modalité ANYONE. La disponibilité de deux agents est donc requise. Le système opérant répond-il à cela, dans systemeOperant.inc ? On constate que non, parce que deux agents, jean et paul sont bien instanciés, mais leur descripteur availabilityStatus est mis à 0. On justifiera lors d'une étape ultérieure cela, mais on va vérifier notre hypothèse en changeant la valeur 0 en 1 pour jean et paul. On relance ensuite la même exécution, et on constate la trace suivante, avec des messages "L'activité SemisBle demande des ressources d'opération non disponibles." : ⏩



On vérifie alors la spécification de 'ressources propres' pour l'opération SemerBle, héritée de OperationBle. On a attaché RequisitionTracteurOutille (Etape 3 de l'Acte VIII), et on se rappelle que cette réquisition est "proportionnelle", au sens que si deux agents sont requis, alors deux tracteurs doivent être disponibles. Ce n'est pas le cas dans notre jeu de données (systemeOperant.inc), puisque la ligne pour un second tracteur (pTO_2) est inhibée. Donc, remettre cette ligne en jeu, en enlevant le '//' initial, puis relancer la même exécution. On obtient une trace qui se termine ainsi :



C'est à nouveau, non plus l'opération, mais l'activité SemisBle qui est en cause, et plus précisément son 'objectEntitySpec', autrement dit sa spécification d'objet opéré, à partir de laquelle le moteur ne parvient pas à identifier l'objet opéré (on a prévu que ce soit la parcelle de blé, voir Etape 5.1 de l'Acte VIII). Cette spécification d'objet opéré a été attachée ActiviteBle dans son constructeur, à l'Etape 5.2.4 de l'Acte VIII .Et on se rappelle qu'on a alors accepté "qu'on ne donne pas [à ce stade] d'indication qui permette ... de désigner un quelconque objet opéré", en prévoyant que ce soit un mécanisme de démon sur le descripteur UniteDeGestion qui fasse le travail (voir Etape 5.1 de l'Acte VIII). On réalise en cet instant que ce démon n'a pas encore été défini ni attaché, et ceci devient une explication vraisemblable (mais pas encore sûre ni complète) de la dernière trace.

Etape 8 : Spécification concise de l'objet opéré par les activités primitives

8.1 Création du moniteur pour le descripteur UniteDeGestion

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Monitor', puis 'DescValueMonitor'. Dans la fenêtre de choix des démons, cliquer sur 'Créer'.

- Dans la fenêtre de modification, choisir P_ENTITY dans le menu du champ 'type', parce qu'on sait que le descripteur visé pointe sur une entité (une parcelle).

- Dans le champ 'Name', taper la chaîne "UniteDeGestion" (sans les guillemets) avant le suffixe suggéré Monitor. ⏩

- Dans l'onglet 'Méthodes', cocher la case 'Nom automatique' à droite de la ligne 'AssignWhenSetEntityMethod'. Puis cliquer le bouton 'Coder'. Dans la fenêtre d'édition ainsi ouverte, coller le § AIX_E8 du fichier ressources_FCT.txt, en remplacement de l'amorce de code généré par Solfege.

On y remarque l'usage du service prédéfini PrimitiveList, qui renvoie la liste des activités primitives dans une activité

agrégée. Pour chaque élément de cette liste (ici les activités culturales du blé : semis, etc.), on récupère la spécification d'objet opérée (on voit ici l'importance de l'avoir instanciée dans le constructeur de ActiviteBle, à l'Etape 5.2.4 de l'Acte VIII). Puis on ajoute la parcelle valeur du descripteur UniteDeGestion (celui sur lequel est posé le démon) à la liste qui est la valeur du descripteur PreExpandedList de la spécification (cette liste n'aura que cet élément).

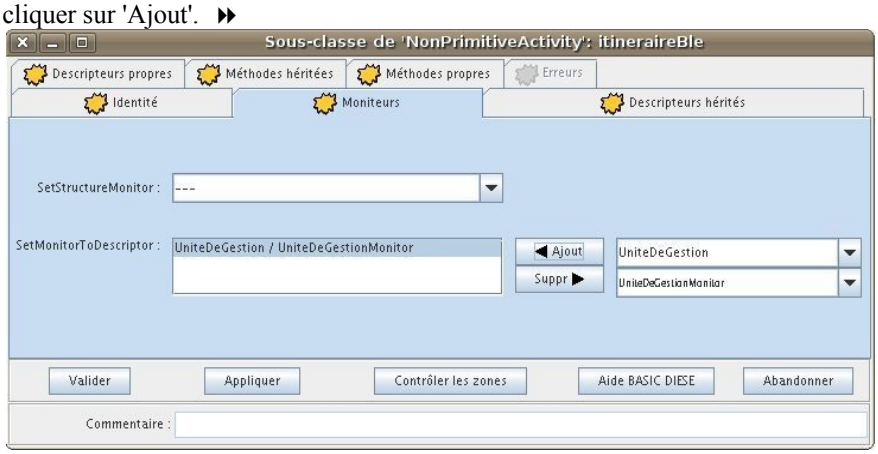
-Sauvegarder ce contenu, puis quitter l'éditeur.

- Valider la définition de ce moniteur par le bouton 'Valider'.

8.2 Pose du moniteur sur le descripteur UniteDeGestion

-Rouvrir la fenêtre de modification de ItineraireBle (en la sélectionnant dans la fenêtre de choix des classes d'activités non primitives, puis en cliquant 'Modifier').

- Dans l'onglet 'Moniteurs', et dans la zone 'SetMonitorToDescriptor', choisir UniteDeGestion dans la liste ouverte par le bouton '▼' à droite du bouton 'Ajout'. Dans le menu situé juste en dessous, choisir le moniteur créé à l'Etape 8.1, puis cliquer sur 'Ajout'. ⏩



- Valider la définition complétée de cette activité par le bouton 'Valider'.

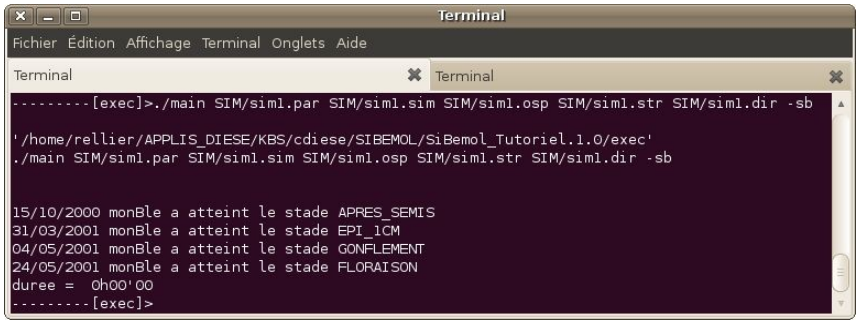
Etape 9 : Nouveau test de la modélisation du plan de conduite

- **Compiler la base de connaissances** par le moyen choisi (la commande "make std" dans le terminal d'exécution, ou bien le menu 'Génération', item 'Compilation', choix ' ... pour mode commande').

- **Lancer l'exécution** par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb
```

On obtient la trace attendue de la progression des stades du blé, identique à celle de l'Etape 4 de l'Acte VI, lorsque le développement était déclenché par un événement externalisé, et non pas comme ici par un événement généré par la procédure de changement d'état de l'opération SemerBle.



ACTE X : Le développement complet des opérations culturales

On développe ici les opérations, en préparation du référencement les opérations dans la définition des activités primitives. On commence arbitrairement par définir sommairement les opérations, notamment leur place dans la hiérarchie. Puis on définira les compléments (spécifications) avant de revenir sur les opérations pour les y intégrer.

Etape 1 : Au-delà de SemerBle, les autres opérations en jeu

1.1 Les autres opérations **sur le blé**

1.1.1 Désherber en automne

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Operation'. Dans la fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper DesherberBle dans le champ 'Name', et choisir OperationBle dans le menu de la ligne 'SetParentClassId'.

- Dans l'onglet 'Méthodes héritées', cocher le bouton 'Nom automatique' de la ligne 'StateTransitionProcedure', puis cliquer sur 'Coder'. Dans la fenêtre d'édition ouverte, remplacer le code proposé par le § AX_E1.1.1a du fichier ressources_FCT.txt. Dans ce code, on ajoute un 'D' à la chaîne de caractères qui est la valeur du descripteur EtapeITBle de la culture de blé. Pour cela, on découpe la chaîne en sous-chaînes (par la fonction C strtok, d'usage fréquent) correspondant aux 5 types d'opérations, on ajoute le 'D' à la seconde sous-chaîne, puis on concatène à nouveau les 5 sous-chaînes. Si l'option '-o' a été prise dans la ligne de commande, la chaîne est affichée.

- De retour dans l'onglet 'Identité', cliquer sur 'Coder' de la ligne 'Constructor'. Dans la fenêtre d'édition, et juste au-dessous de la ligne "DEBUT DU CODE SPECIFIQUE", coller le § AX_E1.1.1b du fichier ressources_FCT.txt. Pour que le service GetRealParameterValue fonctionne sans échec, il faudra penser à ajouter une ligne dans le fichier sim1.par pour la vitesse de cette opération.

- Valider l'opération par le bouton 'Valider'.

1.1.2 Fertiliser

- Dans la fenêtre de choix des opérations, cliquer sur 'Créer'. Dans la fenêtre de modification, taper FertiliserBle dans le champ 'Name', et choisir OperationBle dans le menu de la ligne 'SetParentClassId'.

- Dans l'onglet 'Méthodes héritées', cliquer sur 'Coder' dans la ligne 'StateTransitionProcedure', puis, dans la fenêtre d'édition, remplacer le code proposé par le § AX_E1.1.2a du fichier ressources_FCT.txt. Dans ce code, on ajoute un 'F' à la chaîne de caractères qui est la valeur du descripteur EtapeITBle comme on a ajouté un 'D' pour le désherbage. On note d'ores et déjà que si plusieurs fertilisations sont faites, il y aura autant de 'F' dans la chaîne.

- De retour dans l'onglet 'Identité', cliquer sur 'Coder' de la ligne 'Constructor'. Dans la fenêtre d'édition, coller le § AX_E1.1.2b du fichier ressources_FCT.txt. On pensera à valuer la vitesse dans le fichier sim1.par.

Un autre descripteur est valué dans cette partie spécifique du constructeur : SuspensionThreshold, valeur-seuil de l'AchievementDegree de l'opération telle que, si la valeur courante est devenue supérieure à ce seuil, l'opération n'est plus interruptible (par défaut de ressources notamment). Une interruption devient nécessaire ? C'est alors l'échec du plan global. Ce qu'on veut exprimer, pour la fertilisation, c'est qu'elle n'est jamais interruptible. On fixe donc le seuil à 0 : toute valeur courante est dans la "zone" d'interruption interdite. Un détail en plus : on veut pouvoir comparer cette option avec la situation 'fertilisation interruptible'. On conditionne donc cette mise à 0 du seuil, à un paramètre externe (de type 'chaîne de caractères'). Si ce paramètre est "non", alors la valeur par défaut (1) est remplacée par 0. Penser à augmenter le fichier sim1.par.

- Valider l'opération par le bouton 'Valider'.

1.1.3 Traiter

- Dans la fenêtre de choix des opérations, cliquer sur 'Créer'. Dans la fenêtre de modification, taper TraiterBle dans le champ 'Name', et choisir OperationBle dans le menu de la ligne 'SetParentClassId'.

- Dans l'onglet 'Méthodes héritées', cliquer sur 'Coder' dans la ligne 'StateTransitionProcedure', puis, dans la fenêtre d'édition, remplacer le code proposé par le § AX_E1.1.3a du fichier ressources_FCT.txt. Dans ce code, on ajoute un 'T' à la chaîne de caractères qui est la valeur du descripteur EtapeITBle comme on a ajouté un 'F' pour la fertilisation.

- De retour dans l'onglet 'Identité', cliquer sur 'Coder' de la ligne 'Constructor'. Dans la fenêtre d'édition, coller le § AX_E1.1.3b du fichier ressources_FCT.txt. On pensera à valuer la vitesse et le seuil de suspension dans le fichier sim1.par.

- Valider l'opération par le bouton 'Valider'.

1.1.3 Récolter le blé

- Dans la fenêtre de choix des opérations, cliquer sur 'Créer'. Dans la fenêtre de modification, taper RecolterBle dans le champ 'Name', et choisir OperationBle dans le menu de la ligne 'SetParentClassId'.

- Dans l'onglet 'Méthodes héritées', cliquer sur 'Coder' dans la ligne 'StateTransitionProcedure', puis, dans la fenêtre d'édition, remplacer le code proposé par le § AX_E1.1.4a du fichier ressources_FCT.txt. Dans ce code, on ajoute un 'R' à la chaîne de caractères qui est la valeur du descripteur EtapeITBle comme on a ajouté un 'S' pour le semis.

- De retour dans l'onglet 'Identité', cliquer sur 'Coder' de la ligne 'Constructor'. Dans la fenêtre d'édition, coller le § AX_E1.1.4b du fichier ressources_FCT.txt. On pensera à valuer la vitesse dans le fichier sim1.par.

- Valider l'opération par le bouton 'Valider'.

1.2 L'opération de **coupe de la luzerne**

- Dans la fenêtre de choix des opérations, cliquer sur 'Créer'. Dans la fenêtre de modification, taper CouperLuzerne dans le champ 'Name', et choisir (attention !) OperationTracee dans le menu de la ligne 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', zone des descripteurs constants, choisir OperatedEntityId, puis donner la valeur PARCELLE, avant de cliquer sur 'Ajout'.

- Dans l'onglet 'Méthodes héritées', cliquer sur 'Coder' dans la ligne 'StateTransitionProcedure', puis, dans la fenêtre d'édition, remplacer le code proposé par le § AX_E1.2a du fichier ressources_FCT.txt. Une trace simple est effectuée dans ce code.

- De retour dans l'onglet 'Identité', cliquer sur 'Coder' de la ligne 'Constructor'. Dans la fenêtre d'édition, coller le § AX_E1.2b du fichier ressources_FCT.txt. On pensera à valuer la vitesse dans le fichier sim1.par.

Un autre descripteur est valué dans cette partie spécifique du constructeur : PriorityDegree, le degré de priorité d'exécution de l'opération, une fois qu'on a réussi à lui allouer ses ressources. L'énoncé du problème a en effet stipulé que les opérations sur le troupeau prennent le pas sur les coupes de luzerne, qui prennent le pas sur les opérations sur le blé. On décide ici d'externaliser ces valeurs, pour pouvoir éventuellement expérimenter des ordres différents. On se rappelle au passage qu'on n'a pas surchargé la valeur par défaut (99, priorité faible) pour OperationBle : il faudra y penser !

- Valider l'opération par le bouton 'Valider'.

1.3 L'opération d**'apport de fourrage au troupeau**

- Dans la fenêtre de choix des opérations, cliquer sur 'Créer'. Dans la fenêtre de modification, taper ApporterFourrage dans le champ 'Name', et choisir OperationTracee dans le menu de la ligne 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', zone des descripteurs constants, choisir OperatedEntityId, puis donner la valeur TROUPEAU, avant de cliquer sur 'Ajout'.

- Dans l'onglet 'Méthodes héritées', cliquer sur 'Coder' dans la ligne 'StateTransitionProcedure', puis, dans la fenêtre d'édition, remplacer le code proposé par le § AX_E1.3a du fichier ressources_FCT.txt. Une trace simple est effectuée dans ce code, intégrant un calcul de la quantité d'herbe apportée pour trois jours de consommation sur la base d'un paramètre de consommation par animal et par jour. A valuer dans sim1.par.

- De retour dans l'onglet 'Identité', cliquer sur 'Coder' de la ligne 'Constructor'. Dans la fenêtre d'édition, coller le § AX_E1.3b du fichier ressources_FCT.txt. On pensera à valuer la vitesse et la priorité d'exécution dans le fichier sim1.par.

- Valider l'opération par le bouton 'Valider'.

Sauvegarder la base !

Etape 2 : Définition des spécifications complémentaires

2..1 La définition des **réquisitions de ressources** par les opérations

2..1.1 La réquisition de t**racteur outillé ...**

... pour l'apport de fourrage, les coupes de luzerne et les opérations sur le blé a été définie à l'Etape 3.2 de l'Acte VII.

2..1.2 La réquisition du **matériel pour la récolte du blé ...**

... a été définie à l'Etape 3.1 de l'Acte VII (RequisitionUnitaireMaterielRecolte).

2..2 L'attachement, aux opérations, des réquisitions de ressources

2..2.1 Le tracteur outillé à l'apport de fourrage

- Dans la fenêtre de choix des opérations, sélectionner ApporterFourrage et cliquer sur 'Modifier'. Dans l'onglet 'Descripteurs hérités' de la fenêtre de modification, sélectionner le descripteur variable RequiredResources, puis taper

requisitionTracteurOutille dans la zone de la valeur. Cliquer sur 'Ajout'.

- Valider l'opération par le bouton 'Valider'.

2..2.2 Le tracteur outillé aux coupes de luzerne

- Dans la fenêtre de choix des opérations, sélectionner CouperLuzerne et cliquer sur 'Modifier'. Dans l'onglet 'Descripteurs hérités' de la fenêtre de modification, sélectionner le descripteur variable RequiredResources, puis taper requisitionTracteurOutille dans la zone de la valeur. Cliquer sur 'Ajout'.

- Valider l'opération par le bouton 'Valider'.

2..2.3 Le matériel de récolte à la récolte du blé

- Dans la fenêtre de choix des opérations, sélectionner RecolterBle et cliquer sur 'Modifier'. Dans l'onglet 'Descripteurs hérités' de la fenêtre de modification, sélectionner le descripteur variable RequiredResources, puis taper requisitionUnitaireMaterielRecolte dans la zone de la valeur. Cliquer sur 'Ajout'.

- Valider l'opération par le bouton 'Valider'.

2..3 Surchargement de la valeur de **priorité des opérations sur le blé**

- Dans la fenêtre de choix des opérations, sélectionner OperationBle et cliquer sur 'Modifier'. Dans l'onglet 'Identité', cliquer sur 'Modifier' dans la ligne du constructeur. Dans la fenêtre d'édition, coller le § AX_E2.3 du fichier ressources_FCT.txt. On pensera à valuer cette priorité dans le fichier sim1.par.

- Valider l'opération par le bouton 'Valider'.

Sauvegarder la base !

Etape 3 : Déclaration de valeurs pour les paramètres

Dans le répertoire des données, ouvrir le fichier sim1.par.

- Installer un paragraphe consacré à l'animal, à la suite de celui consacré au végétal. Coller pour cela le § AX_E3a du fichier ressources_SIM.txt. A ce stade, on value le paramètre de consommation quotidienne d'herbe par un animal.

- Dans le paragraphe consacré à la conduite, compléter les paramètres de vitesse, pour qu'il devienne comme le § AX_E3b du fichier ressources_SIM.txt.

- Dans ce même paragraphe, après les lignes fixant les vitesses, ajouter les lignes du AX_E3c du fichier ressources_SIM.txt, pour fixer les degrés de priorité des opérations.

- ... puis enfin, après les lignes fixant les priorités des opérations, ajouter les lignes du AX_E3d du fichier ressources_SIM.txt, pour fixer les seuils de suspension.

Etape 4 : Vérification

4.1 Générer tout le code source, puis compiler la base de connaissances.

4.2 Relancer l'exécution, afin de vérifier la consistance du codé développé, et la bonne lecture du fichier des paramètres modifié :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb
```

On obtient une trace qui comprend le passage suivant, dénotant une erreur de compilation :

```
/home/rellier/APPLIS_DIESE/KBS/cdiese/SIBEMOL/SiBemol_Tutoriel.1.0/source/UserOperation.cc: In
function 'bool apporterFourrage_stateTransitionProcedure_Body(EntityMethod*)':
/home/rellier/APPLIS_DIESE/KBS/cdiese/SIBEMOL/SiBemol_Tutoriel.1.0/source/UserOperation.cc:47:
error: 'HERBE_DISPONIBLE' was not declared in this scope
```

On reconnait que :
- le fichier contenant l'erreur est UserOperation.cc
- la fonction contenant l'erreur est apporterFourrage_stateTransitionProcedure_Body
- l'erreur est signalée à la ligne 47 du fichier
- que le compilateur a rencontré un élément de langage qu'il n'a pas su interpréter : HERBE_DISPONIBLE

Pour faire un diagnostic plus précis, il est utile de visualiser le fichier et la fonction en cause. Pour cela :

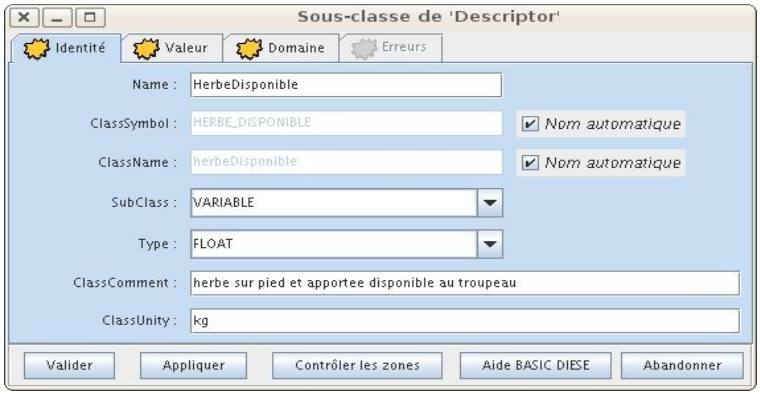
- Dans le menu 'Génération', choisir 'Edition des fichiers source C++'. Une fenêtre s'ouvre portant une liste de fichiers. Sélectionner UserOperation.cc puis cliquer sur 'Editer'. Une fenêtre d'édition du fichier s'ouvre.

