Modélisation de systèmes agronomiques dans l'espace économique. Programmation orientée objet du modèle de culture STICS

Jean-Claude Poupa

Modélisation de systèmes agronomiques dans l'espace économique

Programmation orientée objet du modèle de culture STICS

Jean-Claude Poupa

Mars 2010

L'offre de produits agricoles, marchands et environnementaux, est le résultat de décisions d'agents sur des unités de production, exploitations agricoles qui gèrent des campagnes culturales annuelles sur des parcelles. Les décisions sont d'abord relatives à la gestion des assolements en fonction des sols des parcelles, choix qui déterminent les rotations des cultures pour les campagnes suivantes. Les assolements possibles sont définis pour les types d'exploitations à l'échelle d'un territoire, dit petite région naturelle ou pays, caractérisée par un climat, des types de sols dans un contexte géographique et géologique, mais aussi l'existence d'infrastructures en amont et en aval des exploitations agricoles. Les cultures étant en place, les autres décisions sont relatives aux pratiques culturales, avec des opérations quotidiennes qui définissent des itinéraires techniques, prédéfinis dans des calendriers et/ou ajustés en fonctions d'évènements.L'espace économique regroupe ces régions, partition du territoire agricole des nations.

Les modèles de cultures qui représentent les systèmes agronomiques sont définis à l'échelle de la culture sur la parcelle, unité de simulation (*usm* STICS). Cette unité est définie pour une culture, annuelle ou bisannuelle, par les paramètres du sol, de la plante, un itinéraire technique avec des calendriers prédéfinis d'opérations, un climat sous la forme de séries météorologiques. La culture est évaluée pour la campagne, annuelle ou saisonnière pour les cultures intermédiaires (dont l'absence de cultures), les cultures pérennes étant réinitialisées pour la campagne suivante. Au modèle STICS est associé le logiciel éponyme écrit dans le langage Fortran. Il est piloté depuis une interface graphique sous le système WINDOWS, *Winstics*, laquelle gère l'acquisition des paramètres et les sorties demandées, lance

la simulation, édite les bilans aux récoltes ainsi qu'un état pour la culture suivante. Pour effectuer les simulations sur d'autres d'autres systèmes d'exploitation, dont LINUX, il faut importer ces fichiers fabriqués par *Winstics*, lancer la commande d'exécution sous la forme d'un traitement *batch*, enfin récupérer les résultats dans des fichiers bilans.

La simulation des décisions dans l'espace économique est relative d'abord aux choix des cultures, ensuite aux itinéraires techniques pour ces cultures, pour des campagnes successives avec leurs climats, et cela pour les parcelles du territoire. Les méthodes d'évaluation de fonctions de production utilisent par ailleurs ces données simulées pour des ensembles d'itinéraires techniques pour une culture, par exemple des combinaisons de calendriers (fertilisation, irrigation) et choix variétaux. L'utilisation de ces modèles de cultures pour la modélisation économique nécessite donc de pouvoir effectuer de très grands nombres de simulations, mesurés en millions. Pour réaliser ces simulations dans des applications informatiques, une représentation vectorielle a été adoptée et des fonctions d'évaluation définies. Fabriquées par reconnaissance des formes dans le code code source du logiciel STICS, elles sont archivées dans une bibliothèque mathématique. Le langage C s'est imposé comme langage cible pour produire des fonctions mobilisables en informatique scientifique et dans le monde du logiciel libre.

La chaîne de traitement informatique étant aujourd'hui validée, pratiquement un compilateur, *CStics*[1], qui génère une bibliothèque de fonctions à partir du code informatique des agronomes concepteurs de STICS, des applications peuvent désormais être spécifiées et développées pour simuler les agrosystèmes dans différents contextes, dont l'espace économique. La modélisation objet offre un cadre théorique pour produire de telles applications : l'enjeu est de fournir aux équipes de recherches un environnement d'informatique scientifique pour poser et résoudre des problèmes algorithmiques, puis développer des applications au moyen de composants regroupés sur une plate-forme thématique.