- Se déplacer sur la ligne 47 avec la fonction dédiée de l'éditeur. On constate que la ligne erronée est :

```
float oldDispo = pTroupeau->GetFloatVarValue(HERBE_DISPONIBLE);
```

On réalise alors que le code fait référence à un descripteur de la classe Troupeau, HerbeDisponible dont le symbole de classe est HERBE_DISPONIBLE, sans que ce descripteur ait été créé. Une fois ce diagnostic posé, fermer la fenêtre d'édition de UserOperation.cc (sans l'avoir modifié), puis quitter la fenêtre de choix des fichiers à éditer.

- Créer le descripteur variable HerbeDisponible, de type FLOAT, d'unité le kilogramme, et avec un commentaire tel que "herbe sur pied et apportee disponible au troupeau". ⏩



- Ajouter ce descripteur à la classe d'entités Troupeau, dans l'onglet 'Descripteurs propres' de la fenêtre de modification de cette classe, dans la partie 'AddVariableDescriptor' de l'onglet.

- Puisqu'une nouvelle classe a été créée, c'est encore une fois 'Tous les sources' qu'il faut choisir dans le menu 'Génération', pour compiler la base de connaissances.

Sauvegarder la base !

Compiler la base de connaissances, puis relancer l'exécution, avec l'option '-o', pour suivre la progression du semis, seul en jeu à ce stade par son insertion dans l'activité SemisBle, la seule incluse dans le plan de conduite :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

On obtient la trace ci-dessous : ⏩

```
---------[exec]>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o

'/home/rellier/APPLIS_DIESE/KBS/cdiese/SIBEMOL/SiBemol_Tutoriel.1.0/exec'
./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o


15/10/2000 historique IT sur monBle : S_*_*_*_*
15/10/2000 monBle a atteint le stade APRES_SEMIS
DIM 15/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.200 (jean paul)
DIM 15/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.400 (jean paul)
LUN 16/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.600 (jean paul)
LUN 16/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.800 (jean paul)
LUN 16/10/2000 'semerBle_44' sur 'parcelleBle' -> 1.000 (jean paul)
31/03/2001 monBle a atteint le stade EPI_1CM
04/05/2001 monBle a atteint le stade GONFLEMENT
24/05/2001 monBle a atteint le stade FLORAISON
duree =  0h00'00
---------[exec]>
```

Le caractère 'S' apparaît dans la chaîne des étapes de l'itinéraire (StateTransitionProcedure de SemerBle). La trace des passages de stade reste la même que lors des étapes antérieures du développement.

Et on voir apparaître la trace de la progression du degré de progression de l'effet (AchievementDegree), fait par le démon OperationTraceeDegreProgressionMonitor. Puisque la vitesse est de 1/5 (paramètre "vitesseProgression" "semerBle", doublé par la mobilisation de deux unités de MO, jean et paul), le degré de 1 est atteint en cinq 'pas'.

Il y a cependant une anomalie manifeste : l'effet progresse de 40% le dimanche (et de 60% le lundi), alors qu'on s'attend à seulement 20%, puis 20% de plus lors des quatre jours suivants. On se rappelle alors que :
- le 'pas' de OperationTracee est de 8 (heures), depuis l'Etape 1 de l'Acte VIII.
- c'est à 8h du matin qu'est établie la première liste des opérations à exécuter (Etape 6.2 de l'Axte IX).

Il est donc normal d'observer un passage à 0.2 à 15h (la dernière unité d'horloge dans le 'pas [8h 15h]), le passage à 0.4 à 23h, le passage à 0.6 le lundi à 7h, à 0.8 le lundi à 15h, le passage à 1 le lundi à 23h. Mais ce qui ne va pas, c'est que jean et paul travaillent jour et nuit : il faut empêcher ça !

Noter que si notre unité d'horloge était '1 DAY' et non '1 HOUR', et si, corollairement, le 'pas' de nos opérations était 1, et toutes choses ajustées par ailleurs, alors on observerait bien une progression quotidienne de 0.2, et la durée et la plage de travail dans la journée ne seraient pas utiles dans notre modèle

Parce que l'unité '1 HOUR' a été choisie pour autoriser une modélisation plus fine du travail, et avant d'entamer la définition des activités primitives (en référençant les opérations qu'on vient de créer), on va consacrer un Acte du développement à coder nos connaissances sur les horaires de travail de la main d'œuvre.

ACTE XI : Les horaires de travail de la main d'œuvre

Les unités de MO sont instanciées dans le fichier de données systemeOperant.inc (voir Etape 4.2 de l'Acte VII). Dès cette instanciation, l'état de disponibilité est 'vrai' (valeur 1), autorisant, à l'Acte précédent, l'opération de semis..

C'est la valeur de ce descripteur que le moteur regarde d'abord, pour continuer ou non à examiner la possibilité d'allouer cette ressource à l'activité qui la requiert. C'est donc la valeur de ce descripteur qu'on va tenter de faire alterner entre 'vrai' et 'faux' (valeur 0) : passage de 0 à 1 en début de plage de travail, passage de 1 à 0 en fin de plage de travail. Dans notre situation d'étude, on décide que la plage de travail, commune à tous les agents, démarre à l'heure où le gestionnaire décide des opérations à acter (8h), et se finit 8 heures après, sans considération de la coupure de la mi-journée. La même plage revient tous les jours, sauf éventuellement le dimanche.

CONTROL DIESE fournit des services pour installer cette séquence de plages de disponibilité des ressources. Ce sont les événements (et les processus associés) de mobilisation et d'immobilisation. Succinctement, un événement de mobilisation a un post-événement automatique d'immobilisation, et un événement d'immobilisation a un post-événement automatique de mobilisation. Il suffit au développeur d'une application ... :
- de choisir le premier événement de la série et de fixer son instant d'occurrence
- de particulariser les instants d'occurrence de chaque événement par rapport au précédent.

Dans notre situation d'étude, on va
- lors de l'instanciation d'un agent (à la valeur d'horloge 0, instant de lecture du fichier systemeOperant.inc), programmer un passage à l'état 'disponible' à 8h du matin
- programmer le passage à l'état 'indisponible' 8 heures après le passage à 'disponible' (le même jour, donc)
- programmer le passage à l'état 'disponible' le lendemain à 8h du matin

Etape 1 L'événement qui rend une ressource disponible

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Event', puis cliquer sur le bouton 'Créer'.

- Dans la fenêtre de modification ouverte, taper MobilisationAgent dans le champ 'Name', puis, dans le menu du champ 'ParentClass', choisir la classe prédéfinie ResourceMobilizationEvent .

- Dans l'onglet 'Processus', et dans le menu du champ 'ProcessedEntityClassId', choisir AGENT, puisque c'est d'un agent qu'on va modifier l'état.

- Dans l'onglet 'Post-événements', laisser SetGenerateNextEvent dans le menu à droite de la ligne étiquetée 'NextEventClass' (parce que ce n'est pas seulement la date d'occurrence qu'on veut spécifier, ou un autre attribut quelconque, mais aussi le type d'événement). Puis cocher le bouton 'Nom automatique' dans la ligne 'Set ... NextEvent', avant de cliquer sur le bouton 'Coder' de la même ligne.

- Dans la fenêtre d'édition ainsi ouverte, coller le § AXI_E1 du fichier ressources_FCT.txt, en remplacement de l'amorce de code proposée par Solfege. Pour l'essentiel, on instancie un événement de la classe ImmobilisationAgent (qu'on ne manquera pas de créer), et on calcule sa date d'occurrence (à l'aide d'un paramètre pour la durée de la plage de travail). On remarque que le prochain événement (pNextEvent) est un attribut du précédent (pCurrentEvent), attaché par le service prédéfini SetNextEvent : le moteur saura le retrouver en tant que tel, pour assurer lui-même son insertion dans l'agenda de la simulation. On remarque enfin que le processus va porter sur un agent individuel : il faudra autant d'événements de cette classe que d'agent en jeu.

- Sauvegarder le contenu de cette fonction, quitter l'éditeur, puis valider la définition de l'événement par 'Valider'.

Etape 2 L'événement qui rend une ressource indisponible

- Dans la fenêtre de choix des événements, cliquer sur le bouton 'Créer'.

- Dans la fenêtre de modification ouverte, taper ImmobilisationAgent dans le champ 'Name', puis, dans le menu du champ 'ParentClass', choisir ResourceImmobilizationEvent.

- Dans l'onglet 'Processus', et dans le menu du champ 'ProcessedEntityClassId', choisir AGENT.

- Dans l'onglet 'Post-événements', laisser SetGenerateNextEvent dans le menu à droite de la ligne étiquetée 'NextEventClass'. Cocher le bouton 'Nom automatique' dans la ligne 'Set ... NextEvent', avant de cliquer sur 'Coder'.

- Dans la fenêtre d'édition ainsi ouverte, coller le § AXI_E2 du fichier ressources_FCT.txt, en remplacement de l'amorce de code proposée par Solfege. Pour l'essentiel, on instancie un événement de la classe MobilisationAgent (créée ci-dessus), et on calcule sa date d'occurrence. Le début de plage est normalement le lendemain de la fin de plage. On introduit cependant la possibilité de respecter un jour de repos pour les agents, le dimanche. Ainsi, la reprise du travail après le samedi soir se situe, à cette condition, le lundi matin. Un paramètre permet de choisir ou non cette

option dans le fichier externe sim1.par. Ne pas oublier de l'y écrire. L'usage de la structure C tm et de la fonction C mktime permet d'ajouter 1 ou 2 au jour. Il suffit en outre de fixer l'heure avec le paramètre qui fixe l'heure de révision du plan par le gestionnaire. La mécanique d'insertion dans l'agenda est la même que pour l'événement d'immobilisation.

- Sauvegarder le contenu de cette fonction, quitter l'éditeur, puis valider la définition de l'événement par 'Valider'.

Etape 3 La première mise en 'disponibilité'

- Dans la fenêtre de choix des ressources atomiques (item 'SingleResource'), choisir Agent, puis cliquer sur 'Modifier'. Dans la fenêtre de modification, onglet 'Identité', cliquer sur 'Modifier' dans la ligne du constructeur.

- Dans la fenêtre d'édition, au-dessous de la ligne "DEBUT DU CODE SPECIFIQUE", coller le § AXI_E3 du fichier ressources_FCT.txt. On détermine si le jour courant est un dimanche, pour ne programmer l'événement que le lendemain, en fonction du paramètre dédié. Ensuite, on fixe l'heure comme pour la mobilisation (voir l'Etape 2). Enfin, l'événement créé est inséré dans l'agenda. Par précaution, on envisage l'échec, pour une raison quelconque, de cette insertion ; alors, selon une attitude de programmation qu'on développera ultérieurement, on récupère l'espace-mémoire qu'on vient d'allouer à l'événement qui ne servira pas. DeleteInDepth est un service prédéfini dans DIESE.

On note que dans ce code, on insère explicitement l'événement dans l'agenda, alors qu'on ne l'a pas fait pour le post-événement des (im)mobilisations (Etapes 1 et 2). On rappelle que dans ce dernier cas, le moteur de simulation sait où trouver l'événement à insérer, dès le retour de la fonction qui l'instancie. Une telle convention de portée générale ne peut pas exister pas entre le moteur et un code spécifique.

- Sauvegarder le contenu de ce constructeur, quitter l'éditeur, puis valider la nouvelle définition de l'agent par 'Valider'.

Attention ! Pour que les agents ne deviennent disponibles qu'à heure voulue (et non en t=0), il faut modifier l'instanciation des agents dans le fichier systemeOperant.inc, dans le répertoire des données. Y remplacer les trois lignes avec `availabilityStatus = 1;` par `availabilityStatus = 0;`.

Sauvegarder la base !

Etape 4 Validation du codage des plages de travail

4.1 Compiler la base de connaissances**, puis relancer** l'exécution :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

On obtient la trace ci-dessous : ⏭

```
---------[exec]>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o

'/home/rellier/APPLIS_DIESE/KBS/cdiese/SIBEMOL/SiBemol_Tutoriel.1.0/exec'
./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o


!!!============== InexistantTypedParameter ==============
!!! Tentative d'acc?s (get value) ? un param?tre cha?ne de caract?res inexistant : 'agent" 'tra
vailDimancheAutorise'
!!!============================
---------[exec]>
```

... qui nous rappelle qu'il faut déclarer, dans un fichier de données, les paramètres exploités dans le code.

4.2 Compléter le fichier des paramètres

- Dans le répertoire des données, ouvrir le fichier sim1.par. Insérer le § AXI_E4 du fichier ressources_SIM.txt en fin de la section 'CONDUITE' de sim1.par.

4.3 Relancer l'exécution :

On obtient la trace ci-dessous : ⏭

```
---------[exec]>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o

'/home/rellier/APPLIS_DIESE/KBS/cdiese/SIBEMOL/SiBemol_Tutoriel.1.0/exec'
./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o


15/10/2000 historique IT sur monBle : S_*_*_*_*
15/10/2000 monBle a atteint le stade APRES_SEMIS
DIM 15/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.200 (jean paul)
LUN 16/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.400 (jean paul)
MAR 17/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.600 (jean paul)
MER 18/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.800 (jean paul)
JEU 19/10/2000 'semerBle_44' sur 'parcelleBle' -> 1.000 (jean paul)
31/03/2001 monBle a atteint le stade EPI_1CM
04/05/2001 monBle a atteint le stade GONFLEMENT
24/05/2001 monBle a atteint le stade FLORAISON
duree =  0h00'00
---------[exec]>
```

On y lit que le semis commence un dimanche, pour progresser de 1/5 les 4 jours suivants, comme attendu..

Si, dans le fichier sim1.par, on remplace "oui" par "non" comme valeur du paramètre 'travailDimancheAutorise', la trace devient la suivante, comme attendu : ⏩

```
16/10/2000 historique IT sur monBle : S_*_*_*_*
16/10/2000 monBle a atteint le stade APRES_SEMIS
LUN 16/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.200 (jean paul)
MAR 17/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.400 (jean paul)
MER 18/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.600 (jean paul)
JEU 19/10/2000 'semerBle_44' sur 'parcelleBle' -> 0.800 (jean paul)
VEN 20/10/2000 'semerBle_44' sur 'parcelleBle' -> 1.000 (jean paul)
01/04/2001 monBle a atteint le stade EPI_1CM
05/05/2001 monBle a atteint le stade GONFLEMENT
24/05/2001 monBle a atteint le stade FLORAISON
```

Noter que l'atteinte des stades EPI_1CM et GONFLEMENT a été retardée d'un jour, puisqu'elle est fonction d'une accumulation d'unités de développement. Quant au stade FLORAISON, un effet de seuil le maintient à la même date. En effet le blé accumule 192 UDev entre le 04/05 et le 23/05 (fichier UDEV_BLE.txt dans le répertoire IN_FILES), insuffisants (il en faut 200) pour passer à FLORAISON le 23/05. Par contre, le blé accumule 2022 UDev entre le 05/05 et le 24/05, suffisants pour passer à FLORAISON le 24/05, comme quand GONFLEMENT est atteint le 04/05.

Un autre test de validité de notre développement consiste à modifier le pool de ressources, et la modalité d'allocation des deux agents.

Si on enlève paul du pool du personnel, dans systemeOperant.inc, La trace ne montre plus l'exécution du semis, ni bien entendu la progression de son effet. Le semis n'étant pas effectué, les stades suivants de son développement ne sont pas atteints.

Si, toujours sans la présence de paul, on change la valeur de ModalitéDeuxAgents de monManager, de ANYONE à MAX (dans systemeDecisionnel.inc), en supprimanr les '//' en début d'e ligne, alors on obtient la trace suivante : ⏩

```
16/10/2000 historique IT sur monBle : S_*_*_*_*
16/10/2000 monBle a atteint le stade APRES_SEMIS
LUN 16/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.100 (jean)
MAR 17/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.200 (jean)
MER 18/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.300 (jean)
JEU 19/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.400 (jean)
VEN 20/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.500 (jean)
SAM 21/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.600 (jean)
LUN 23/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.700 (jean)
MAR 24/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.800 (jean)
MER 25/10/2000 'semerBle_43' sur 'parcelleBle' -> 0.900 (jean)
JEU 26/10/2000 'semerBle_43' sur 'parcelleBle' -> 1.000 (jean)
01/04/2001 monBle a atteint le stade EPI_1CM
05/05/2001 monBle a atteint le stade GONFLEMENT
24/05/2001 monBle a atteint le stade FLORAISON
```

Le semis est réalisé par la seule unité de MO disponible, mais la durée totale de l'opération est doublée. On de travaille pas non plus le dimanche 22/10.

Si on supprime un tracteur outillé du pool de matériels, le déroulé du semis est identique, parce qu'une seule unité de MO mobilise l'autre tracteur outillé, qui est disponible. Si on enlève les deux tracteurs outillés, alors le semis n'est pas effectué.

ACTE XII : Le développement complet des activités primitives

On développe ici les activités primitives, en référençant les opérations développées dans l'Acte X. Comme pour les opérations, on définit d'abord sommairement les activités, notamment leur place dans la hiérarchie. Puis on développe les spécifications de 'performer', et les connaissances fonctionnelles essentielles sur leur ouverture et leur fermeture.

Au cours de ce développement, on pensera nécessairement aux activités agrégées qui encapsulent les primitives (par exemple, l'itinéraire technique du blé). Par exemple, on pourra décider le report dans une activité-séquence du prédicat d'ouverture du premier élément de la séquence. De tels choix seront gardés en mémoire, et implémentés ultérieurement.

Etape 1 : Au-delà de SemisBle, les autres activités en jeu

1.1 Les autres activités **sur le blé**

1.1.1 Désherbage d'automne

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Activity', puis '... Primitive'. Dans la fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper DesherbageAutomne dans le champ 'Name', et choisir ActiviteBle dans le menu de la ligne 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', zone des descripteurs variables, choisir OperationClassId, puis donner la valeur DESHERBER_BLE, avant de cliquer sur 'Ajout'.

- Valider la classe d'activité par le bouton 'Valider'.

1.1.2 Fertilisation

- Dans la fenêtre de choix des activités primitives, cliquer sur 'Créer'. Dans la fenêtre de modification, taper ApportAzote dans le champ 'Name', et choisir ActiviteBle dans le menu de la ligne 'SetParentClassId'.

- Fixer à FERTILISER_BLE le descripteur hérité variable OperationClassId, puis valider la classe.

1.1.3 Traitement

- Dans la même fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper TraitementFongicideBle dans le champ 'Name', et choisir ActiviteBle dans le menu de la ligne 'SetParentClassId'.

- Fixer à TRAITER_BLE le descripteur hérité variable OperationClassId, puis valider la classe.

1.1.4 Récolte

- Dans la même fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper MoissonBle dans le champ 'Name', et choisir ActiviteBle dans le menu de la ligne 'SetParentClassId'.

- Fixer à RECOLTER_BLE le descripteur hérité variable OperationClassId, puis valider la classe.

1.2 L'activité de **coupe de la luzerne**

- Dans la même fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper RecolteLuzerne dans le champ 'Name', et choisir (attention !) PrimitiveActivity dans le menu de la ligne 'SetParentClassId'.

- Fixer à COUPER_LUZERNE le descripteur hérité variable OperationClassId.

- Dans l'onglet 'Identité', cliquer sur le bouton 'Modifier' de la ligne 'Constructor'. Une fenêtre d'édition s'ouvre, avec le code que Solfege a pu générer à partir des spécifications déjà fournies.

- Coller le § AXII_E1.2 du fichier ressources_FCT.txt, et cela juste après la ligne contenant la chaîne "DEBUT DU CODE SPECIFIQUE" (et juste avant le '};' final).

Comme dans le constructeur de ActiviteBle (Etapes 5.2.2 et 5.2.4 de l'Acte VIII), on surcharge la valeur du SelectorFctId de la spécification du 'performer', seulement si, dans le fichier externe, on a donné au descripteur ModalitéDeuxAgents une valeur différente de celle par défaut. D'autre part, on crée une instance de OperatedObjectSpecification sans indication qui permette à ce stade au développement de désigner le lieu des coupes. Il faudra déclarer ce lieu (par descripteur déjà utilisé UniteDeGestion, on verra plus tard où), et utiliser le mécanisme de démon (déjà utilisé) pour désigner, dans l'OperatedObjectSpecification, l'objet opéré par l'activité primitive.

- Sauvegarder le contenu du constructeur de RecolteLuzerne, quitter l'éditeur, puis cliquer sur 'Valider'.

1.3 L'activité d**'apport de fourrage au troupeau**

- Dans la même fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper Affouragement dans le champ 'Name', et choisir PrimitiveActivity dans le menu de la ligne 'SetParentClassId'.

- Fixer à APPORTER_FOURRAGE le descripteur hérité variable OperationClassId.

- Dans l'onglet 'Identité', cliquer sur le bouton 'Modifier' de la ligne 'Constructor'.

- Dans la fenêtre d'édition, coller le § AXII_E1.3 du fichier ressources_FCT.txt, et cela juste après la ligne contenant la chaîne "DEBUT DU CODE SPECIFIQUE" (et juste avant le '};' final).

On crée une instance de OperatedObjectSpecification sans indication qui permette à ce stade au développement de désigner le lieu des coupes. Comme pour les coupes de luzerne ci-dessus, on fera usage du descripteur UniteDeGestion pour désigner, dans l'OperatedObjectSpecification, l'objet opéré par l'activité primitive.

De plus, on précise explicitement que la priorité d'allocation de l'activité doit être relativement forte [8] [9] [10]. En effet, Il faut éviter que la ressource spécialisée de type AgentCompetentVaches ne soit accaparée par une activité qui n'a besoin que d'un agent en général, spécialisé ou non, faisant ainsi échouer l'allocation pour l'affouragement. Cette situation peut se produire, par exemple dans la concurrence avec le semis de blé, parce que l'opérateur l'allocation ANYONE pioche n'importe quelle instance de AGENT. Quant à l'opérateur MAX choisi en option, il va tout accaparer, et empêcher toute allocation pour l'affouragement.

- Sauvegarder le contenu du constructeur de Affouragement, quitter l'éditeur, puis cliquer sur 'Valider'.

Etape 2 : Définition des spécifications complémentaires

2..1 La définition des **réquisitions de 'performer'** par les activités

2..1.1 La réquisition de **l'équipe de récolte du blé ...**

... a été définie à l'Etape 3.3.2 de l'Acte VII (RequisitionUnitairePersonnelRecolte).

2..1.2 La réquisition de **deux agents quelconques** (exactement ou au maximum) **...**

... a été définie à l'Etape 5.2.1 de l'Acte VIII. On l'a déjà affectée à l'activité-mère ActivitéBle (Etape 5.2.2 de l'Acte VIII). Il reste à l'affecter à la récolte de luzerne, comme envisagé à l'Etape 3 de l'Acte VII.

2.1.3 La réquisition d'**un agent 'compétent vaches'**

Qu'on utilisera pour l'affouragement, comme envisagé à l'Etape 3 de l'Acte VII.

-Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Specification', puis 'PerformerSpec'. Dans la fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper UnVacher dans le champ 'Name'. Puis choisir PERFORMER_SPECIFICATION dans le champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', et dans la zone des descripteurs variables, donner à EntitySpecReferenceClassId (descripteur variable) la valeur AGENT_COMPETENT_VACHES. Dans la zone descripteurs constants, donner la valeur ANYONE au descripteur SelectorFctId, et la valeur 1 au descripteur SelectorFctArg.

- Valider la définition de cette spécification de 'performer' par le bouton 'Valider'.

2.1.4 La réquisition d'**un agent quelconque**

Qu'on utilisera pour la fertilisation, le désherbage et le traitement, en surchargement de la spécification DeuxAgents attribuée à la classe-mère ActivitéBle, et comme envisagé à l'Etape 3 de l'Acte VII.

-Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Specification', puis 'PerformerSpec'. Dans la fenêtre de choix, cliquer sur 'Créer'. Dans la fenêtre de modification, taper UnAgent dans le champ 'Name'. Puis choisir PERFORMER_SPECIFICATION dans le champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', et dans la zone des descripteurs variables, donner à EntitySpecReferenceClassId (descripteur variable) la valeur AGENT. Dans la zone descripteurs constants, donner la valeur SETS au descripteur SelectorFctId (justification à l'Etape 3 de l'Acte VII), et la valeur 1 au descripteur SelectorFctArg.

- Valider la définition de cette spécification de 'performer' par le bouton 'Valider'.

2..2 L'attachement, aux activités, des réquisitions de ressources

2..2.1 L'agent 'compétent vaches' à l'affouragement

- Dans la fenêtre de choix des activités primitives, sélectionner Affouragement et cliquer sur 'Modifier'. Dans l'onglet 'Descripteurs hérités' de la fenêtre de modification, sélectionner le descripteur variable PerformerSpecAttribute, puis

[8] Si une activité A a une plus forte priorité d'allocation (e.g. 80) qu'une activité B (e.g. 99), et si A et B figurent dans le même jeu d'activités à mettre en œuvre concurremment, alors le moteur d'allocation commencera par essayer de satisfaire les réquisitions de ressources de A. Puis il essaiera de satisfaire les réquisitions de ressources de B, compte tenu des allocations au profit de A.

[9] Le développeur peut coder lui-même une fonction d'ordre des activités pour l'allocation des ressources, pour un motif particulier à son domaine d'étude. Par défaut, le moteur de simulation ordonne les activités selon la valeur du descripteur AllocationPriotityDegree, puis envoie en fin de liste les activités dont une spécification de ressource est dotée de l'opérateur MAX (en conservant l'ordre interne à ce sous ensemble).

[10] Bien différencier la priorité d'allocation (d'une activité) et la priorité d'exécution (d'une opération). On peut vouloir que Act_A soit prioritaire sur Act_B pour l'allocation des ressources, et vouloir que l'effet de Ot_A soit réalisé après l'effet de Ot_B,. Par exemple quand l'effet de Ot_A dépend de l'effet de Ot_B pour la même valeur de l'horloge de simulation.

taper unVacher dans la zone de la valeur. Cliquer sur 'Ajout'.

- Valider l'activité par le bouton 'Valider'.

2..2.2 Le couple de deux agents à la récolte de luzerne

- Dans la même fenêtre de choix, sélectionner RecolteLuzerne et cliquer sur 'Modifier'. Dans l'onglet 'Descripteurs hérités', taper deuxAgents une fois choisi le descripteur variable PerformerSpecAttribute. Cliquer sur 'Ajout'.

- Valider l'activité par le bouton 'Valider'.

2..2.3 L'équipe de récolte pour le blé

- Dans la même fenêtre de choix, sélectionner MoissonBle et cliquer sur 'Modifier'. Dans l'onglet 'Descripteurs hérités', taper requisitionUnitairePersonnelRecolte pour le descripteur variable PerformerSpecAttribute. Cliquer sur 'Ajout'.

- Valider l'activité par le bouton 'Valider'.

2..2.4 Un agent tout seul à certaines activités de l'itinéraire sur blé

Dans la même fenêtre de choix, sélectionner successivement TraitementFongicideBle, DesherbageAutomne et ApportAzote. Pour chacune, cliquer sur 'Modifier', et ...

- Dans l'onglet 'Identité', ouvrir une fenêtre d'édition du constructeur (par 'Modifier'). Y coller, dans la partie spécifique du code, le §AXII_E2.2.4 du fichier ressources_FCT.txt. On commence par récupérer la mémoire allouée à l'instance de la classe DeuxAgents créée lors de l'instanciation de la classe-mère[11]. Ensuite, on crée une instance de UnAgent, qui devient la nouvelle valeur de la spécification de 'performer'.

- Sauvegarder le contenu, quitter l'éditeur et valider l'activité par le bouton 'Valider'.

Sauvegarder la base !

Etape 3 : Les connaissances sur les ouvertures/fermetures des activités primitives

Pour que les opérations développés dans les Actes précédents, puis enchassées dans les activités primitives, puissent être exécutées, il faut, entre autres choses, que lesdites activités soient passées dans la situation OPEN. On a vu, à l'Etape 3 de l'Acte IX, que la situation initiale d'une activité est WAITING. L'éventualité du passage de WAITING à OPEN est testé par le moteur lors de chaque événement RevisionMatinaleDuPlan (voir l'Etape 6.1 de l'Acte IX).Ce passage dépend des spécifications d'ouverture des activités, exprimées par :
- une fenêtre temporelle : c'est un intervalle entre deux valeurs de l'horloge de simulation
- un prédicat codé par le développeur pour renvoyer 'vrai' quand il est opportun d'effectuer le passage à OPEN

Par défaut se spécification, la fenêtre est infiniment large, et le prédicat renvoie toujours 'vrai'. Autrement dit, il n'y a aucun obstacle à l'ouverture de l'activité dès que possible. "Dès que possible" s'entend compte tenu des contraintes venant de la position de l'activité dans une activité agrégée. Par exemple, dans la séquence {A B}, l'ouverture "dès que possible" de B, ce n'est qu'immédiatement après la fermeture de A.

Une autre spécification est la date d'ouverture au plus tard, celle à partir de laquelle l'activité de doit plus être ouverte. La place de l'activité dans le plan détermine alors si l'exécution du plan peut se poursuivre ou non.

Quant à la fermeture d'une activité primitive, elle est généralement commandée par l'atteinte de l'effet complet de l'opération. Sans s'étendre sur le cas particulier des opérations permanentes (définies dans l'ontologie, et pour lesquelles c'est inversement la fermeture explicite de l'activité qui provoque l'arrêt de l'effet de l'opération), la fermeture contrainte de l'activité provoque sans échec l'arrêt de l'effet de l'opération seulement si le seuil de fermeture de l'activité le permet.

Dans notre périmètre d'étude, nous établissons de la manière suivante les conditions d'ouverture/fermeture des activités :

activité	fenêtre ouverture	prédicat d'ouverture	ouverture au plus tard	prédicat de fermeture
semis blé				
désherbage		jour julien >=3 20 (année semis)		jour julien >=3 35
apport azote		- n° 1 : j. julien >= 53 (année récolte) - n° 2 : stade Epi1cm atteint - n° 3 : stade Gonflement atteint	- n° 1 : julien >= 68 - n° 2 : 7j après Epi1cm - n° 3 : 7j après Gonflement	
traitements		- n° 1 : julien >= 110 (année récolte) - n° 2 : stade Floraison atteint	- n° 1 : julien >= 115 - n° 2 : 5j après Floraison	

[11] L'opérateur C++ new, appliqué à une classe, commence par construire ses classes-mères, dans l'ordre décroissant d'éloignement de parenté (la plus "ancêtre" d'abord, donc), en allouant à chaque fois de la mémoire pour les compléments de définition au niveau courant. Lors du complément de définition pour ApportAzote, l'allocation pour le 'performer' de ActiviteBle est donc faite. Noter au passage que la désallocation par l'opérateur C++ delete récupère la mémoire dans l'ordre inverse.

récolte du blé		julien >= 211 (année récolte)		
couper la luzerne		- n° 1 : 1er juin - suivantes : après délai		- n° 1 : 10j après début - suivantes : 7j après début
apporter le fourrage		croissance nette de l'herbe < seuil + au moins 5j après le dernier apport		

On remarque que l'ouverture n'est jamais spécifiée par une fenêtre explicite. Cette attitude est générale dans l'étude des systèmes cyclés : pour qu'une seule expression de l'ouverture soit valide pour tous les cycles, elle doit être fonctionnelle. Pour les systèmes à cycle annuel, comme le nôtre, c'est notamment le cas de l'expression par un jour julien, ou par une date calendaire sans mention du millésime.

Le semis du blé ne comporte aucune spécification, il peut donc débuter dès le début de la simulation.

Remarque importante sur la fermeture, en trois considérations :

- Les TimeGranulatedOperation's et les QuantityGranulated Operation's sont contraintes, dans l'ontologie, d'avoir un seuil de fermeture de l'activité de 1. En conséquence, et avant que l'opération ne soit terminée, le moteur de simulation ne sait pas fermer sans échec l'activité primitive qui les contient. La raison en est que le mode de progression de l'effet (voir la note de bas de page dans l'Etape 1 de l'Acte VIII) permet de quantifier mais pas d'identifier, dans l'objet opéré, une partie opérée et une partie non opérée. Ainsi, tout processus biophysique ou opération portant ultérieurement sur cet objet non totalement opéré supposerait un état uniforme de manière erronée. Et la différenciation de deux processus ou opérations, chacun sur une des deux parties, est impossible sans identification des deux parties. Cet avantage est réservé aux UnitGranulatedOperation's parce que leur effet touche à chaque 'pas' des instances identifiées comme éléments de l'objet opéré.

- La spécification de fermeture du désherbage et des coupes de luzerne est donc incompatible avec le choix tu type TIME_GRANULATED_OPERATION. Supposons que l'opération de désherbage dure au-delà du jour 335, pour une quelconque raison. Alors, parce que la fermeture de cette activité n'est requise par aucune autre qui lui est liée dans le plan (l'activité suivante dans l'itinéraire est programmée pour beaucoup plus tard), le prédicat de fermeture n'est pas testé par le moteur puisque le seuil autorisant la fermeture de l'activité (1.0) n'est nécessairement pas atteint. Le prédicat de fermeture ne sera testé qu'une fois terminée l'opération, pour vérifier qu'aucune autre condition n'est violée, mais c'est alors sans intérêt pour nous.

- Si le développeur a une bonne raison de mettre le plan en échec si le désherbage n'est pas terminé au jour 335, il faut donc utiliser une autre voie d'arrêt. Par exemple : (i) lors de la réalisation de l'effet (dans la StateTransitionProcedure de l'opération, ou bien dans le démon sur le degré de progression), programmer un événement daté du jour 335, (ii) avoir attaché à cet événement un processus qui teste si le degré de progression de l'opération est inférieur à 1 et qui arrête la simulation si c'est le cas, avec un message de justification pour l'utilisateur. DIESE propose des services pour accéder au plan, le parcourir pour accéder à l'activité primitive, et pointer sur l'opération en cours dans l'activité.

3.1 Les activités **sur le blé**

3.1.1 Désherbage d'automne

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Activity', puis '... Primitive'. Dans la fenêtre de choix, sélectionner DesherbageAutomne, puis cliquer sur 'Modifier'.

- Dans l'onglet 'Méthodes héritées', cocher le bouton 'Nom automatique' de la ligne 'OpeningPredicate'. Dans la fenêtre d'édition, remplacer le code généré par Solfege par le § AXII_E3.1.1a du fichier ressources_FCT.txt. On transforme la valeur courante de l'horloge en une structure de date tm du langage C, et on exploite son attribut dont la valeur est le jour julien pour cette date. On renvoie 'vrai' (par convention avec le moteur de simulation pour signifier une ouverture possible) si ce jour est postérieur à la spécification.

Remarque : voir note de bas de page[12].

- Sauvegarder le contenu de la fenêtre, quitter l'éditeur.

- Pour la condition de fermeture, la remarque faite ci-dessus sur la fermeture rend inopérant cet ajout. Cependant, si on voulait donner un type différent à l'opération, le § AXII_E3.1.1b du fichier ressources_FCT.txt donne le contenu que pourrait avoir ce prédicat (ClosingPredicate).

- Valider l'activité par 'Valider'.

3.1.2 Fertilisation

[12] Le service DescribedEntity est appliqué à l'objet pM, qui est la méthode (au sens de l'ontologie) dont le prédicat est l'attribut-corps. La transmission de cet objet méthode à la fonction qui en est le corps, dans la position d'un argument de la fonction, est une convention passée entre DIESE et de développeur de l'application. Plus précisément, lorsque le moteur de DIESE rencontre une instruction d'exécution d'une méthode (par exemple, OpeningPredicate de DesherbageAutomne), il lance l'exécution de la fonction-corps (desherbageAutomne_openingPredicate_Body) de la méthode, après avoir placé la méthode elle-même en argument de la fonction-corps. Il faut donc que le code de l'application respecte cette syntaxe d'appel conventionnelle.

- Dans la fenêtre de choix des activités primitives, sélectionner ApportAzote, puis cliquer sur 'Modifier'.

3.1.2.1 Le prédicat d'ouverture

- Dans l'onglet 'Méthodes héritées', cocher le bouton 'Nom automatique' de la ligne 'OpeningPredicate'. Dans la fenêtre d'édition, remplacer le code généré par Solfege par le § AXII_E3.1.2.1 du fichier ressources_FCT.txt.

On anticipe dans ce code une option sur la structure du plan global : il n'y aura pas, dans le plan déclaré en début de simulation, autant d'instances de la primitive ApportAzote que d'apports prévus. Au contraire, il n'y en aura qu'une, qu'on remettra en jeu (situation WAITING) autant de fois que désiré en cours de simulation. Cette option augmente la flexibilité du plan. Puisqu'il n'y aura qu'une instance d'ApportAzote, le corps unique du prédicat d'ouverture doit envisager les cas des différents apports :c'est le rôle de la structure C 'if(premier apport)/else ...' visible dans le code.

Comment le code peut-il établir le numéro d'ordre de l'apport ? L'anticipation définie et justifiée ci-dessus amène à placer l'unique instance de ApportAzote comme élément d'une activité agrégée de type 'itération', qui sera spécifiée dans un Acte ultérieur. On sait qu'une telle classe possède un descripteur dont la valeur est augmentée de 1 chaque fois que l'activité-fille passe à la valeur CLOSED. Ce descripteur, ReplicationCounter, indique donc le numéro d'ordre qui nous est utile. Sa valeur est captée en début du code du prédicat d'ouverture de ApportAzote. Si la valeur est 0, on teste l'ouverture du premier apport.

Pour le premier apport, le test d'ouverture repose sur une comparaison de jours juliens, comme dans les activités déjà développées. Cependant, la valeur seuil 53 est ambigüe : s'agit-il du 53ème jour de l'année de semis ou de l'année de récolte. Le seconde option est naturellement la bonne. Or un test sans précaution tel que 'dateCourante >= 53' sera vérifié dès le semis (jour julien toujours supérieur à au moins 200), et l'apport d'azote 'ouvert' dès cette date. On ne compare à 53 que si on est en année de récolte, c'est-à-dire si on est avant juillet.

Pour les apports suivants, (partie 'else'), le test repose sur l'atteinte de stades phénologiques. Il suffit de capter la valeur du stade en cours pour décider l'ouverture.
Le stade en cours, c'est la valeur entière du descripteur CurrentElementIndex le l'objet Age valeur du descripteur AttributAge du blé (Etapes 1 et 2.7 de l'Acte VI). On la compare avec un élément de l'énumération des symboles des stades, créée à l'Etape 3 de l'Acte VI.
Le blé, c'est la valeur du descripteur CouvertVegetal de l'objet Parcelle (Etape 2 de l'Acte II).
La parcelle, c'est par construction l'unique élément de la valeur du descripteur PreExpandedList de l'objet opéré de l'activité (Etape 8.1 de l'Acte IX).
L'objet opéré, c'est la valeur du descripteur prédéfini OperatedObjectSpecAttribute de l'activité, valué initialement dans le constructeur de l'activité (à l'Etape 5.2.4 de l'Acte VIII), puis surchargé par le démon sur UniteDeGestion (Etapes 8.1 et 8.2 de l'Acte IX).
Ce schéma de navigation se rencontre fréquemment dans les applications.

- Sauvegarder le contenu de la fenêtre, quitter l'éditeur.

3.1.2.2 L'ouverture au plus tard

- Dans l'onglet 'Méthodes héritées', cocher le bouton 'Nom automatique' de la ligne 'MaxBegStatePredicate'. Dans la fenêtre d'édition, remplacer le code généré par Solfege par le § AXII_E3.1.2.2 du fichier ressources_FCT.txt. Ce code appelle le commentaire complémentaire de ceux pour le prédicat d'ouverture :

On veut abandonner l'apport d'azote au-delà d'un délai après l'apparition d'un stade. Il faut donc avoir capturer le jour d'atteinte du stade. C'est la raison de la présence de JourJulienQuandStadeAtteint parmi les descripteurs de l'objet Stade, qu'on value naturellement dans le processus de développement, quand l'atteint est détectée (Etape 3 de l'Acte VI).

- Sauvegarder le contenu de la fenêtre, quitter l'éditeur. Valider l'activité par 'valider'.

3.1.3 Traitement

Contrairement à l'option prise pour la série d'apports d'azote, on considère que le nombre de traitements est toujours de deux, sans besoin de flexibilité. On anticipe donc ici qu'on va créer deux instances de TraitementFongicideBle, et que ces deux instances seront des éléments d'une activité agrégée de type 'séquence'. Lors de leur instanciation, on dotera ces deux activités d'un prédicat d'ouverture codé en accord avec la spécification posée en introduction de cette Etape 3. On fera de même pour la fonction de date de début au plus tard.

En réalité, on surchargera le corps de la méthode OpeningPredicate (puis MaxBegStatePredicate) avec une fonction qu'on va écrire. Mais alors que c'est Solfege qui génère le code du surchargement lorsqu'on passe par l'interface de l'onglet 'Méthodes héritées', c'est le programmeur du constructeur de l'activité agrégée qui va explicitement donner l'instruction du surchargement juste après l'instanciation de chaque TraitementFongicideBle.

Il n'y a donc pas lieu de modifier l'activité TraitementFongicideBle à ce stade.

3.1.4 Récolte

- Dans la fenêtre de choix des activités primitives, sélectionner MoissonBle, puis cliquer sur 'Modifier'.

- Dans l'onglet 'Méthodes héritées', cocher le bouton 'Nom automatique' de la ligne 'OpeningPredicate'. Dans la fenêtre

d'édition, remplacer le code généré par Solfege par le § AXII_E3.1.4 du fichier ressources_FCT.txt. Ce code n'appelle pas commentaires particuliers, au-delà de ceux faits pour les autres activités.

3.2 L'activité de **coupe de la luzerne**

Avec la même justification de flexibilité du plan que pour la fertilisation, on décide de gérer ces coupes par une activité agrégée d'itération. Mais alors que l'ouverture d'un apport d'azote a été codée dans le prédicat d'ouverture de l'activité itérée, on choisit pour la luzerne une autre localisation de la connaissance : la connaissance pour la première coupe sera installée dans le prédicat d'ouverture de l'itération ; la connaissance sur les suivantes sera installée dans sa méthode prédéfinie en charge de relancer l'examen l'activité itérée, une fois fermée. Ce choix est en partie arbitraire, mais l'uniformité du critère d'ouverture des coupes accrédite l'idée qu'il s'agit d'une connaissance sur l'itération elle-même.