Ce document présente d'abord les formalismes, méthodes numériques et langages informatiques, puis les différentes classes construites autour du paradigme de programmation objet pour traduire les algorithmes et réaliser les simulateurs.

[1] **Jean-Claude Poupa**, *Intégration de modèles de cultures sur des plates-formes de modélisation agronomique et environnementale, Application au modèle Stics*, 257p.

1 Modèles de cultures et langages informatiques

Le modèle STICS est défini sous la forme d'un grand système d'équations regroupées en modules, avec en pratique quelque 1500 variables et paramètres utilisés dans 200 équations, les formalismes étant documentés dans un ouvrage.[2] Le logiciel associé fonctionne en mode *traitement par lots* (batch) avec des fichiers en entrée et sortie pour l'unité de simulation, dite *usm*, laquelle constitue une transaction informatique, atomique.

Vu dans l'espace économique, un modèle de culture est vu sous la forme de fonctions qui reçoivent des paramètres et évaluent des variables dans un processus temporel. Paramètres et variables définissent des entités autour des décisions de production, par exemple un itinéraire technique appliqué pour une culture sur une parcelle. D'autres formes sont à construire pour structurer ces données et aboutir à une représentation algébrique utilisable dans les applications informatiques.[3]

1.1 Modèles numériques

Les modèles de culture évaluent des suites d'états quotidiens : ces évaluations peuvent être formalisées dans des fonctions temporelles, vectorielles. Ces fonctions algébriques vont constituer le socle des spécifications formelles sur lesquelles seront développées les applications informatiques.

1.1.1 Forme basique

En regroupant l'ensemble des paramètres dans un vecteur X et les états quotidiens dans Y_t pour la période t, une évaluation peut être traduite au moyen d'une fonction Φ sous la forme :

$$Y_t = \Phi(X, Y_{t-1})$$

Cette première forme, facile à fabriquer, le vecteur Y regroupant les variables évaluées dans la boucle quotidienne, a d'abord permis de maîtriser la dimension temporelle pour pouvoir récupérer à la demande dans des applications les valeurs du jour.

[2] **Nadine Brisson, Marie Launay, Bruno Mary, Nicolas Beaudoin**, *Conceptual basis, formalisations and parameterization of the STICS crop model*, éditions Quæ, 2009.

[3] **Jean-Claude Poupa**, *Du modèle de culture à la modélisation des surfaces cultivées sur le long terme*, *Spatialisation et dynamique des cultures*, décembre 2007.

1.1.2 Séparation des paramètres

Les simulations dans l'espace économique nécessitent l'accès aux familles de paramètres répartis dans des vecteurs, soit le sol S, la plante cultivée V, l'itinéraire technique W et les séries météorologiques M. Si le logiciel STICS lit ces catégories de paramètres dans différents fichiers, d'autres sont évalués dans l'étape d'initialisation dans des formules qui utilisent des pseudo-constantes, paramètres dits généraux, contexte K. A ces paramètres s'ajoutent des conditions initiales relatives à l'état de la parcelle en début de campagne, regroupées dans X.

Pour pouvoir classer ces paramètres, un graphe a été construit pour reconstituer les arbres d'évaluation, graphe essentiel pour structurer l'ensemble des noms symboliques STICS, mais aussi pour pouvoir suivre les processus d'évaluation et retrouver les algorithmes effectifs. La construction de ce graphe a permis de vérifier la cohérence des paramètres de STICS, vu qu'il n'existe pas de sommets successeurs de deux familles disjointes, par exemple un paramètre polymorphe qui dépendrait du sol et de la plante. Avec ces nouveaux vecteurs, une évaluation peut être traduite au moyen d'une fonction Φ sous la forme :

$$Y_t = \Phi(K, S, V, W, M, X, Y_{t-1}).$$

1.1.3 Succession des cultures

Les cultures intermédiaires et la gestion des intercultures sont traduites dans des unités de simulation : la campagne culturale est alors saisonnière.