Au niveau de cette activité primitive, on ne devrait donc installer que les conditions de fermeture. La remarque faite sur la fermeture en introduction de cette étape rend inopérante cette action. Si on voulait donner un type différent à l'opération, le § AXII_E3.2 du fichier ressources_FCT.txt donne le contenu que pourrait avoir ce prédicat.

3.3 L'activité d'**apport de fourrage au troupeau**

Deux schémas sont envisageables pour la conduite de l'affouragement.

a) (i) Repérer dynamiquement le prochain instant de passage de la croissance nette en dessous du seuil ; (ii) Repérer de même le prochain passage au-dessus du seuil ; (iii) entre les deux seulement, itérer une activité d'affouragement au pas de 5 jours ; (iv) itérer le triplet {(i) (ii) (iii)} sur la période simulée

b) Itérer une tentative d'affouragement chaque jour de la période simulée : on opère seulement si la croissance nette est en dessous du seuil, et qu'on est au moins 5 jours après le dernier apport. L'activité itérée (affouragement) doit alors être optionnelle pour que une non-réalisation (4 jours sur 5 en période de croissance faible) n'entraîne pas l'échec du plan.

On choisit, arbitrairement, le schéma b).

3.3.1 Le prédicat d'ouverture

- Dans la fenêtre de choix des activités primitives, sélectionner Affouragement, puis cliquer sur 'Modifier'.

- Dans l'onglet 'Méthodes héritées', cocher le bouton 'Nom automatique' de la ligne 'OpeningPredicate'. Dans la fenêtre d'édition, remplacer le code généré par Solfege par le § AXII_E3.3.1 du fichier ressources_FCT.txt. On note que :
  . le code passe le l'activité itérée à l'itération via l'activité 'option' et donc deux appels au service GetSuperSet.
  . il faudra désigner le troupeau comme valeur de UniteDeGestion de l'itération
  . il faudra doter le troupeau d'un descripteur DateDernierApport (valué dans ce prédicat si l'ouverture est décidée)
  . on value le descripteur prédéfini MinBegDate de l'activité itérée, pour exploitation dans le prédicat d'ouverture au plus tard (voir ci-dessous).

3.3.1 L'ouverture au plus tard

La condition d'ouverture au plus tard, si elle st exprimée, sert à abandonner la mise en œuvre de l'activité quand la condition est rencontrée. Et on rappelle que cet abandon est sans échec pour le plan si l'activité est optionnelle. Quand la condition d'ouverture n'est pas satisfaite, on veut fermer tout de suite l'affouragement pour que l'itération qui l'encapsule remette en examen un nouvel affouragement le lendemain. Il y a deux techniques alternatives pour réaliser cela :

- on code un MaxBegStatePredicate qui renvoie 'vrai' si le prédicat d'ouverture a échoué. Le § AXII_E3.3.2 du fichier ressources_FCT.txt présente un tel code.

- on exploite le descripteur IsOneShot de la classe prédéfinie ActivityOptional. Quand sa valeur est 'vrai' (ou 1), alors le moteur ne teste qu'une seule fois le prédicat d'ouverture. Si ce dernier renvoie 'faux', alors l'activité optionnelle est abandonnée. C'est cette option commode qu'on va choisir, et on se le rappellera lors du développement de l'activité-option. Dans le prédicat d'ouverture, l'affectation de l'instant courant à MinBegDate peut être retirée du code.

Sauvegarder la base !

Etape 4 : Vérification du code

Générer 'tout les sources' et lancer la compilation par le moyen choisi.

La compilation du fichier UserActivity.cc échoue parce que le symbole DATE_DERNIER_APPORT n'a pas été déclaré. On se rappelle en effet que le descripteur doit encore être créé et attaché à la classe Troupeau (voir Etape 3.3.1).

4.1 Le descripteur **DateDernierApport**

- Dans la fenêtre de choix des descripteurs constants, cliquer sur 'Créer'.dans le champ 'Name' de l'onglet 'Identité' de la fenêtre de modification, taper DateDernierApport, puis confirmer la 'SubClass' CONSTANT et le 'Type' INT. Taper un commentaire tel que 'la date (clock) du dernier apport de fourrage au troupeau'. Valider par le bouton 'Valider'.

- Dans la fenêtre de choix des entités, sélectionner Troupeau, puis cliquer sur 'Modifier'. Dans l'onglet 'Descripteurs

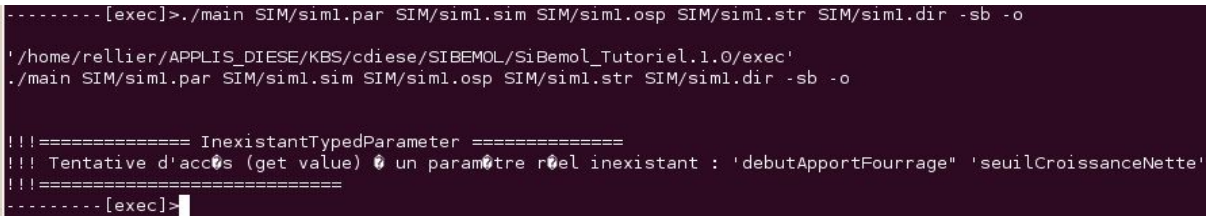
propres', zone des descripteurs constants, ajouter DateDernierApport. Valider par le bouton 'Valider'.

4.2 Sauvegarder la base, générer à nouveau tout le code et **recompiler**

Le code est désormais sans erreur lexicale ou grammaticale. En tentant l'exécution du code par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

... on obtient la trace ci-dessous : ⏭



.

- Dans le répertoire des données, éditer le fichier sim1.par. Insérer le § AXII_E4.2 du fichier ressources_SIM.txt dans la section 'CONDUITE' du fichier des paramètres[13]. Sauvegarder ce nouveau contenu.

- Une nouvelle exécution provoque alors la même trace, attendue, qu'à l'étape 4.3 de l'Acte XI.

[13] On rappelle ici qu'un service de la famille Get[Int Real String]ParameterValue renvoie la valeur de l'attribut 'valeur' d'une instance existante de la classe Parameter : celle dont deux autres attributs à valeur de clés ont pour valeur les arguments du service. C'est lors de la lecture du fichier des paramètres (ici sim1.par) que sont créées les instances de Parameter. Si la ligne contenant les chaînes "debutApportFourrage" et "seuilCroissanceNette" est absente, l'instance correspondante n'est pas créée, et le service GetRealParameterValue, argumenté avec ces chaînes, ne peut pas la trouver. D'où le message lisible dans la trace.

Les activités primitives correspondent à des actions techniques élémentaires organisées dans le temps, respectant ainsi notamment des contraintes de séquencement ou de mises en œuvre concurrentes. Les "opérateurs d'organisation" que nous avons rencontrés, dans l'énoncé initial du problème ou résultant de choix de modélisation, sont les suivants :
- certaines activités sur le blé sont en séquence : semis, désherbage, récolte ;
- les deux traitements fongicides sont en séquence ;
- les apports d'azote, les récoltes de luzerne et les affouragements sont itérés ;
- les apports d'azote et les traitements sont en concurrence, dans une période située, dans la séquence sur le blé, entre le désherbage et la moisson ;
- la conduite du blé, les coupes de luzerne et l'alimentation complémentaire du troupeau sont en concurrence.

Etape 1 : La conduite du blé

On continue à développer d'abord les éléments de plus bas niveau d'intégration, pour les agréger ensuite. Aussi, on commence par créer les classes pour la séquence de traitement fongicides et l'itération d'apports d'azote. On a crée, à l'Etape 1 de l'Acte IX, l'activité ItineraireBle pour y réaliser l'agrégation.

1.1 La séquence de traitements fongicides

1.1.1 Le constructeur de l'agrégation

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Activity', et '...Non Primitive". Cliquer sur 'Créer'.

- Dans l'onglet 'Identité' de la fenêtre de modification, taper SequenceTraitementsFongicides dans le champ 'Name', puis choisir ACTIVITY_BEFORE dans le menu du champ 'SetParentClassId'. Dans le menu du champ 'ElementClassId', choisir TRAITEMENT_FONGICIDE_BLE.

- Cliquer sur le bouton 'Coder' de la ligne 'Constructor'. Dans le fenêtre d'édition, après la ligne "DEBUT CODE SPECIFIQUE ...", coller le § AXIII_E1.1.1 du fichier ressources_FCT.txt. On y lit les instructions d'instanciation des deux activités primitives TraitementFoncicideBle, et l'ajout de ces instances comme éléments de l'activité agrégée en construction. Par convention, l'ordre d'ajout respecte l'ordre chronologique de mise en œuvre.

Comme anticipé dans l'Etape 3.1.3 de l'Acte XII, on dote maintenant chacune de ces deux activités d'un corps pour les deux méthodes prédéfinies OpeningPredicate et MaxBegStatePredicate. Ces quatre fonctions doivent maintenant être créées. Mais sauvegarder d'abord le contenu du constructeur, quitter l'éditeur et valider l'activité agrégée par 'Valider'.

1.1.2 Les corps des prédicats régissant l'ouverture

- Dans le menu 'Développement', item 'Function', choisir 'Fonctions globales'.

- Un menu de choix apparaît, encore vide. Cliquer sur 'Créer'. Dans le champ unique de la nouvelle fenêtre de saisie, taper traitementFongicide1_openingPredicate, puis cliquer sur 'Ok'. Dans la fenêtre d'édition ouverte, remplacer l'amorce de code générée par Solfege par le § AXIII_E1.1.2a du fichier ressources_FCT.txt. Remarquer que le type de la fonction devient 'bool', puisque le prédicat doit, par convention, renvoyer 'vrai' ou 'faux'. Le test s'assure d'abord qu'on est entré dans l'année de récolte (avec une date courante antérieure au 31 juillet, arbitrairement), puis il renvoie 'vrai' dès qu'on a atteint le jour julien 110.

- Cliquer sur 'Créer' dans la même fenêtre de choix. Taper traitementFongicide1_maxBegStatePredicate, puis cliquer sur 'Ok'. Dans la fenêtre d'édition , coller le § AXIII_E1.1.2b du fichier ressources_FCT.txt. Le code est identique à celui de l'OpeningPredicate, avec une date seuil augmentée de 5 jours.

- Cliquer sur 'Créer' dans la même fenêtre de choix. Taper traitementFongicide2_openingPredicate, puis cliquer sur 'Ok'. Dans la fenêtre d'édition , coller le § AXIII_E1.1.2c du fichier ressources_FCT.txt. On récupère le stade phénologique courant, et on renvoie vrai si c'est le stade Floraison. Le test étant fait tous les jours, ce prédicat renverra 'vrai' dès que le stade sera atteint. Se rappeler ici que la progression des stades est un processus de priorité plus forte que l'examen du plan d'activité, donc réalisée avant pour un jour donné.

- Cliquer sur 'Créer' dans la même fenêtre de choix. Taper traitementFongicide2_maxBegStatePredicate, puis cliquer sur 'Ok'. Dans la fenêtre d'édition , coller le § AXIII_E1.1.2d du fichier ressources_FCT.txt. Le code récupère aussi le stade courant. Si c'est la Floraison, alors ce second traitement doit être démarré dans les 5 jours après que le stade soit apparu, pour qu'il ne soit pas abandonné, mettant ainsi le plan en échec. On récupère à cet effet la valeur du descripteur JourJulienQuandStadeAtteint, auquel il suffit d'ajouter 5 pour informer le test.

1.2 L'itération d'apports d'azote

1.2.1 Le constructeur de l'agrégation

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Activity', et'...Non Primitive". Cliquer sur 'Créer'.

- Dans l'onglet 'Identité' de la fenêtre de modification, taper IterationApportsAzote dans le champ 'Name', puis choisir

ACTIVITY_ITERATION, symbole de la classe prédéfinie ActivityIteration, dans le menu du champ 'SetParentClassId'. Dans le menu du champ 'ElementClassId', choisir APPORT_AZOTE.

- Dans l'onglet 'Descripteurs hérités', zone des descripteurs variables, choisir MinNumberOfReplications et donner la valeur 3 avant de cliquer sur 'Ajout'. Faire de même pour MaxNumberOfReplications. Ainsi, le nombre d'apports d'azote est fixé à 3, sans flexibilité.

- Cliquer sur le bouton 'Coder' de la ligne 'Constructor'. Dans le fenêtre d'édition, après la ligne "DEBUT CODE SPECIFIQUE ...", coller le § AXIII_E1.2.1 du fichier ressources_FCT.txt. On y lit l'instruction d'instanciation de l'activité primitive ApportAzote, et l'ajout de cette instance comme élément unique et permanent, par définition ontologique, de l'activité agrégée en construction. Cette instance sera ouverte (puis fermée) trois fois avant la fermeture de l'itération elle-même.

- Sauvegarder le contenu du constructeur, quitter l'éditeur et valider l'activité agrégée par 'Valider'.

1.2.2 Le mécanisme d'itération

On a équipé l'activité primitive ApportAzote d'un prédicat d'ouverture, à l'étape 3.1.2.1de l'Acte XII. Ce prédicat tient compte du nombre exact d'apports (3), et spécifie la condition d'ouverture pour chacun d'eux, y compris le premier, donc. On sait qu'en l'absence de spécification d'ouverture d'une activité agrégée, le moteur de simulation cherche dans les activités-filles un motif pertinent d'ouverture. Par exemple before(A, B) sera ouverte si A peut l'être. De même iterate(A) sera ouverte si A peut l'être. Il n'est donc pas nécessaire de spécifier une condition d'ouverture pour IterationApportsAzote, et notamment son OpeningPredicate.

Une fois fermée l'activité primitive après la première itération (i.e. après la première mise en œuvre complète de l'opération FertiliserBle), le moteur remet l'instance dans la situation WAITING, puis teste à nouveau périodiquement la condition d'ouverture, dans le cas '2ème apport'. Idem après le second apport, pour ouvrir le troisième. Il n'est donc pas nécessaire de spécifier, par un corps particulier de UpdateReactivatedSon, la fenêtre d'ouverture de l'activitè-fille de IterationApportsAzote.

1.3 L'assemblage les éléments de l'itinéraire de conduite du blé

- Dans la fenêtre de choix des activités non primitives, sélectionner ItineraireBle, puis cliquer sur 'Modifier'.

- Cliquer sur le bouton 'Modifier' de la ligne 'Constructor'. Dans le fenêtre d'édition, supprimer les deux lignes contenant respectivement '/*' et */', pour mettre en jeu ce paragraphe. On ajoute ainsi trois éléments à l'itinéraire : le désherbage et la moisson, et, entre les deux, la mise en œuvre conjointe de la fertilisation et des traitements, sans contrainte temporelle. La classe prédéfinie dont la conception et les services assurent cette concurrence est ActivityConjunction.

- Sauvegarder le contenu du constructeur, quitter l'éditeur et valider l'activité agrégée par 'Valider'.

1.4 La validation du codage de la conduite du blé

- Générer 'tous les sources', sauvegarder la base, et lancer la compilation par le moyen choisi.

- A l'étape 4.3 de l'Acte XI, on a enlevé paul du pool du personnel (fichier systemeOperant.inc), et on a activé l'option MAX pour la ModaliteDeuxAgents (systemeDecisionnel.inc). Dans cette situation, l'exécution de la simulation par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

... génère une trace dans laquelle on lit que :
. le semis est effectué par jean en 10 jours entre le 16/10 et le 26/10 (pas de travail le dimanche)
. le désherbage est effectué par jean en 3 jours entre le 16/11 et le 18/10
. chaque apport d'azote dure 3 jours ; les dates de début sont les 23/02, 02/04 et 05/05
. chaque traitement dure 3 jours ; les dates de début sont les 21/04 et 24/05
. la récolte n'est pas faite !

- La récolte mobilise en effet deux outils, la MB et une remorque tractée, disponibles. Mais aussi deux unités de MO pour servir ces outils. Or, une seule existe. Si on remet paul dans le pool de personnel, la trace montre que la récolte est faite par jean et paul en 14 jours entre le 31/07 et le 15/08. Par ailleurs, la durée du semis, effectué désormais par deux agents de par l'option MAX, est réduite de moitié.

- Si on enlève un outil de récolte du pool de matériels (et comme composant de la ressource agrégée de classe UniteMaterielRecolte), à nouveau la récolte n'est pas faite. Le moteur signale d'ailleurs que la ressource agrégée est incomplètement construite et la simulation s'arrête. Remettre l'outil enlevé dans son pool et dans la ressource agrégée.

Etape 2 : Les coupes de luzerne

2.1 La spécification de l'itération de coupes

A l'étape 3.2 de l'Acte XII, on a décidé de gérer ces coupes par une activité agrégée d'itération. Et on a opté pour installer la connaissance pour l'ouverture de la première coupe dans le prédicat d'ouverture de l'itération, et celles sur les

coupes suivantes dans sa méthode prédéfinie en charge de relancer l'examen l'activité itérée, une fois fermée.

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Activity', puis '...Non Primitive". Cliquer sur 'Créer' dans la fenêtre de choix.

- Dans l'onglet 'Identité' de la fenêtre de modification, taper **IterationRecolteLuzerne** dans le champ 'Name', puis choisir ACTIVITY_ITERATION dans le menu du champ 'SetParentClassId'. Dans le menu du champ 'ElementClassId', choisir RECOLTE_LUZERNE.

- Dans l'onglet 'Descripteurs hérités', zone des descripteurs variables, choisir **MinNumberOfReplications** et donner la valeur 3 avant de cliquer sur 'Ajout'. Faire de même pour MaxNumberOfReplications. Ainsi, le nombre de coupes est fixé à 3, sans flexibilité.

- Dans l'onglet 'Descripteurs propres', zone des descripteurs variables, ajouter le descripteur **UniteDeGestion**.

- Dans l'onglet 'Moniteurs', et dans la zone 'SetMonitorToDescriptor', taper UniteDeGestion dans le champ à droite du bouton 'Ajout' (parce que, sans fermeture eu nouvelle ouverture de la fenêtre, le descripteur récemment créé n'est pas encore dans la liste). Dans le menu situé juste en dessous, choisir le moniteur **UniteDeGestionMonitor**, puis cliquer sur 'Ajout'.

- Dans l'onglet 'Méthodes héritées', cocher la case 'Nom automatique' de la ligne '**OpeningPredicate**'. Dans la fenêtre d'édition, remplacer le code proposé par le § AXIII_E2.1a du fichier ressources_FCT.txt.

Le test d'ouverture repose sur une comparaison de dates calendaires. Cependant, la date du 1er juin est ambigüe : s'agit-il du 1er juin de l'année de semis ou de l'année de récolte. Le seconde option est naturellement la bonne. Or un test sans précaution tel que 'dateCourante >= 1er juin' sera vérifié dès le semis (dont la date est toujours postérieure), et l'itération de coupes 'ouverte' dès cette date. On introduit donc un schéma de programmation fréquent dans cette situation : Lorsque la date courante est postérieure à une date bien choisie (ici 31 juillet), alors on précise que le 1er juin visé est celui de l'année suivante, en ajoutant 1 au millésime de la structure tm correspondant à date courante.

- Dans le même onglet, cocher la case 'Nom automatique' de la ligne '**UpdateReactivatedSon**'. Dans la fenêtre d'édition, remplacer le code proposé par le § AXIII_E2.1b du fichier ressources_FCT.txt.

La date de début de la coupe qui vient de se terminer est la valeur de son descripteur prédéfini BegDate. Il suffit d'ajouter le délai spécifié entre deux coupes pour avoir la date minimale (MinBegDate) de la prochaine coupe (dont on rappelle qu'elle est représentée par la même instance de CoupeLuzerne, ici pSon). On value d'abord MaxBegDate (complétant ainsi la fenêtre d'ouverture), parce l'inverse produirait, dès la valuation de MinBegDate, un intervalle dont la borne sup. serait inférieure à la borne inf..

- Dans l'onglet 'Identité', cliquer sur le bouton 'Coder' de la ligne 'Constructor'. Dans le fenêtre d'édition, après la ligne "DEBUT CODE SPECIFIQUE ...", coller le § AXIII_E2.1c du fichier ressources_FCT.txt. On y lit l'instruction d'instanciation de l'activité primitive RecolteLuzerne, et l'ajout de cette instance comme élément unique et permanent de l'activité agrégée en construction.

- Sauvegarder le contenu du constructeur, quitter l'éditeur et valider l'activité agrégée par 'Valider'.

2.2 La validation du codage des coupes de luzerne

- Générer 'tous les sources', sauvegarder la base, et lancer la compilation par le moyen choisi.

- Dans le répertoire des données, éditer le fichier systemeDecisionnel.inc.

. insérer le § AXIII_E2.2a juste après le paragraphe sur l'itinéraire de blé (`+I itineraireBle ...;`).

. insérer le § AXIII_E2.2b (ligne unique) juste après la ligne d'ajout de l'itinéraire du blé dans la conduite globale (`<- E <I> <, maConduiteBle>;`).

- Dans le répertoire des données, éditer le fichier sim1.par.

. insérer le § AXIII_E2.2c juste après le paragraphe sur les paramètres du blé.

- Dans cette configuration, l'exécution de la simulation par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

... génère une trace dans laquelle on lit que :
. le semis est effectué par jean et paul en 5 jours entre le 16/10 et le 20/10 (pas de travail le dimanche)
. le désherbage est effectué par paul en 3 jours entre le 16/11 et le 18/10
. chaque apport d'azote dure 3 jours ; les dates de début sont les 23/02, 02/04 et 05/05
. chaque traitement dure 3 jours ; les dates de début sont les 21/04 et 24/05
. la récolte est faite par jean et paul en 14 jours entre le 31/07 et le 15/08.
. les trois coupes de luzerne durent 7 jours : les dates de début sont les 01/06, 13/07 et 24/08.

Etape 3 : L'alimentation complémentaire du troupeau

3.1 La spécification de l'itération d'affouragement

A l'étape 3.3 de l'Acte XII, on a décidé de de gérer ces coupes par une activité agrégée d'itération. Et on a opté pour installer la connaissance pour l'ouverture des affouragements dans l'activité primitive itérée (Affouragement).

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'Activity', puis '...Non Primitive". Cliquer sur 'Créer' dans la fenêtre de choix.

- Dans l'onglet 'Identité' de la fenêtre de modification, taper **AlimentationComplementaireTroupeau** dans le champ 'Name', puis choisir ACTIVITY_ITERATION dans le menu du champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs propres', zone des descripteurs variables, ajouter le descripteur **UniteDeGestion**.

- Dans l'onglet 'Moniteurs', et dans la zone 'SetMonitorToDescriptor', taper UniteDeGestion dans le champ à droite du bouton 'Ajout' (parce que, sans fermeture eu nouvelle ouverture de la fenêtre, le descripteur récemment créé n'est pas encore dans la liste). Dans le menu situé juste en dessous, choisir le moniteur **UniteDeGestionMonitor**, puis cliquer sur 'Ajout'.

- Dans l'onglet 'Identité', cliquer sur 'Coder' dans la ligne du 'Constructor Ajouter le § AXIII_E3.1 du fichier ressources_FCT.txt après la ligne "DEBUT CODE SPECIFIQUE ...". L'activité primitive est enveloppée d'une activité-option, qu'on déclare "à un coup", pour permettre l'abandon de l'activité optionnelle dès le premier échec de son prédicat d'ouverture (revoir l'Etape 3.3.1 de l'Acte XII).

3.2 La validation de l'alimentation complémentaire du troupeau

- Générer 'tous les sources', sauvegarder la base, et lancer la compilation par le moyen choisi.

- Dans le répertoire des données, éditer le fichier systemeDecisionnel.inc.

. insérer le § AXIII_E3.2a après le § sur les coupes de luzerne (`+I iterationRecolteLuzerne ...;`).

. insérer le § AXIII_E3.2b (ligne unique) juste après la ligne d'ajout de l'itération de coupes de luzerne dans la conduite globale (`<- E <I> <, maConduiteLuzerne>;`).

- Dans cette configuration, l'exécution de la simulation par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

... génère une trace dont le début est tel que ci-dessous : ⏭

```
16/10/2000 historique IT sur monBle : S_*_*_*_*
16/10/2000 monBle a atteint le stade APRES_SEMIS
LUN 16/10/2000 'semerBle_83' sur 'parcelleBle' -> 0.200 (jean paul)
MAR 17/10/2000 'semerBle_83' sur 'parcelleBle' -> 0.400 (jean paul)
MER 18/10/2000 'semerBle_83' sur 'parcelleBle' -> 0.600 (jean paul)
JEU 19/10/2000 'semerBle_83' sur 'parcelleBle' -> 0.800 (jean paul)
VEN 20/10/2000 'apporterFourrage_1213' sur 'monTroupeau' -> 1.000       (paul)
VEN 20/10/2000 'semerBle_83' sur 'parcelleBle' -> 1.000 (jean paul)
MER 25/10/2000 'apporterFourrage_1731' sur 'monTroupeau' -> 1.000       (paul)
```

Le premier apport de fourrage est daté du 20/10. En effet, par défaut, l descripteur DateDernierApport a la valeur 0. Comme codé à l'Etape 3.3 de l'Acte XII, il faut attendre 5j*24h pour autoriser l'ouverture de l'activité. En complément, on peut vérifier dans le fichier DISPO_HERBE.txt que la valeur de croissance nette (0.0) autorise l'apport. L'apport suivant est bien effectué cinq jours après.

Au 20/10, l'apport de fourrage mobilise le seul AgentCompetentVaches, et le semis, alloué avec l'opérateur MAX récupère deux agents. Il apparaît cependant anormal que paul soit mobilisé en même temps par deux activités qui l'amènent en deux lieux différents ! L'Acte suivant installera une contrainte empêchant cette situation.

Les apports de fourrage s'arrêtent à partir du 29/03, date à laquelle la croissance de l'herbe repasse au-dessus du seuil fixé (10 g/m2, dans sim1.par), puis sont repris à partir du 27/06. C'est le déroulé d'actions attendu.

ACTE XIV : Expression de contraintes sur les ressources

La trace des actions obtenue à la fin de l'Acte précédent était inattendue sur un point : une unité de MO était occupée en même temps en deux lieux différents. Cette situation, jugée impossible par le développeur du simulateur, doit être déclarée comme telle au moteur de simulation, pour qu'il l'élimine automatiquement quand il la rencontre. Une telle déclaration prend la forme d'une contrainte.

L'ontologie reconnaît trois types de contraintes :
- celles qui limitent le partage de l'usage d'une ressource entre plusieurs opérations, ou qui limitent le co-engagement simultané d'une ressource avec plusieurs autres ressources. Une telle contrainte pourra empêcher paul de se partager entre plusieurs tâches.
- celles qui déclarent indésirable une conformation particulière d'une activité : par exemple, la conjonction de deux valeurs de descripteurs (spécification d'objet opéré et de 'performer', par exemple), ou la conjonction d'un type pour une activité-fille avec le type de l'activité-mère, etc.
- celles qui déclarent indésirable une conformation particulière d'un jeu d'activité : par exemple, la mise en œuvre conjointe de deux activités dont l'une possède telle propriété et l'autre telle autre propriété. Avec une telle contrainte aussi, on peut empêcher qu'un jeu d'activités contienne deux activités dont les ensembles de 'performers' contiennent au moins une unité de MO commune. On pressent que le codage de la reconnaissance de cette situation serait plus lourd (parce que combinatoire) que l'expression d'une contrainte du premier type.

Ces contraintes expriment une situation impossible ou indésirable. Elles sont dotées d'un prédicat qui limitent leur mise en jeu. Par exemple, telle contrainte de partageabilité ne sera mise en jeu que dans une période spécifiée. Telle contrainte de conformation d'une activité ne sera mise en jeu que si une option de conduite le demande, etc.

Etape 1 : La mono-occupation exprimée par une contrainte de partageabilité

On souhaite exprimer qu'il est impossible qu'un agent, quel qu'il soit, soit occupé par deux opérations différentes. La construction de la contrainte respecte cette expression : on va associer le symbole de classe AGENT à la formule "plus de une opération". Plus précisément à un triplet {OPERATION, >, 1}, ou OPERATION est le symbole de la classe Operation. Le moteur de simulation sait qu'il doit rejeter cette association s'il la rencontre.

1.1 La situation inacceptable

-Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'InconsistencyCondition', puis 'OccurrenceDomain'. Dans la fenêtre de choix, encore vide, cliquer sur 'Créer'.

- Dans l'onglet 'Identité' de la fenêtre de modification, taper PluriOccupation dans le champ 'Name', puis choisir OCCURRENCE_DOMAIN dans le menu du champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', zone des descripteurs constants, surcharger les valeurs des descripteurs suivants, avec la valeur indiquée :
  . EntityIdAttribute, valeur : OPERATION
  . BinaryOperatorId, valeur : C_GT (pour "plus grand que")
  . AcceptabilityThreshold, valeur : 1

- Valider cette description par le bouton 'Valider'.

1.2 La situation est inacceptable **pour un agent**

-Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'InconsistencyCondition', puis 'ResourceSharingViolationCondition'. Dans la fenêtre de choix, encore vide, cliquer sur 'Créer'.

- Dans l'onglet 'Identité' de la fenêtre de modification, taper AgentMonoOccupe dans le champ 'Name', puis choisir RESOURCE_SHARING_VIOLATION_CONDITION dans le menu du champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', surcharger la valeur du descripteur variable suivant, avec la valeur indiquée :
  . ConstrainedEntityClassId, valeur : AGENT

Valider cette description sans quitter la fenêtre, par le bouton 'Appliquer'.

- De retour dans l'onglet 'Identité', cliquer sur le bouton 'Créer' de la ligne 'Constructor'. Dans la fenêtre d'édition, coller le § AXIV_E1.2 du fichier ressources_FCT.txt, juste après la ligne "DEBUT du code spécifique...". On associe ainsi la situation PluriOccupation à la classe Agent, déjà indiquée par le descripteur ConstrainedEntityClassId.

Noter que la PluriOccupation, instance de OccurrenceDomain, est dans notre cas, l'élément unique d'une conjonction. De manière générale, la situation inacceptable est la rencontre conjointe d'un ensemble de condition élémentaires.

Noter aussi qu'on n'a pas donné de corps particulier, dans l'onglet 'Méthodes héritées' à la méthode HoldingCondition : la contrainte développée s'applique dans tout l'espace des situations.

- Sauvegarder ce contenu du constructeur, quitter l'éditeur et valider cette description par le bouton 'Valider'.

Etape 2 : La mono-localisation exprimée par une contrainte de partageabilité

On souhaite exprimer qu'il est impossible qu'un agent, quel qu'il soit, soit présent simultanément sur plus d'un lieu, avec le rôle d'objet opéré. Pour la construction de cette contrainte, de même type que AgentMonoOccupe, on voudrait associer le symbole de classe AGENT à la formule "plus de un lieu". Il nous faut alors désigner la classe qui représente les lieux. Or, les objets opérés par les différentes activités sont soit des parcelles, soit le troupeau. Et on n'a pas imaginé que ces deux classes descendent (héritent) d'une même classe-mère, qui aurait pu s'appeler Lieu, avec le symbole LIEU.

Il ne serait pas trop tard pour le faire, pour ensuite poser le triplet {LIEU, >, 1}, et l'associer ensuite à la classe Agent pour définir une contrainte AgentMonoLocalise. Une autre voie est empruntée, permettant d'illustrer le rôle et le codage d'une HoldingCondition, ainsi que la manipulation des structures modélisant les allocations.

2.1 La situation inacceptable

- Nous allons utiliser la classe PluriOccupation, puisqu'elle est disponible. En toute rigueur, un triplet {OPERATION, >, 0} serait suffisant, parce que la multi-localisation est interdite même pour réaliser deux opérations du même type.

2.2 La situation est inacceptable **pour un agent**

-Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'InconsistencyCondition', puis 'ResourceSharingViolationCondition'. Dans la fenêtre de choix, encore vide, cliquer sur 'Créer'.

- Dans l'onglet 'Identité' de la fenêtre de modification, taper AgentMonoLocalise dans le champ 'Name', puis choisir RESOURCE_SHARING_VIOLATION_CONDITION dans le menu du champ 'SetParentClassId'.

- Dans l'onglet 'Descripteurs hérités', surcharger la valeur du descripteur variable suivant, avec la valeur indiquée :
  . ConstrainedEntityClassId, valeur : AGENT

Valider cette description sans quitter la fenêtre, par le bouton 'Appliquer'.

- De retour dans l'onglet 'Identité', cliquer sur le bouton 'Créer' de la ligne 'Constructor'. Dans la fenêtre d'édition, coller le § AXIV_E1.2 du fichier ressources_FCT.txt, le même que celui exploité à l'étape 1.2. Et cela juste après la ligne "DEBUT du code spécifique...". On associe ainsi encore la situation PluriOccupation à la classe Agent, déjà indiquée par le descripteur ConstrainedEntityClassId.

- Sauvegarder ce contenu du constructeur, quitter l'éditeur.

2.2 La situation est inacceptable **si elle implique deux lieux différents**

- Dans l'onglet 'Méthodes héritées', cocher la case 'Nom automatique' pour la 'HoldingCondition'. Cliquer sur 'Créer'.

- Dans la fenêtre d'édition, coller le § AXIV_E2.2du fichier ressources_FCT.txt. Ce code est exécuté à chaque tentative d'allocation d'une ressource à une activité : la variable locale pInst, valeur, par convention, de l'argument ReferenceEntityArgument. Cette ressource vient s'ajouter aux autres déjà allouées avec succès (i) à l'activité : l'ensemble est alors structure dans une instance de InstructionAllocation (la structure V de l'ontologie), et (ii) au jeu d'activités en cours d'allocation : l'ensemble est alors structuré dans une instance de InstructionSetAllocation (la structure U de l'ontologie). La présente contrainte de mono-localisation d'un agent doit inspecter l'allocation au jeu d'activités pour éventuellement trouver tel ou tel agent engagé (dans plusieurs activités) sur au moins deux lieux différents.

On récupère donc la structure U attachée au jeu : c'est la valeur de l'argument CandidateEntityValueArgument, par convention (variable locale pU_alloc). Cette structure a autant d'éléments que de ressources (et d'opérations) allouées au jeu. Chacun de ces éléments, instance pUI_i de la classe UsageItem, contient la ressource, disons Ri, et une série de CoengagementItem's pCEI_i_j, un par classe Cj de ressource avec laquelle Ri se trouve engagée dans le jeu. Dans un pCEI_i_j, la référence à la classe Cj est accompagnée de la liste de ses instances coengagées avec Ri dans le jeu. Il suffit donc de rechercher dans pU_alloc les UsageItem's dont Ri soit un Agent puis, dans ce pU_i, de chercher les CoengagementItem's dont Cj soit la classe Troupeau ou la classe Parcelle, accumulant au passage l'effectif de la liste d'instances rsc_i_j_k de Cj. Dès que le nombre cumulé atteint 2, on fait renvoyer 'vrai' à la HoldingCondition.

Seulement quand la HoldingCondition renvoie 'vrai', le moteur d'allocation teste si la situation inacceptable de PluriOccupation est rencontrée. C'est nécessairement vrai, puisque chaque lieu est l'objet d'une opération différente. La contrainte AgentMonoLocalise est donc violée, et la tentative d'allocation en cours (Ri dans le paragraphe ci-dessus) est abandonnée. Si une autre ressource est candidate, la procédure est relancée (commençant par le test de la HoldingCondition). Sinon, l'allocation de l'activité en cours d'examen (pInst) est en échec, et le jeu d'activité contenant pInst doit être réduit pour espérer atteindre une allocation complète sans échec.

- Sauvegarder ce contenu de la méthode, quitter l'éditeur et valider cette description par le bouton 'Valider'.

Etape 3 : La validation du codage de la mono-occupation et de la mono-localisation

- Générer 'tous les sources', sauvegarder la base, et lancer la compilation par le moyen choisi.

- Dans le répertoire des données, éditer le fichier systemeOperant.inc.

. insérer le § AXIV_E3a du fichier ressources_SIM.txt juste avant le § qui instancie le système opérant (`+ I operatingSystem pOS ...;`).

. insérer le § AXIV_E3b dans le § qui instancie le système opérant, juste après la ligne qui ajoute le composant lesPools (`<- C <I> <, lesPools>;`). Noter l'opérateur '<<' qui ajoute un élémernt de valeur à un descripteur dont la valeur est un ensemble.

- Cette configuration inhibe la contrainte de mono-localisation. Pour mettre alternativement en jeu la contrainte de mono-localisation, il suffit de déplacer les signes '//' du début de ligne. Dans les deux cas, l'exécution de la simulation par :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

... génère une trace dont le début peut présenter deux déroulés d'actions différents : ⏭

```
16/10/2000 historique IT sur monBle : S_*_*_*_*
16/10/2000 monBle a atteint le stade APRES_SEMIS
LUN 16/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.200 (jean paul)
MAR 17/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.400 (jean paul)
MER 18/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.600 (jean paul)
JEU 19/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.800 (jean paul)
VEN 20/10/2000 'semerBle_86' sur 'parcelleBle' -> 1.000 (jean paul)
SAM 21/10/2000 'apporterFourrage_1216' sur 'monTroupeau' -> 1.000       (paul)
MER 25/10/2000 'apporterFourrage_2498' sur 'monTroupeau' -> 1.000       (paul)
```

ou bien : ⏭

```
16/10/2000 historique IT sur monBle : S_*_*_*_*
16/10/2000 monBle a atteint le stade APRES_SEMIS
LUN 16/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.200 (jean paul)
MAR 17/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.400 (jean paul)
MER 18/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.600 (jean paul)
JEU 19/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.800 (jean paul)
VEN 20/10/2000 'apporterFourrage_1216' sur 'monTroupeau' -> 1.000       (paul)
VEN 20/10/2000 'semerBle_86' sur 'parcelleBle' -> 0.900 (jean)
SAM 21/10/2000 'semerBle_86' sur 'parcelleBle' -> 1.000 (jean paul)
MER 25/10/2000 'apporterFourrage_2616' sur 'monTroupeau' -> 1.000       (paul)
```

Dans le premier déroulé, les deux agents sont alloués au semis du blé, et l'apport de fourrage, bien que décidé, se retrouve sans ressource MO et repoussé au lendemain. L'apport suivant est bien réalisé le 25/10, 5 jours après l'ouverture au 20/10 du premier. Dans le second déroulé, l'agent compétent est alloué à l'apport de fourrage, qui peut être réalisé. Le semis, dont l'allocation se fait avec l'opérateur MAX, récupère l'autre agent resté disponible et peut être continué mais avec une moindre vitesse. Il faut encore travailler le lendemain pour l'achever. C'est ce dernier schéma qui respecte le mieux la priorité qu'on a souhaité donner à l'apport de fourrage dans l'énoncé du problème, comme rappelé à l'Etape 1.2 de l'Acte X.

Si on exécutait la simulation un grand nombre de fois, chacun des deux déroulés adviendrait en nombre égal. En effet, le 20/10, deux alternatives valides résultent de la procédure d'allocation : (SemisBle (jean paul)) et (ApportFourrage (paul), SemisBle (jean)). On sait alors que le moteur de simulation choisit une alternative au hasard, en l'absence de spécification particulière posée dans la base de connaissances. Le tirage étant fait uniformément, la probabilité de "sortie" de chaque déroulé est 1/2.

Comme évoqué ci-dessus, le développeur peut indiquer au moteur de simulation une manière de choisir entre au moins deux alternatives d'action, en fin de procédure d'allocation des ressources. Deux localisations sont possibles pour cette connaissance. Soit dans la classe du gestionnaire (dans notre cas MonManager, *cf.* Etape 5.2.3 de l'Acte VIII). Ce sera l'option choisie. Soit dans le bloc 'activités-ressources' gérant les activités en jeu (cf. Etape 3 de l'Acte IX)[14]. Dans les deux cas, une méthode prédéfinie prend en argument d'entrée un ensemble de jeux d'activités alloués, et renvoie celui dont les opérations sous-jacentes seront exécutées en parallèle. Pour la classe Manager, cette méthode est BestActivitySetSelector[15]. En conséquence :

- Dans le menu 'Développement', item 'CONTROL DIESE', choisir 'System', puis 'Manager'. Sélectionner MonManager, puis cliquer sur 'Modifier'. Dans l'onglet 'Méthodes héritées', cocher la case 'Nom automatique' de la ligne 'BestActivitySetSelector', puis cliquer sur 'Coder'.

- Dans la fenêtre d'édition, remplacer le code proposé par Solfege par le § AXIV_E3 du fichier ressources_FCT.txt. Comme indiqué dans le commentaire du code, le schéma général est ici une réduction progressive de l'ensemble des

[14] Posée dans le Manager, la connaissance est commune aux différents blocs 'activités-ressources'. Posée dans les différents blocs, elle peut naturellement inclure des critères de choix locaux aux blocs.

[15] Pour la classe ActivitiesResourcesBlock, cette méthode est LocalBestActivitySetSelector.

alternatives, puis un choix arbitraire parmi les jeux non éliminés. D'autres schémas sont admis (par exemple l'attribution d'une note à chaque alternative, par application, séparée ou conjointe, de critères, suivi par le choix du jeu qui a la meilleure note), pourvu que le corps de la méthode renvoie un et un seul des jeux.

On fait usage d'une méthode prédéfinie de la classe OperatingSystem, SelectAlternativesOnMaxOperationPriority, qui renvoie la disjonction de jeux ex-æquo sur la priorité maximale. Noter qu'on raisonne bien sur le degré de priorité de l'opération (descripteur PriorityDegree), et non sur la priorité d'allocation de l'activité. L'opération est valeur du descripteur AllocatedOperation de l'activité. C'est la procédure d'allocation qui a instancié la classe dont le symbole est la valeur d u descripteur OperationClassId de l'activité (voir par exemple l'Etape 5.1 de l'Acte VIII, pour SemisBle). Noter enfin l'usage de la fonction prédéfinie RandomInteger(N), qui tire aléatoirement un nombre dans [0 N-1].

- Sauvegarder ce contenu de la méthode, quitter l'éditeur et valider cette description complétée du gestionnaire par le bouton 'Valider'.

Désormais, l'exécution de la simulation ne peut générer qu'une seule trace : la seconde parmi celles présentées plus haut.