STICS évalue l'état quotidien de la culture sur la base d'un cycle annuel ($\leq$ 366 jours), étant entendu qu'une culture bisannuelle mobilise les séries météorologiques des années de semis et récolte, paramètres climatiques. Pour les cultures pérennes, la culture est réinitialisée pour le cycle suivant au moyen de valeurs initiales.

Afin d'enchaîner les unités de simulation pour des cultures successives, le logiciel édite dans un fichier un état final de la parcelle, éventuellement relu comme état initial dans l'étape d'initialisation, après acquisition des paramètres, pour la culture suivante : ces données initiales de campagnes sont gérées dans le vecteur X. En introduisant l'indice n de campagne, la fonction d'évaluation s'écrit alors

$$Y_t = \Phi(K, S, V_n, W_n, M - n, X_{n-1}, Y_{t-1}).$$

X_n devra alors être évalué à l'issue de la campagne courante pour être transmis à la suivante, X_0 désignant l'état initial de la parcelle.

1.1.4 Dimension spatiale et long terme

La fonction précédente est appelée sur un ensemble de parcelles, unité de production ou petite région, avec un contexte et un climat. Les processus peuvent alors s'exécuter en parallèle, avec éventuellement des synchronisations annuelles pour récupérer les valeurs de variables économiques ou environnementales. Il est aussi possible d'envisager des synchronisations quotidiennes pour des couplages avec d'autres modèle, pour les relations de voisinage entre parcelles notamment.

La prolongation des simulations sur le long terme nécessite également l'examen des paramètres généraux du vecteur K : c'est par exemple la valeur du taux de CO_2 dans l'atmosphère.

1.1.5 Modularisation

STICS est un modèle multidisciplinaire divisé en modules qui regroupent des équations : la fonction Φ est une composition de n fonctions

$$\Phi = \varphi_1 \circ \ldots \circ \varphi_i \circ \ldots \circ \varphi_n.$$

Les couplages au niveau des évaluations quotidiennes peuvent être spécifiés sous la forme d'une fonction supplémentaire à intégrer dans cette composition, non commutative.

1.2 Langages informatiques

La compilation *CStics* génère des fonctions vectorielles C avec les vecteurs précédents. Ces vecteurs sont déclarés comme réels simple précision, ce qui est suffisant pour les évaluations d'un modèle de culture qui n'effectue pas de calculs aux limites.[4] Les paramètres ou variables STICS entiers, dont les codes textuels (types énumérés), sont convertis en réels. Les fonctions rendent une valeur entière utilisée pour transmettre des informations.

Les unités de programmes Fortran, pour la plupart des sous-programmes (subroutine), sont convertis en fonctions déclarées sous la forme générique

$$\mathsf{int}\ Fstics(\mathsf{float}\ *K, \mathsf{float}\ *S, \mathsf{float}\ *V,\ \mathsf{float}\ *W, \mathsf{float}\ *M, \mathsf{float}\ *X, \mathsf{float}\ *Y),$$

l'opérateur $*$ référençant des tables, ici des vecteurs réels. Ces fonctions sont compilées et archivées dans une bibliothèque. La forme adoptée apparaît, dans les environnements de programmation, comme canonique, l'uti-

[4] Pour les analyses de sensibilité au moyen de méthodes de différentiation automatique, il est préférable de générer une version en double précision.

lisation d'autres types de données, tables multidimensionnelles ou structures, s'avérant complexe et dépendante des langages.

La compilation produit également un ensemble de structures avec des tables des symboles qui décrivent variables et fonctions, des graphes des relations entre variables, les arbres des appels de fonctions, les prototypes de ces fonctions... La vectorisation applique un algorithme qui répartit les paramètres et variables du modèle de culture dans les vecteurs, une même variable informatique STICS pouvant être vectorisée simultanément comme paramètre dans (S, V, W), comme variable dans Y, parfois comme donnée initiale dans X. Un dictionnaire des variables est construit pour préciser pour chaque variable le type, la dimension, les bornes, le(s) vecteur(s) d'allocation et le(s) position(s). *CStics* génère des fichiers C à inclure pour déclarer et initialiser ces structures.