Pour les autres activités :
- le désherbage, la fertilisation et les traitements se déroulent comme décrit à l'Etape 2.2 de l'Acte XIII.
- la récolte du blé et les coupes de luzerne, démarrent comme décrit à l'Etape 2.2 de l'Acte XIII, leur durée "nette" est restée la même, mais la durée "brute" est allongée du nombre des jours consacrés à l'apport de fourrage. cela est dû à la réquisition de deux unités de MO, ce qui empêche une exécution simultanée avec l'apport de fourrage.

Si on ajoute une unité de MO dans le pool du personnel, en enlevant la chaîne '//' en début des deux lignes instanciant l'agent luc, dans systemeOperant.inc, alors la récolte du blé se voit allouer jean et luc le jour où l'affouragement occupe paul, et peut être alors terminée le jour-même. Quant aux coupes de luzerne, elles récupèrent les trois unités de MO, puisque l'option MAX est activée dans la description du gestionnaire (sauf les jour d'affouragement, comme le 17/07).

En phase de développement de la base de connaissances, on a besoin de vérifier que chaque élément de code (fonction globale[16], corps de méthode propre ou héritée) se comporte comme spécifié, dans toutes les situations que ce code est censé gérer. besoin aussi de vérifier que la conjonction du code de la base de connaissances et du code du moteur de simulation provoque le comportement attendu du système, notamment en termes d'ouverture/fermeture des activités, d'allocation de ressources et de progression de l'effet des opérations.

A cette fin, le programmeur insère dans son code tous les éléments de trace techniques qui lui paraissent utiles à la vérification (ces éléments pourront être effacés avant la livraison à l'utilisateur). Avant l'observation d'une trace, le programmeur s'efforcera d'exprimer le résultat attendu, pour détecter plus sûrement les anomalies. De telles traces sont visibles dans les morceaux de code du fichier ressources_FCT.txt, mais la mise au point de cette version vérifiée du code a nécessité d'autres traces, non livrées ici. De manière générale, et idéalement, la vérification du code doit être faite unité de code par unité de code, ou dès qu'un ensemble d'unités de code indissociables a été programmé. On prend soin de faire varier les données d'entrée dans toute leur gamme.

Le développeur livre ainsi à l'utilisateur du simulateur[17] un outil vérifié au sens où le modèle conceptuel est entièrement, exclusivement et correctement respecté par le code informatique[18].

Quelle que soit l'attention portée à la vérification incrémentale du code, il peut rester imparfait sur trois aspects :

- une incapacité du code à respecter le modèle conceptuel dans telle ou telle situation pourtant incluse dans le domaine que les connaissances prétendent couvrir. Le développeur n'a pas vérifié le comportement du code dans ces situations, pratiquement ou théoriquement.

- une incapacité du processeur à poursuivre l'exécution du code compilé. Notamment parce qu'une variable pointe sur une mémoire non encore allouée.(pointeur nul, etc.) ou qui ne lui est pas alloué (indice de tableau illégal, etc.). Il s'agit d'un non respect du langage de programmation, et non du modèle conceptuel.

- une maîtrise incomplète de la désallocation de la mémoire allouée aux variables, fonctions et structures C++. Cela n'entraîne pas directement l'arrêt de l'exécution ou des sorties inattendues. Cependant, une telle "fuite de mémoire" est parfois le signe d'un codage incohérent, pouvant produire une erreur fatale.

Pour éviter les deux dernières situations, il est recommandé d'utiliser en cours de développement un outil de débogage, tel que gdb ou, seulement sous Linux, valgrind. On montre ici un cas d'utilisation en fin de phase de développement, en utilisant l'outil valgrind.

Etape 1 : Diagnostic et réparation d'erreurs fatales

- Dans le terminal d'exécution, taper et lancer la commande suivante :

```
>valgrind ./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str
SIM/sim1.dir -sb -o
```

La trace rend compte d'une erreur rencontrée par le processeur du code compilé : ⏩

```
==3489== Mismatched free() / delete / delete []
==3489==    at 0x4024B3A: free (vg_replace_malloc.c:366)
==3489==    by 0x4041CBE: uniteDeGestion_whenSetEntity_Body(MonitorMethod*) (in /home/rellier/APPLIS_DIESE/libKBS/lib_SiBemol_Tutoriel.1.0_CD.so.5.8)
==3489==    by 0x4140690: EntityVariableDescriptor::SetValue(BasicEntity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_BD.so.5.8)
==3489==    by 0x40F13E8: DescriptedEntity::SetEntityValue(int, BasicEntity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_BD.so.5.8)
==3489==    by 0x4284788: ParserStruct_parse() (in /home/rellier/LIB_DIESE/libDIESE/lib_BD_Parsers.so.5.8)
==3489==    by 0x4282C41: ParserStruct_parse() (in /home/rellier/LIB_DIESE/libDIESE/lib_BD_Parsers.so.5.8)
==3489==    by 0x4282C41: ParserStruct_parse() (in /home/rellier/LIB_DIESE/libDIESE/lib_BD_Parsers.so.5.8)
==3489==    by 0x428F1DB: Structure_Parser(_IO_FILE*, LexicalInputFile*) (in /home/rellier/LIB_DIESE/libDIESE/lib_BD_Parsers.so.5.8)
==3489==    by 0x412FD3B: ParseStructureFile(char*) (in /home/rellier/LIB_DIESE/libDIESE/lib_BD.so.5.8)
==3489==    by 0x804895F: main (in /home/rellier/APPLIS_DIESE/KBS/cdiese/SIBEMOL/SiBemol_Tutoriel.1.0/exec/main)
```

L'enchaînement des appels de fonctions, à partir de la fonction d'entrée main(...) s'arrête sur le corps de la méthode WhenSetInt du démon posé sur le descripteur UniteDeGestion. L'erreur se situe donc dans ce code. Il y est passé une instruction free, et l'outil de débogage signale qu'une instruction delete aurait dû être passée à la place.

- Dans le menu 'Génération', item 'Edition des fichiers source C++', choisir UserMonitor.cc, puis cliquer sur 'Editer'. Rechercher la fonction en cause, et dans son corps, vérifier la présence de l'instruction free erronnée. Cette instruction doit en effet être remplacée par la suivante :

```
delete pInstList;
```

- Effectuer ce changement, sauvegarder le contenu du fichier sans quitter l'éditeur.

- Compiler le simulateur, puis relancer l'exécution par la même commande que ci-dessus. Dans sa partie terminale, le

[16]Une fonction globale est installée dans une partie de la mémoire accessible, par son nom, de n'importe quel module du code de l'application.

[17]... qui ne sont pas toujours des personnes différentes.

[18] Pour autant, le simulateur n'est un outil valide pour aborder une étude sur le système visé que si le modèle conceptuel l'est d'abord.

résumé des diagnostics généré par valgrind, la trace indique que le code est désormais sans erreur : ⏭

```
==4235== HEAP SUMMARY:
==4235==     in use at exit: 536 bytes in 16 blocks
==4235==   total heap usage: 166,631 allocs, 166,615 frees, 8,639,062 bytes allocated
==4235==
==4235== LEAK SUMMARY:
==4235==    definitely lost: 484 bytes in 11 blocks
==4235==    indirectly lost: 0 bytes in 0 blocks
==4235==      possibly lost: 0 bytes in 0 blocks
==4235==    still reachable: 52 bytes in 5 blocks
==4235==         suppressed: 0 bytes in 0 blocks
==4235== Rerun with --leak-check=full to see details of leaked memory
==4235==
==4235== For counts of detected and suppressed errors, rerun with: -v
==4235== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 83 from 10)
```

La ligne probante à ce sujet est 'ERROR SUMMARY; 0 errors from ...'

- Maintenant que l'amélioration du code a été établie, il faut introduire la modification dans la base de connaissances. A cette fin, dans le menu 'Développement', item 'BASIC DIESE', choisir 'Monitor' puis 'DescValueMonitor'. Ouvrir une fenêtre de modification pour UniteDeGestionMonitor. Dans son onglet 'Méthodes héritées', cliquer sur 'Modifier' dans la ligne AssignWhenSetIntMethod. Dans la fenêtre d'édition, effectuer la même modification que dans le fichier UserMonitor.cc (free –> delete). Sauvegarder ce contenu, quitter l'éditeur et valider la classe par 'Valider'.

- **Sauvegarder la base !**

Etape 2 : Diagnostic et réparation des fuites de mémoire

On remarque, dans le résumé des diagnostics généré par valgrind, dans le chapitre 'LEAK SUMMARY' que 484 unités de mémoire sont devenues inaccessibles ('definitely lost'), c'est-à-dire telle que le code ne contient plus de pointeur actif vers elles. Elles ne peuvent donc plus être manipulées, et notamment désallouées et remise à la disposition du programme. On lit dans le résumé qu'il est conseillé de relancer valgrind avec l'option '--leak-check=full' pour en savoir plus. Donc, relancer l'exécution par :

```
>valgrind --leak-check=full ./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp
SIM/sim1.str SIM/sim1.dir -sb -o
```

La trace donne la précision suivante :

```
==4390== 44 bytes in 1 blocks are definitely lost in loss record 6 of 7
==4390==    at 0x402569A: operator new(unsigned int) (vg_replace_malloc.c:255)
==4390==    by 0x40E02C1: ClockValueToTm(int) (in /home/rellier/LIB_DIESE/libDIESE/lib_BD.so.5.8)
==4390==    by 0x4036EB4: iterationRecolteLuzerne_openingPredicate_Body(EntityMethod*) (in /home/rellier/APPLIS_DIESE/libKBS/lib_SiBe
mol_Tutoriel.1.0_CD.so.5.8)
==4390==    by 0x42E77B5: Activity::NotFalseOpeningPredicate() (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
==4390==    by 0x42E795A: Activity::IsOpenConditionSatisfied() (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
==4390==    by 0x42EB542: Activity::CheckOpenGeneralPrecondition(C_UpDown, Activity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.
5.8)
==4390==    by 0x42EAF37: Activity::ValidateOpen(C_UpDown, Activity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
==4390==    by 0x42FBC62: ActivityConjunction::PropagateOpenToSons(Activity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
==4390==    by 0x42ECACC: Activity::PropagateOpenToSons(Activity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
==4390==    by 0x42E541E: Activity::TurnToOpen(int, Activity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
==4390==    by 0x42FBDCA: ActivityConjunction::UpdateIfSonOpen(Activity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
==4390==    by 0x42ECFD6: Activity::UpdateIfSonOpen(Activity*) (in /home/rellier/LIB_DIESE/libDIESE/lib_CD.so.5.8)
```

L'enchaînement des appels de fonctions s'arrête dans le corps de la méthode OpeningPredicate de l'activité agrégée IterationRecolteLuzerne. L'erreur se situe donc dans ce code. Plus précisément, dans le service prédéfini ClockValueToTm, lequel, par l'opérateur new alloue une unité de mémoire, nécessairement associée à une variable qui pointe sur elle[19]. Et valgrind signale que cette unité de mémoire, désormais allouée, a ultérieurement perdu son pointeur avant qu'elle puisse être désallouée. D'où l'attitude de réparation suivante :

- Dans l'éditeur de code encore ouvert, charger le fichier UserActivity.cc. Rechercher le prédicat en cause, et dans son corps, repérer l'instruction ClockValueToTm incriminée. Il s'agit, sans ambiguïté, de :

```
tm* pTm = ClockValueToTm(now);
```

La documentation de ce service stipule que l'utilisateur est en charge de la désallocation de la structure tm créée et retournée. Il s'agit ici la la variable locale pTm. Puisque cette variable est locale, sa "durée de vie" s'arrête quand ClockValueToTm redonne le contrôle au code qui l'a invoqué (c'est-à-dire lors du 'return'). Et on constate que la mémoire pointée par pTm n'est effectivement pas désallouée entre la création du pointeur et l'instruction return.

- Ecrire, donc, l'instruction de désallocation dès que la variable pTm n'est plus d'utilité. On obtient :

```
int seuilPourDebutCoupe = ....;
delete pTm;
```

[19] Ce qui signifie que la valeur de cette variable est l'adresse de l'unité de mémoire dans l'espace du programme.

- Sauvegarder le contenu du fichier sans quitter l'éditeur.

- Compiler le simulateur, puis relancer l'exécution par la même commande que ci-dessus. Dans sa partie terminale, le résumé des diagnostics généré par valgrind, la trace indique que le code est désormais sans fuite de mémoire : ⏩

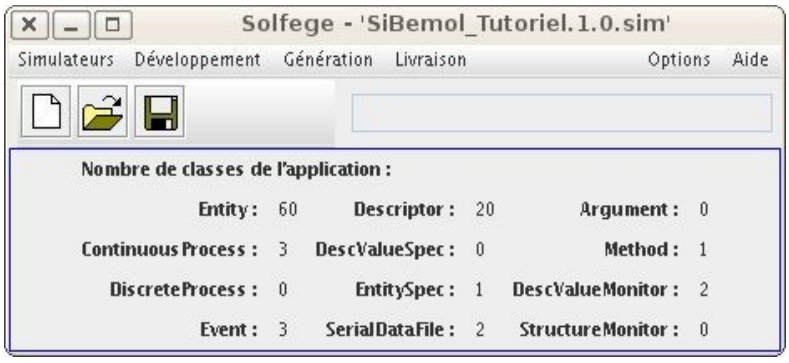
```
==4675== LEAK SUMMARY:
==4675==    definitely lost: 0 bytes in 0 blocks
==4675==    indirectly lost: 0 bytes in 0 blocks
==4675==      possibly lost: 0 bytes in 0 blocks
==4675==    still reachable: 52 bytes in 5 blocks
==4675==         suppressed: 0 bytes in 0 blocks
==4675== Reachable blocks (those to which a pointer was found) are not shown.
==4675== To see them, rerun with: --leak-check=full --show-reachable=yes
==4675==
==4675== For counts of detected and suppressed errors, rerun with: -v
==4675== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 83 from 10)
```

Quant aux 52 unités de mémoire déclarées 'still reachable', elles étaient encore allouées juste avant la sortie du simulateur, et le retour à la ligne de commande du système hôte. Il s'agit de structures générées par le moteur de DIESE[20] et non par le code de la base de connaissances. Le développeur de la base doit donc ignorer cette indication, sans conséquence négative pour le fonctionnement du simulateur.

- Introduire la modification dans la base de connaissances. Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Activity' puis '... nonPrimitive'. Ouvrir une fenêtre de modification pour IterationRecolteLuzerne. Dans son onglet 'Méthodes héritées', cliquer sur 'Modifier' dans la ligne OpeningPredicate. Dans la fenêtre d'édition, effectuer la même modification que dans le fichier UserActivity.cc (ajout du delete). Sauvegarder ce contenu, quitter l'éditeur et valider la classe par 'Valider'.

- **Sauvegarder la base !**

A ce stade du développement de la base de connaissances, vu comme un terme provisoire, la fenêtre d'accueil de l'interface Solfège présente les statistiques suivantes pour la création d'objets : ⏩



[20] Plus précisément par les automates d'interprétation des fichiers placés en argument de la commande `./main`. Le code C de ces automates sont générés par les outils lex (ou flex) et yacc (ou bison), et livrés avec la bibliothèque DIESE. Les 5 blocs mentionnés dans la trace correspondent aux 5 fichiers placés en argument s de la commande.

ANNEXE 1 : La livraison de la base de connaissances

On traite ici la manière dont un simulateur est livré par le développeur d'une base de connaissances à ses utilisateurs, en termes de forme du produit livré et de procédure d'élaboration.

Situation 1 : Le développeur est un des utilisateurs du simulateur

C'est souvent le cas : l'utilisation se fait dans la continuité du développement, et on observe des allers-retours entre l'espace de développement (le répertoire de la base de connaissances) et l'espace d'utilisation (le répertoire d'exécution, lui-même pointant sur l'espace des données). La fréquence des allers-retours baisse jusqu'à la (quasi-)stabilisation de la base. On n'identifie pas de "produit livré", qui serait autre chose que la base de connaissances en développement.

Au cours du développement mené dans les Actes précédents, le développeur a exploité le simulateur en "mode commande". C'est-à-dire en écrivant directement une commande exécutable dans un terminal et en demandant au processeur de commandes (par la touche 'Enter') de la traiter. Le simulateur peut aussi être exploité au travers d'une interface : l'utilisateur gère alors les entrées/sorties du simulateur par des outils (graphiques) de visualisation / édition / spécification / exécution, et c'est l'interface qui sollicite le processeur de commandes par délégation de l'utilisateur.

Une telle interface, **MI_Diese**, est livrée dans le paquet DIESE, avec son manuel d'utilisation. Son lancement se fait de la même manière que pour l'interface de développement Solfege, en passant la commande suivante dans un terminal :

```
>java -jar midiese.jar &
```

Noter que le programme exécutable doit avoir été équipé de capacités additionnelles, par rapport à l'exécutable main exploité en mode 'commande'. Notamment la réception de commande en provenance de l'interface et l'envoi vers l'interface de messages sur l'évolution de l'exécution. A cet effet, la commande de compilation 'make std' doit être remplacée par la commande 'make Ihm'.L'exécutable crée n'est plus main mais mainIhm. On rappelle que celui-ci est invoqué par l'interface MI_Diese, en réaction à un événement sur un des éléments graphiques de l'interface.

Etape nécessaire : la génération du fichier des noms de classes.

L'interface MI_Diese est générique (au sens où elle doit être particularisée pour servir dans un domaine donné). Pour présenter à l'utilisateur des fenêtres et autres cadres faisant référence aux entités du domaine étudié, l'interface doit être informé de ces entités. Pour cette information, une convention de contenu, de format et d'emplacement a été établie pour la relation développeur / utilisateur en mode "interface" : un fichier, nommé classe.dev est créé par le développeur dans un répertoire (de nom Liv) situé à côté du répertoire d'exécution. Il contient une ligne par classe d'objet (entité, processus, etc.) développée, et une ligne par descripteur et méthode (propre ou hérité(e)) de ces objets. L'interface MI_Diese accède d'elle-même, automatiquement, à ce fichier. Pour créer ce fichier :

- Dans le menu 'Livraison', choisir 'Génération du fichier des noms de classes'.

On peut, par pure curiosité, regarder le contenu de ce fichier, en se déplaçant dans le répertoire Liv (celui-ci a été automatiquement créé lors de l'identification de la base de connaissance, à l'Etape 3 de l'Acte I). En voici un extrait pour la base développée ici : ⏭

```
Entity  "troupeau"      ParentclassNames    ""
VariableDescriptor  "troupeau"  "nombreAnimaux"
VariableDescriptor  "troupeau"  "herbeDisponible"
ConstantDescriptor  "troupeau"  "animalRepresentatif"
ConstantDescriptor  "troupeau"  "parcelleSupport"
ConstantDescriptor  "troupeau"  "dateDernierApport"
Entity  "unAgent"       ParentclassNames    "performerSpecification"
Entity  "unVacher"      ParentclassNames    "performerSpecification"
Entity  "unitGranulatedOperation"       ParentclassNames    "operation"
ConstantDescriptor  "unitGranulatedOperation"   "unitSpeed"
```

Situation 2 : Un utilisateur exploite un simulateur qu'il n'a pas développé

La livraison d'un simulateur peut s'opérer de deux manières :

a) En relation étroite, le développeur livre à l'utilisateur une copie de ses répertoires de développement et de données. On recommande un contenu allégé et compressé.

. Allégé des fichiers inutiles ou qui devront nécessairement être régénérés dans l'environnement de l'utilisateur : les répertoires dans .Trash/ (lire ci-dessous), les modules compilés dans les répertoires libLinux/ et libCygwin/, les exécutables main et mainIhm dans le répertoire exec/, le lien symbolique vers le répertoire des données créé dans le répertoire exec à l'Etape 1.2 de l'Acte 3.

. Compressé dans un fichier-archve, par des utilitaires tels que gz, gzip, tar.

Pour alléger le répertoire .Trash/ (mémoire d'anciennes versions du code, voir la remarque en fin de l'Acte IV), dans le menu 'Simulateurs', choisir 'Alléger la corbeille', puis suivre les instructions. Le dialogue demande pour chaque répertoire correspondant à une date de version si le développeur veut supprimer le contenu du répertoire, et s'il veut

supprimer le répertoire lui-même. Les répertoires sont examinés par ordre d'ancienneté croissante.

b) En relation éloignée ou inexistante, le développeur livre à l'utilisateur un paquet élaboré à partir de deux choses :

1) un ensemble de cadres et de formulaires, partiellement pré-remplis, réceptacles des données d'entrées dans la situation d'étude de l'utilisateur.

2) la base de connaissances, sous deux formes alternatives :

. 2.1) la forme de l'exécutable accompagné de la bibliothèques de modules compilés. L'utilisateur ne peut pas entrer, pour le développer ou le modifier, dans cet objet fermé qu'est devenu le simulateur.

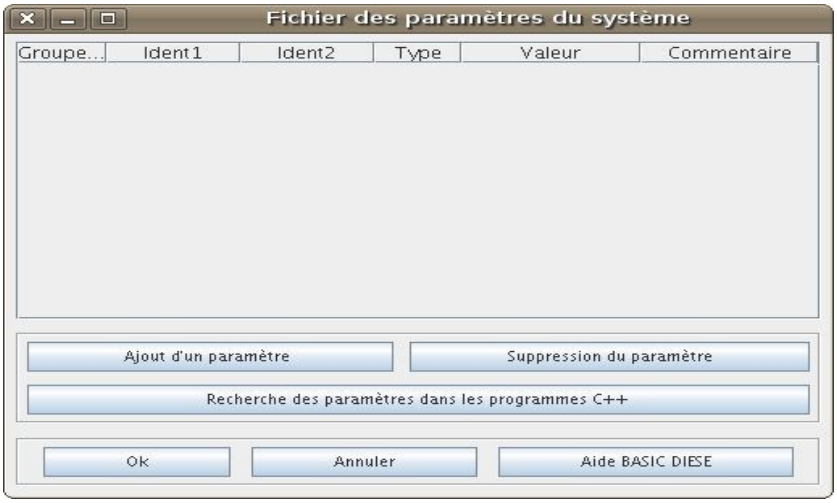
. 2.2) la forme d'un ensemble de spécifications, excluant ainsi sa traduction en C++ et son assemblage en une bibliothèque de modules exécutables. L'utilisateur peut alors se placer lui aussi en position de développeur, en utilisant l'interface Solfege.

2.1) Création des cadres pour les données

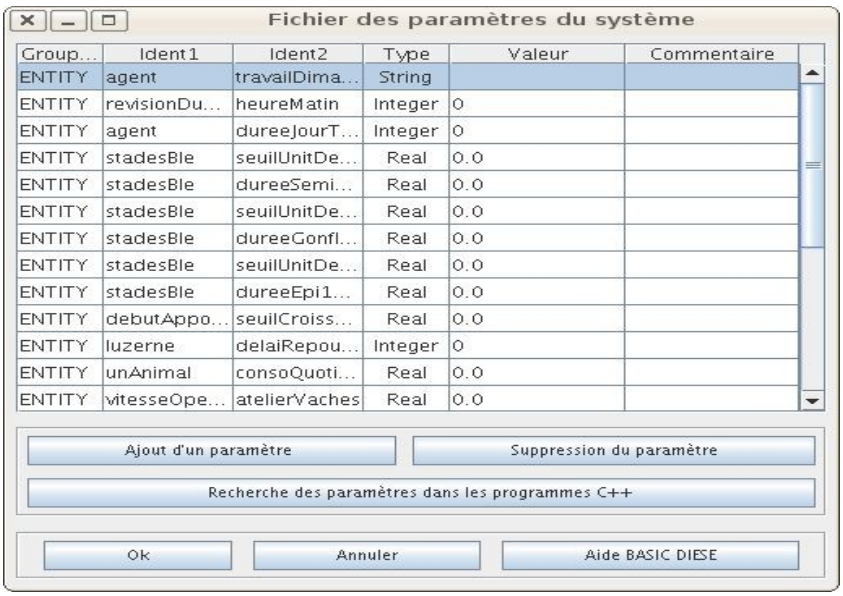
- Dans le menu 'Livraison', item 'Génération des fichiers d'entrée', choisir 'tous'. On demande ainsi à Solfege de générer un cadre initial pour les cinq fichiers qui seront placés en argument de la commande d'exécution ./main.

2.1.1) Les paramètres du système

- Pour la création du cadre du fichier des paramètres du système (celui auquel on a donné le nom sim1.par dans les Actes précédents), Solfège ouvre la fenêtre ci-dessous : ⏭



- Cliquer sur le bouton 'Recherche des paramètres ...', pour engager Solfege à parcourir l'ensemble du code de l'application à la recherche des appels aux services Get<type>ParameterValue(<ident1>, <ident2>). Le résultat de cette recherche est affichée dans la fenêtre, avec autant de lignes que de paramètres identifiés. ⏭



Dans les colonnes 'Valeur' et 'Commentaire', le développeur peur suggérer une valeur typique, ou recommandée, et une indication documentaire.

- Cliquer sur le bouton 'Ok' pour valider ce contenu et laisser Solfege enregistrer les données dans un fichier de nom pattern.par dans le répertoire Liv/. Et en même temps les fichier de suffixe .sim, .osp; .str et .dir.

2.1.2) Les paramètres de la simulation

On va stipuler que la base de connaissances a été conçue pour fonctionner avec une unité d'horloge d'une heure, et que le générateur de nombre aléatoire doit être utilisé (au moins lors de la sélection du meilleur jeu d'activité (cf. Etape 3 de l'Acte XIV).

- Dans le menu 'Livraison', item 'Edition des fichiers d'entrée', choisir 'Paramètres de la simulation'. Dans la fenêtre d'édition :
. enlever les '//' en début de la directive INITRAND. On laisse l'utilisateur libre de donner ou non un argument à la directive. Dans le premier cas, la série générée sera toujours la même d'une simulation à l'autre. Cette option est utile pour comparer deux cas différents sur un ensemble de points, "toutes choses égales par ailleurs", y compris les dynamiques aléatoires.
. remplacer "UNITE_TPS MINUTE" par "UNITE_TPS HOUR"
. supprimer le paragraphe final relatif à l'ouverture d'un fichier séquentiel : ces aspects sont gérés autrement dans la base de connaissances
. les autres spécifications sont laissées à convenance de l'utilisateur.

Le contenu devient ainsi tel que ci-dessous : ⏭

```
//initialisation de la simulation de SiBemol_Tutoriel1.0
//TRACE PARSING

INITRAND [<intExpression>];

INITIALISATION SIMULATION
    UNITE_TPS HOUR; //DAY MINUTE SECOND
    NB_UNITE_TPS 1;
    DATE_DEBUT <dd> JAN <yy> [<hh> <mm> <ss>];
    DATE_FIN <dd> JAN <yy> [<hh> <mm> <ss>];
    //IN_DIR " <path> ";
    //OUT_DIR " <path>";
;
```

2.1.2) Les autres fichiers d'entrée

Le fichier **pattern.osp** est généralement laissé vide, puisque le développeur n'a pas de suppositions sur le domaine précis d'intérêt pour un utilisateur autre que lui-même.

Le fichier **pattern.dir** est généralement valide tel que généré par Solfège, puisqu'il contient les directives quasi-universelles d'une simulation. On ne le modifie que pour des besoins très particuliers.

Quant au fichier **pattern.str**, l'effort de rédaction produit par le développeur pour la mise au point de la base de connaissance et la vérification du code peut être valorisé en le transmettant, pour exemple, aux utilisateurs. celui tirera utilement parti de cet exemple pour décrire ses propres cas d'étude. A cette fin, le développeur pourra coller dans pattern.str son fichier sim1.str, précédé des fichiers d'inclusions 'démunis de leur directive 'END INCLUDE;'. Ou bien livrer, d'une manière ou d'une autre, ces fichiers avec la base.

2.2) Paquetage de la base de connaissances

Avec la bibliothèque DIESE, sont livrés aux développeurs d'applications deux petits programmes pour empaqueter une base de connaissance en vue d'une livraison à ses utilisateurs.
- makeRelease_APPLI_lib est le programme qui permet de livrer la base sans le code source (donc l'exécutable accompagné de la bibliothèques de modules compilés) : c'est la base qualifiée de "fermée".
- makeRelease_APPLI_src permet de livrer le code 'source' de la base, que l'utilisateur aura la charge de transformer en un simulateur exécutable. C'est la base qualifiée d'"ouverte".

Le programme d'installation de DIESE a placé les programmes makeRelease_APPLI_[lib src] dans le répertoire des applications : ce répertoire est /home/rellier/APPLIS_DIESE/ dans la situation de développement de SiBemol_Tutoriel (voir l'Etape 2 de l'Acte I). Plus précisément dans son sous-répertoire scripts/. Ces programmes sont en réalité des scripts, au sens d'un programme d'instructions adressées directement au système hôte (telles qu'un changement de répertoire de travail, une copie/déplacement/suppression de fichier, etc.).

2.2.1) Préparation du paquetage, quelle que soit sa forme

Une action indispensable déjà décrite (ci-dessus à l'Etape 2.1) est la création dans le rangement, dans le sous-répertoire Liv/ de la base, des fichiers de cadres pour les données.

En complément, le développeur peut placer dans le sous-répertoire doc/ de la base tout document utile à l'utilisateur du simulateur : modèle conceptuel, manuel d'utilisation, manuel de référence, articles, etc.

Dans les répertoires in/, le développeur peut placer des fichiers d'entrées des simulations, pour exemple ou pour exploitation recommandée. Il peut s'agir de fichier *.par, *.str, etc (ceux à placer en argument de la commande) ou de tout autre fichier lu dans le cours de la simulation (par exemple les fichiers UDEV_BLE et DISPO_HERBE utilisés

dans ce tutoriel.

Dans les répertoires out/, le développeur peut placer des fichiers de sorties des simulations, pouvant correspondre au fichier d'entrée du répertoire in/.

Une version standard du fichier READ_ME est généré par le script d'empaquetage, mais il peut être développé librement.

2.2.2) Le paquetage de la bibliothèque : la base fermée

- Dans le répertoire exec/, générer les exécutables main et mainIhm.

- Aller dans le répertoire des scripts d'empaquetage :

```
>cd /home/..../APPLIS_DIESE/scripts
```

- Lancer le script adapté au système hôte : suffixé '.sh' dans un "Bourne shell", sufcixé '.csh' dans un "C shell"[21] :

```
>./makeRealease_APPLI_lib.sh   cdiese/SIBEMOL   SiBemol_Tutoriel.1.0
```

Par convention, la base de connaissance a été développée dans le sous-répertoire KBS/ frère de scripts/ (voir l'Etape 2 de l'Acte I). Le second argument de la commande ci-dessus est la partie du chemin pour aller de KBS/ au répertoire de la base. Le troisième argument est le nom de la base.

Le paquet généré est un répertoire compressé et archivé. Il est placé sous /home/..../APPLIS_DIESE/releases/lib/. Plus précisément dans le sous-répertoire Linux/ ou Cygwyn/ selon que le script d'empaquetage makeRelease_APPLI_lib a été exécuté sous Linux ou sous Windows/Cygwin.

Son contenu est le suivant :
. répertoire exec/ : les exécutables main et mainIhm,
. répertoire lib/ : la librairie des modules compilés de l'application
. répertoires Liv/, doc/, in/, out/ et fichier READ_ME : voir ci-dessus
. deux scripts d'installation de la base de connaissances dans le système de fichiers de l'utilisateur

Le nom du paquet généré est : <nom_BDC>_lib[W]D<major><minor>, par exemple SiBemol_Tutoriel_libD5.8, puisque la version de DIESE utilisée dans le présent développement a été DIESE.5.8. Le 'W' est présent dans les paquets créés sous Windows/Cygwin. Sa forme archivée et compressée est alors SiBemol_Tutoriel_libD5.8.tar.gz : c'est ce fichier qui est à livrer aux utilisateurs.

2.2.3) Le paquetage des spécifications 'source' : la base ouverte

- Aller dans le répertoire des scripts d'empaquetage :

```
>cd /home/..../APPLIS_DIESE/scripts
```

- Lancer le script adapté au système hôte : suffixé '.sh' dans un "Bourne shell", sufcixé '.csh' dans un "C shell" :

```
>./makeRealease_APPLI_src.sh   cdiese/SIBEMOL   SiBemol_Tutoriel.1.0
```

Avec les mêmes convention que pour la livraison de la librairie, ci-dessus.

Le paquet généré est un répertoire compressé et archivé. Il est placé sous /home/..../APPLIS_DIESE/releases/src/LinuxCygwin/. Que le script d'empaquetage makeRelease_APPLI_src ait été exécuté sous Linux ou sous Windows/Cygwin, le paquet est identique.

Son contenu est le suivant :
. répertoires Liv/, doc/, in/, out/ et fichier READ_ME : voir ci-dessus
. deux scripts d'installation de la base de connaissances dans le système de fichiers de l'utilisateur
. répertoires Cls/, Fct/,et Var/ : la bses de connaissances exploitable par l'interface Solfege.

Le nom du paquet généré est : <nom_BDC>_srcD<major><minor>, par exemple SiBemol_Tutoriel_srcD5.8, puisque la version de DIESE utilisée dans le présent développement a été DIESE.5.8. Sa forme archivée et compressée est alors SiBemol_Tutoriel_srcD5.8.tar.gz : c'est ce fichier qui est à livrer aux utilisateurs.

[21] Pour identifier le shell local, taper la commande 'ps | grep sh' dans un terminal.

ANNEXE 2 : Sauvegarde et traitement de données recueillies en cours de simulation

En phase d'utilisation du simulateur, on établit en principe un plan d'expérience. Ce plan peut être une série finie de cas. Chaque cas est une configuration du système piloté (lieux de production, moyens de production, plan de conduite) assortie d'une ou plusieurs hypothèse sur l'environnement tout au long de la période de simulation. Le plan d'expérience peut aussi être le croisement (complet ou non) de quelques facteurs du comportement du système, chacun résumé en une gamme de modalités. Dans les deux cas, l'utilisateur a besoin d'enregistrer des traces de chaque exécution, ciblées sur des indicateurs pertinents pour son étude.

Il y a deux manières de produire des traces :

- Le développeur augmente la base de connaissances avec des structures de données et des modules de code pour produite des données en sortie. Cette sortie s'impose alors à tous les utilisateurs du simulateur (sauf le cas particulier où cette sortie a été rendue optionnelle par le développeur).

- L'utilisateur exprime dans un fichier placé en entrée de la simulation (le fichier suffixé par '.osp') ce qu'il souhaite produire en sortie de l'exécution d'une simulation. Chaque utilisateur écrit des directives personnelles, et peut en changer d'une simulation à l'autre.

Ces deux manières peuvent être mises en œuvre de manière additionnelle, pour produire des sorties sur des aspects différents de la simulation. Elles peuvent aussi être complémentaires : les sorties sur directives externalisées sont préparées et permises par du code internalisé par le développeur.

Etape 1 : Usage des spécifications de sortie

- Dans le répertoire des données, éditer le fichier sim1.osp, créé à l'Etape 1 de l'Acte III. Y coller le § N2_E1 du fichier ressources_SIM.txt.

Dans le fichier de nom herbeDispo.txt, on demande la trace de l'évolution tout au long de la période de simulation (mot-clé ALL), de deux variables d'état des instances de la classe Troupeau : HerbeDisponible et NombreAnimaux. Par le mot-clé NEW, on demande que le fichier soit écrasé à chaque nouvelle simulation. La date sera tracée en nombre d'unités d'horloge depuis le début de la période (mot-clé CLOCK)[22]. Dans le fichier herbeDispo2.txt, la date est le triplet <heure, jour, mois>, et on n'affiche pas le nombre d'animaux.

Les fichiers de trace sont placés par le moteur dans le répertoire mentionné dans le fichier des paramètres de la simulation, sim1.sim, en regard du mot-clé OUT_DIR. Tel que spécifié à l'Etape 2 de l'Acte III ('../out/'), ce répertoire se nomme out/, et est placé à côté du répertoire d'exécution (exec/).

Remarque importante : les descripteurs constants sont admis dans une spécification, mais leur éventuelle évolution ne sera pas tracée dans le fichier de sortie.

- Dans le répertoire d'exécution, relancer la commande habituelle :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
```

- Se déplacer dans le répertoire des sorties :

```
>cd ../out
```

- Le fichier herbeDispo.txt commence comme ci-dessous ⏭, et herbeDispo2.txt comme ci-dessous ⏭:

```
---------[out]>more herbeDispo.txt
#troupeau       clock   herbeDi nombreA
monTroupeau     0       _       20
monTroupeau     128     300.00  20
monTroupeau     248     600.00  20
monTroupeau     369     900.00  20
monTroupeau     489     1200.00 20
monTroupeau     609     1500.00 20
```

```
---------[out]>more herbeDispo2.txt
#troupeau       hour    day     month   herbeDi
monTroupeau     8       20      10      300.00
monTroupeau     8       25      10      600.00
monTroupeau     8       30      10      900.00
monTroupeau     8       4       11      1200.00
monTroupeau     8       9       11      1500.00
```

Comme attendu, le nombre d'animaux est initialisé à 0 lors de la lecture des fichiers en entrée, puis ne bouge plus. La première augmentation de la quantité d'herbe disponible est à 8 heures le sixième jour de la période jour de la période (puisque au moins 5 fois 24 heures après de début de simulation, comme spécifié dans le prédicat d'ouverture de l'affouragement, et à 8h du matin puisque c'est l'heure de l'examen du plan).

On remarque que le troisième apport est fait, comme attendu, 10 jours après le premier, et à 8h. Mais les instants d'horloge sont 128 et, pour le troisième, 369 et non 368 (128 + 10*24h). Cela est dû au passage à l'heure d'hiver le dimanche 29 octobre 2000 : ce jour-là, la montre passe à 8h 25heures après y être passé la veille. Le nombre d'heures de simulations, c'est-à-dire le nombre d'unités d'horloge écoulées, entre les deux passages à 8h est donc de 25.

[22] Pour description complète du langage d'expression des spécifications de sortie, se reporter à la documentation de BASIC DIESE (rubrique 'Les spécifications de sortie').

Etape 2 : Internalisation de fonctions de sorties dans la base

Le développeur est en position de penser qu'une sortie pertinente des simulations est le déroulé des actions techniques (nature et dates) et qu'une forme naturelle de cette sortie est un diagramme sur lequel le temps serait en abscisses. En ordonnées pourrait être un booléen jour travaillé/non travaillé, ou bien un nombre d'heures travaillées. En outre, il paraît utile de visualiser de manière différenciée les trois "ateliers" : blé, luzerne et troupeau.

Le développeur peut alors intégrer dans la base de connaissances la fabrication d'un fichier pour ces données, à traiter en différé par un générateur de graphiques. Il est raisonnable de ne pas pousser jusqu'au codage de la génération automatique des graphiques, et de laisser à l'utilisateur le choix de son outil.

Le fichier évoque dans le paragraphe précédent devrait contenir autant de lignes que de jours de simulation. Et dans chaque ligne on devrait trouver une indication temporelle (numéro d'ordre ou date explicite), et le nombre d'heures travaillées pour chacun des trois ateliers, toujours dans le même ordre.

2.1 Pourquoi ne pas faire usage d'une spécification de sortie ?

Une spécification de sortie, pour ne considérer que celles sur les descripteurs d'entités[23], porte sur une classe et sur un sous-ensemble de ses descripteurs variables. Elle trace, instance par instance, les changements de valeur intervenus pendant la période de simulation. Il y a un élément de trace (une "ligne"), et une seule, par instant où au moins un des descripteurs a changé de valeur. Dans l'état actuel de la base de connaissance, il n'y a pas de classe qui encapsule à elle seule des informations sur les différents ateliers, si ce n'est Exploitation, mais par une relation indirecte structurelle, et la classe n'a pas de descripteurs propres.

On peut donc envisager doter cette classe Exploitation de tels descripteurs. La solution complète comporte alors les éléments suivants :

- Création des descripteurs variables NbHeuresApportFourage, NbHeuresCoupeLuzerne et NBHeuresOperationBle La valeur est le nombre entier d'heures travaillées le jour courant. Attachement à la classe d'entités Exploitation.

- Au début de chaque journée, la valeur des descripteurs est initialisée à zéro. On peut choisir de faire cela dans la méthode héritée SetPostConsequence de l'événement RevisionMatinaleDuPlan.

- En cours de journée, et si une opération progresse, on cumule dans le descripteur correspondant le nombre d'heures travaillées. L'endroit naturel pour faire cela est le moniteur sur le degré de progression. On augmente alors le cumul avec la valeur du 'pas' de progression de l'opération.

- Parce qu'on ne souhaite en sortie qu'une seule valeur quotidienne (et non une ligne pour la remise à zéro matinale, plus les changements à chaque 'pas' si celui-ci était inférieur à la journée), il faut installer les détails suivants :

- Pour la remise à zéro et les cumuls, on désactive l'enregistrement du changement au titre de la spécification de sortie[24].

- En fin de journée, on réaffecte sa valeur à chaque descripteur, en ayant pris soin de réactiver l'enregistrement du changement au titre de la spécification de sortie. On crée à cet effet un processus ponctuel dédié, exécuté par un événement autogénéré chaque fin de journée.

- Le fichier sim1.osp contient une spécification telle que :

```
SAVE DESCRIPTOR "exploitation" ALL "heuresTravaillees.txt' NEW DATE_HDM
"nbHeuresApportFourage" 0 "nbHeuresCoupeLuzerne" 0 "nbHeuresCoupeLuzerne" 0;
```

Cette conception se révèle un peu complexe. Et aussi délicate à mettre en œuvre, parce qu'elle demande une bonne connaissance du mécanisme des spécifications de sortie . On va donc rechercher une solution plus simple et moins "experte" : l'écriture du fichier de sortie directement par le code de l'application.

2.2 La gestion d'un fichier de sortie par le code de l'application

La solution comporte les éléments suivants :

- Création d'une structure interne pour gérer un fichier externe et ouverture physique "en écriture".

- Création des descripteurs variables NbHeuresApportFourage, NbHeuresCoupeLuzerne et NBHeuresOperationBle La valeur est le nombre entier d'heures travaillées le jour courant. Attachement à la classe d'entités Exploitation.

- En cours de journée, et si une opération progresse, on cumule dans le descripteur correspondant le nombre d'heures travaillées. Dans le moniteur sur le degré de progression, on augmente le cumul avec la valeur du 'pas' de l'opération.

- En fin de journée, on écrit une nouvelle ligne dans un fichier, avec la valeur des trois descripteurs. Puis on remet les valeurs à zéro. On crée à cet effet un processus dédié, exécuté par un événement autogénéré chaque fin de journée.

- Fermeture du fichier externe et désallocation de la structure interne de sa gestion..

[23] Il en existe aussi sur la structure des entités, et sur les occurrences d'événements.

[24] DIESE fournit les services à cet effet : DisableRecording et EnableRecording, dans la classe Descriptor.

2.2.1 Gestion du fichier de sortie externe

DIESE a prédéfini la classe OutputFile pour encapsuler informations et services associés à ce type de fichier externe. Une information majeure est la localisation sur le système de fichiers hôte, et les services sont notamment l'ouverture et la fermeture. On choisit d'introduire ce code relativement simple dans la fonction principale 'main'. Une alternative possible, mais plus lourde, aurait été de dédier des événements (et leurs processus) à cette tâche.

Le pointeur sur la structure OutputFile devant être accessible, non seulement dans la fonction 'main', mais aussi dans le futur code de l'écriture des valeurs, il sera pratique que ce pointeur soit une variable globale.

Enfin, le développeur a l'idée de rendre cette fonction de sortie optionnelle, commandée par un argument optionnel dans la ligne de commande. On profitera de cet argument pour "passer" le nom du fichier au simulateur. L'existence de ce nom sera le signal, pour le reste du code et sous la forme d'une variable globale, de l'activation de la fonction de sortie.

- Dans le menu 'Développement', choisir '**Variables globales**'. Dans la fenêtre de choix, cliquer sur 'Créer', puis taper gFileName_HeuresTravaillees. Choisir 'char*' comme 'Type'. Dans la champ 'Value', taper deux guillemets pour que la valeur par défaut soit une chaîne vide.

- Dans la même fenêtre de choix, cliquer à nouveau sur 'Créer', puis taper gFile_HeuresTravaillees. Choisir 'OutputFile*' comme 'Type'. Dans la champ 'Value', taper NULL pour que la valeur par défaut soit le pointeur NULL

- Dans le menu 'Développement', choisir 'Function' puis **'Interprétation ligne de commande...'**. Dans la fenêtre d'édition, :

. insérer le § N2_E2.2.1a du fichier ressources_FCT.txt juste après la ligne qui value gTraceOperations à TRUE si un argument est '-o'.

. plus haut, insérer le § N2_E2.2.1b juste après la ligne qui imprime l'usage optionel de l'option '-o'.

L'information à écrire dans la ligne de commande sera constituée du mot-clé '-f' suivi du nom du fichier externe, lequel sera placé par le moteur dans le répertoire désigné après le mot-clé OUT_DIR dans sim1.sim..

- Dans le menu 'Développement', choisir 'Function' puis **'Fonction 'main''**. Dans la fenêtre d'édition, insérer le § N2_E2.2.1b du fichier ressources_FCT.txt juste après l'instruction ParseStructureFile(...) d'interprétation du fichier de la structure du système. Si le nom du fichier a été renseigné dans la ligne de commande, on instancie la classe OutputFile, puis on ouvre le fichier, sur lequel on écrit une ligne d'entête. Le paragraphe qui suit installe dans l'agenda l'événement autogénéré d'écriture sur le fichier. C'est ici une instance de la classe de base Event, à laquelle on attache une instance de la classe de processus dédiée à l'écriture, ProcessusMemorisationHeuresTravaillees (avec la directive EXEC), et qu'on développera plus tard. On attache à ce processus l'exploitation comme entité touchée. On indique que chaque occurrence intervient 24 heures après la précédente. Et que la première intervient tout à fait en fin de journée (12 heures après le début de journée).

- Dans la même fenêtre d'édition de la fonction 'main', insérer le § N2_E2.2.1c du fichier ressources_FCT.txt juste après l'instruction C_ParseDirectiveFile(...) d'interprétation du fichier des directives de la simulation. La première instruction ferme le fichier de sortie, par Close(), comme le Open() l'avait ouvert. Mais attention ! On sait que le fichier sim1.dir contient une directive de désallocation de tous les fichiers alloués (DELETE FILE;, voir l'Etape 3.4 de l'Acte IV). sans précaution, l'instruction de fermeture va pointer sur une mémoire désallouée, provoquant une erreur fatale. On va donc inhiber la directive DELETE FILE; dans sim1.sim (avec la chaîne '//' en début de ligne), et compléter le code de 'main' avec le service de DIESE invoqué par cette directive (DeleteFilesInstances).

- Sauvegarder la fonction 'main', puis quitter l'éditeur.

2.2.2 Création des descripteurs et attachement

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Descriptor' puis 'Variable'[25]. Dans la fenêtre de choix, cliquer sur 'Créer'.

- Dans la fenêtre de modification, taper NbHeuresApportFourage dans le champ 'Name', puis confirmer INT pour le 'Type'. Ajouter un commentaire tel que "valeur journaliere". Valider la classe par le bouton 'Valider'.

- Procéder de la même manière pour les descripteurs NbHeuresCoupeLuzerne et NbHeuresOperationBle.

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Entity'. Sélectionner Exploitation et cliquer 'Modifier'.

- Dans la fenêtre de modification, onglet 'Descripteurs propres', ajouter les trois descripteurs variables précédemment créés. Valider cette définition augmentée de la classe par le bouton 'Valider'.

2.2.3 Cumul des heures travaillées

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Monitor' puis "DescValueMonitor'. Dans la fenêtre de choix, sélectionner 'OperationTraceeDegreProgressionMonitor', puis cliquer sur 'Modifier.

- Dans la fenêtre de modification, onglet 'Méthodes', cliquer sur 'Modifier' dans la ligne 'AssignWhenSetFloatMethod'.

[25] On aurait pu choisir Constant, puisqu'on ne va pas attacher de démon à ces descripteurs, et qu'on a pris une autre voie que la trace par spécification de sortie. On suggère 'Variable', pour que le lecteur-développeur puisse, à sa convenance, installer aussi la voie par spécification de sortie sans retour en arrière dans la définition des descripteurs.

Dans la fenêtre d'édition, coller le § N2_E2.2.3 du fichier ressources_FCT.txt, après l'instruction 'delete perfSetList' et juste avant l'instruction de retour ('return newValue;'). Si on a demandé la sortie dans la ligne de commande, et en fonction du type de l'opération dont de degré de progression est en train d'augmenter, on accroît la valeur de tel ou tel descripteur de l'exploitation (ici le pointeur pEA). Noter l'utilisation du service ClassIdToCharTab pour afficher '"en clair" (une chaîne de caractères) le symbole (numérique) d'une classe.

- Sauvegarder ce corps de méthode, puis quitter l'éditeur. Valider le démon ainsi modifié par le bouton 'Valider'.

2.2.4 L'écriture dans le fichier externe

- Dans le menu 'Développement', item 'BASIC DIESE', choisir 'Process', puis 'DiscreteProcess'. Dans la fenêtre de choix, encore vide, cliquer sur 'Créer'.

- Dans la fenêtre de modification, taper ProcessusMemorisationHeuresTravaillees dans le champ 'Name'.

- Dans l'onglet 'Descripteurs', choisir EXPLOITATION dans le menu de la ligne 'ProcessedEntityClassId'.

- Dans l'onglet 'Méthodes', cocher le bouton 'Nom automatique' de la ligne 'SetExec', puis cliquer sur 'Coder'. dans la fenêtre d'édition alors ouverte, coller le § N2_E2.2.4 du fichier ressources_FCT.txt. L'exploitation (pointeur pEA) est l'entité touchée (voir Etape 2.2.1 de cette Annexe). On a choisit une structure d'enregistrement où cohabitent la valeur courante de l'horloge de la simulation et la date calendaire complétée par l'heure. Précédant, bien entendu, les trois valeurs de cumul journalier des heures travaillées. Noter l'usage du service prédéfini GetPhysicalFile, qui désigne le fichier externe que gère la structure interne OutputFile. En fin de code, on remet les compteurs à zéro, initialisant ainsi correctement le cumul du lendemain.

- Sauvegarder le corps de la méthode. Quitter l'éditeur. Valider la construction de ce processus par le bouton 'Valider'.

2.2.5 Le test du codage de la fonction de sortie

- Générer tous les sources. Sauvegarder la base de connaissances.

- Dans le répertoire d'exécution, compiler à nouveau la base de connaissances, puis lancer la commande habituelle, à laquelle on ajoute l'indication d'un fichier de sortie des heures travaillées :

```
>./main SIM/sim1.par SIM/sim1.sim SIM/sim1.osp SIM/sim1.str SIM/sim1.dir -sb -o
-f heuresTravaillees.txt
```

- Se déplacer dans le répertoire des sorties :

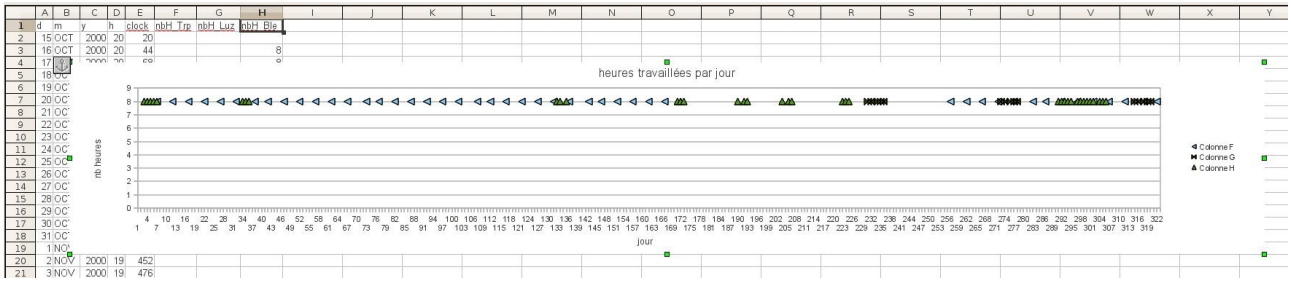
```
>cd ../out
```

- Le fichier heuresTravaillees.txt commence comme ci-dessous, au-delà de la ligne d'entête ⏭ :

```
15 OCT 2000 20 20 0 0 0
16 OCT 2000 20 44 0 0 8
17 OCT 2000 20 68 0 0 8
18 OCT 2000 20 92 0 0 8
19 OCT 2000 20 116 0 0 8
20 OCT 2000 20 140 8 0 8
21 OCT 2000 20 164 0 0 0
22 OCT 2000 20 188 0 0 0
23 OCT 2000 20 212 0 0 0
24 OCT 2000 20 236 0 0 0
25 OCT 2000 20 260 8 0 0
26 OCT 2000 20 284 0 0 0
27 OCT 2000 20 308 0 0 0
28 OCT 2000 20 332 0 0 0
29 OCT 2000 19 356 0 0 0
30 OCT 2000 19 380 8 0 0
```

Comme attendu, il y a un enregistrement par jour, même non travaillé. L'heure de l'écriture[26] est 20h (l'heure de la révision matinale du plan augmentée de 12) jusqu'au passage à l'heure d'hiver ; alors, le retour à 20h prenant 25 heures, nous ne sommes qu'à 19h 24 heures après. Par contre, toutes les valeurs d'horloge (20, 44;=, etc.) sont bien séparées par 24 heures, comme le commande la construction de l'événement autogénéré de trace.

Repris dans un tableur, entre autres outils candidats, ce fichier permet de créer un graphique tel que ci-dessous ⏭ :



[26] Naturellement sans autre intérêt ici que l'illustration d'un effet de la gestion de l'heure d'hiver.

ANNEXE 3 : Ordonnancement global des priorités

On rappelle ici que le développeur introduit des priorités dans le modèle du système étudié. Cette section synthétise les principes d'établissement des priorités, et les valeurs choisies dans le présent développement.

Dans notre modèle en objets, ces priorités sont attachées :
- aux événements : pour ordonnancer des occurrences en un même instant
- aux opérations : pour fixer la priorité des événements d'exécution de la StateTransitionProcedure
- aux activités : pour indiquer au moteur d'allocation de ressources l'activité qui par qui commencer, à un instant donné

Dans notre modèle conceptuel, ces valeurs de priorité servent à contrôler :
- l'ordre dans lequel deux événements, intervenant dans l'environnement dans la même unité d'horloge, impactent le système
- l'ordre dans lequel doivent être invoquées, au même instant simulé, deux procédures de changement d'état du système biophysique, pour respecter la sémantique temporelle des variables d'entrée/sortie des procédures
- la mise en œuvre des opérations conformément à leur opportunité et leur urgence vis-à-vis du système piloté
- le partage des ressources conformément à la pratique du gestionnaire du système

Ces connaissances, relatives aux instants, constituent le modèle dynamique du système étudié. On ne dit pas "relatives au temps", parce qu'un aspect du temps, les durées, sont modélisées dans le modèle fonctionnel

Les priorités posées sur les activités guident un algorithme d'allocation géré par un unique événement sur le système décisionnel.

Les priorités posées sur événements ($P(E_k)$) et sur les opérations ($P(O_k)$) sont d'une autre nature : elles servent à ordonner plusieurs événements dans l'agenda de la simulation. Elles doivent donc être établies conjointement. Autrement dit, il est insuffisant le donner un ordre relatif interne aux $P(E_k)$ et un ordre relatif interne aux $P(O_k)$; il faut aussi positionner les $P(O_k)$ relativement aux $P(E_k)$.

Dans notre développement, nous avons établi le positionnement suivant, par ordre de priorités décroissantes (les nombres en début de ligne sont les valeurs de priorité) :

1 - lecture environnement :
. UDev,
. croissance nette de l'herbe
10 - processus développement blé
35 - mobilisation/immobilisation des unités de MO
40 - révision matinale du plan
45 - établissement du jeu d'activités à acter
50 - établissement de la liste des opérations à exécuter, suspension d'opérations en cours
60, 65, 70 - opérations culturales

Ce qui appelle les commentaires suivants :

- L'ordre de lecture des UDev et de la croissance nette de l'herbe est indifférent. On communique cela par une priorité unique (on laisse le moteur choisir l'ordre des événements dans l'agenda). Deux priorités différentes, bien que très fortes devraient être justifiées.

- La priorité du processus développement du blé est attribuée lors de la création du processus, en conséquence de l'opération SemerBle (StateTransitionProcedure).

- La priorité de la mobilisation/immoblisation des unités de MO reste celle attribuée par défaut par le moteur de simulation. On ne la modifie pas dans l'application. La mobilisation/immobilisation doit intervenir avant la procédure d'allocation des ressources (événement prédéfini MakeInstructionListEvent de priorité 45, voir une des remarques ci-dessous).

- La priorité de la révision matinale du plan reste celle attribuée par défaut par le moteur de simulation. On ne la modifie pas dans l'application.

- Les priorités entre les opérations culturales sont données dans le fichier des paramètres du système (sim1.par). Leurs valeurs sont au moins 60 parce que la documentation de CONTROL DIESE indique que la priorité par défaut de l'événement prédéfini ProceedOperationEvent est justement 60, pour que les instances de cette classe soient programmées après les événements UpdateSituationEvent (ouverture/fermeture d'activités, dont RevisionMatinaleDuPlan est une classe-fille), MakeInstructionListEvent et ActInstructionListEvent).

- Au-delà du degré 70, et en-deçà du degré 35, on laisse des plages suffisamment larges pour accueillir d'autres processus dans un éventuel élargissement de la base de connaissances.

ANNEXE 4 : L'énoncé du problème de modélisation/simulation[27]

Le modèle choisi est celui d'une exploitation (EA) de polyculture élevage dans laquelle travaillent trois agriculteurs. L'exploitant cultive du blé et de la luzerne et dispose de praires dans lesquelles des vaches sont en pâturage permanent.

On distingue 3 ateliers : un atelier Blé, un atelier Luzerne et un atelier Vaches.

Les trois agriculteurs peuvent intervenir sur l'ensemble des ateliers "culture". En revanche, seul un des trois est « spécialisé » vache et peut donc s'occuper de l'atelier Vaches.

Les tâches à accomplir dans chaque atelier sont les suivantes :

Atelier Blé

Les interventions sont datées en jours juliens depuis le 1er janvier 2000, ou se font en fonction des stades de développement du blé. Les données disponibles sont un fichier rassemblant les unités de développement du blé calculées chaque jour du 01/01/2000 au 31/12/2001 : UDEV_BLE.txt.

Intervention	Date de début	Date de réalisation au plus tard	Durée de réalisation pour les parcelles de l'EA ...	... sur la base de N agents
Semis	Jour 289	-	5 jours	2
Désherbage automne	Jour 320	Jour 335	3 jours	1
Azote 1er apport	Jour 418	Jour 433 (*)	3 jours	1
Azote 2ème apport	Stade épi 1 cm soit le jour où UDEV_BLE ~ 2692 unités	Date début + 7 jours (*)	3 Jours	1
Azote 3ème apport	Stade gonflement soit le jour où UDEV_BLE ~ 2900 unités	Date début + 7 jours (*)	3 Jours	1
Fongicide 1	Jour 476	Jour 481 (*)	3 Jours	1
Fongicide 2	Stade floraison soit le jour où UDEV_BLE ~ 3092 unités	Date début + 5 jours (*)	3 Jours	1
Récolte	Jour 567	Date début + 20 jours	NS	2

De manière générale, la date de réalisation au plus tard est celle au-delà de laquelle la tâche n'est plus poursuivie, même si elle n'est pas terminée. Où la date est marquée par (*), c'est celle au-delà de laquelle on abandonne la réalisation si la tâche n'a pas commencé, pour une raison de ressources non disponibles ou pour une autre. Dans ce cas, l'expression "Date début" doit se comprendre "première date de début possible".

Seulement pour le semis, la durée de réalisation peut être augmentée ou diminuée, à surface constante, si le nombre d'agents est diminué ou augmenté.

Atelier Luzerne

Les interventions se font en fonction des dates moyennes de coupe pour la luzerne. Les interventions se font dans l'ordre suivant (jours exprimés en jours juliens depuis le 1er janvier 2000) :

Intervention	Date de début	Date de fin	Durée de réalisation pour les parcelles de l'EA ...	... sur la base de N agents
Année 1				
Coupe 1	Jour 153	Jour 163	7 jours	2
Coupe 2	Jour coupe 1 + 42 jours	Jour coupe 1 + 49 jours	7 jours	2
Année 2				
Coupe 1	Jour 518	Jour 528	7 jours	2
Coupe 2	Jour coupe 1 + 42 jours	Jour coupe 1 + 49 jours	7 jours	2
Coupe 3	Jour coupe 2 + 42 jours	Jour coupe 2 + 49 jours	7 jours	2

La tâche de "coupe" comprend aussi, sans distinction, le fanage et le pressage.

[27] Cet énoncé est une synthèse amendée des textes présentés aux participants de l'atelier MODELISAD tenu les 3 et 4 octobre 2011 à Paris.

On remarque que les jours 153 et 518 correspondent au 5 juin des deux années.

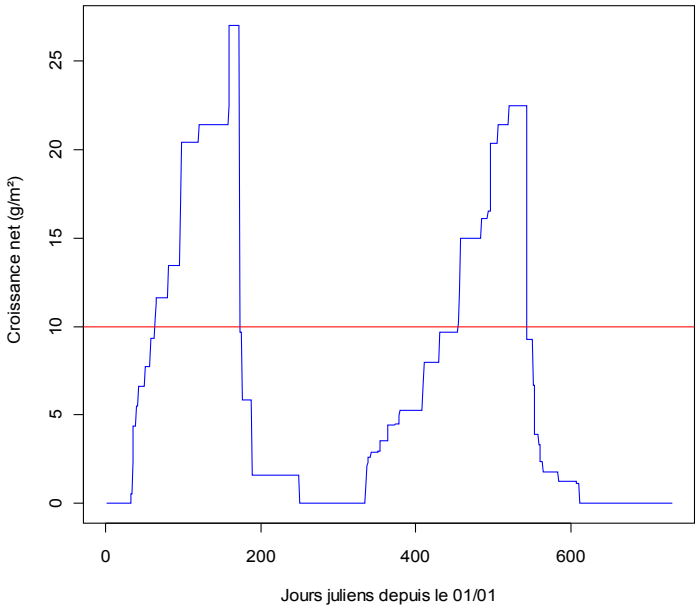
La coupe pourra être réalisée par plus ou moins d'agents, avec une vitesse modifiée proportionnellement. Si la vitesse est diminuée, la date de fin (7 jours après la date de début) pourra être atteinte avant que la coupe ne soit terminée ; celle-ci sera donc arrêtée.

Atelier Vaches

Le troupeau se nourrit au pâturage. Pour cela, la production d'herbe est simulée par un modèle basé sur la croissance de l'herbe.

La croissance nette de l'herbe est simulée dans le fichier DISPO_HERBE.txt. On considère que si la croissance nette est inférieure à 10, l'agriculteur spécialisé "vaches" doit complémenter les animaux. Cette opération se fait tous les 5 jours et lui prend 1 jour.

On peut si on le souhaite modifier le seuil de 10 pour augmenter ou au contraire libérer les contraintes.



La priorité entre les ateliers est : atelier Vaches > atelier Luzerne > atelier Blé.
Les tâches peuvent être interrompues en cas de concurrence avec une autre tâche plus prioritaire.

Objectif : On souhaite simuler le déroulé journalier des interventions sur les trois ateliers conjointement, et observer les différences de déroulés en fonction des ressources disponibles dans l'exploitation.

ANNEXE 5 : Index