Ces composants informatiques étant désormais disponibles, l'enjeu est maintenant de se doter d'un environnement de développement pour pouvoir les mobiliser dans l'espace économique : la modélisation objet apporte des solutions sur la base des représentations formelles adoptées.

1.2.1 Langage procédural

L'utilisation du langage C demeure indispensable d'abord pour traiter et adapter les versions logicielles STICS. C'est dans ce langage, et dans l'environnement de développement libre du projet GNU, que s'effectuent les validations, les évaluations issues des codes Fortran et C devant être toutes équivalentes pour l'intégralité de l'unité de simulation, le déroulement des algorithmes étant suivi au pas à pas dans les deux langages si nécessaire. Par ailleurs, si des "bugs" subsistent dans les fonctions, c'est dans le code C des fonctions de bibliothèques qu'il faudra réaliser le "débogage" pour ensuite corriger l'erreur, dans le code Fortran ou dans le code C de la chaîne de traitement *CStics*.

La modularisation du code et son optimisation relèvent aussi indirectement de cet environnement de développement : il s'agit d'automatiser la production de différentes versions pour des contextes applicatifs.

Modularisation Les modules sont les fonctions φ de la spécification fonctionnelle. Ces modules peuvent être divisés en sous-modules avec des possibilités de choix liées aux options du modèle : ces sous-modules définissent aussi des fonctions, avec éventuellement des choix conditionnels controlés par des expressions logiques. Pour restituer cette structure modulaire dans le code Fortran, il suffit d'associer un sous-programme à chaque module, ce

qui est déjà fait en partie et est à compléter en éclatant ou regroupant des sous-programmes existants. La compilation *CStics* de cette version modularisée va générer les fonctions C associées aux modules.

Cette organisation modulaire liée à une représentation formelle ne doit pas être confondue avec l'organisation des unités de programmes, soit en Fortran des arbres d'appels avec comme feuilles des unités terminales. Un module peut gérer un arbre d'appels : les fonctions appelées sont présentes dans la bibliothèque mais invisibles dans cette architecture modulaire.

Optimisation L'utilisation dans l'espace économique nécessite une optimisation informatique des temps d'exécution des fonctions élémentaires mais aussi de la taille des vecteurs. Pour un modèle de culture, les évaluations coûteuses sont relatives au système sol-plante : il peut être utile de contrôler le nombre d'itérations et le volume des tables, vectorisées dans Y pour les variables du modèle.

Dans le code Fortran du logiciel STICS, toutes les tables sont déclarées avec des bornes fixes, suffisamment grandes pour ne pas dépasser les limites physiques, reprises dans les boucles : des directives de compilation *CStics* permettent d'ajuster ces bornes pour produire un code plus performant.[5]

Des séries temporelles sont ainsi définies sur une période bisannuelle alors que l'état du jour ne dépend que de celui de la veille. Un autre mode d'allocation, dit *Retro*, peut être demandé pour ces variables au moyen de directives *CStics* pour ne conserver que les périodes utiles, avec un décalage pour la période suivante : des versions sur mesure peuvent alors être générées en regard des options de simulation.[6]

1.2.2 Langage objet

La modélisation objet offre un cadre théorique pour mettre en relation les formalismes STICS, un grand système d'équations, et les structures algébriques de l'espace économique, des vecteurs et séries temporelles. Les changements d'échelles des systèmes de cultures aux décisions des agents économiques font qu'il faut se doter d'un langage thématique pour piloter de grands nombres d'évaluations pour les séries de cultures sur les parcelles des territoires, langage que permet de construire la modélisation objet.

Le modèle objet doit ensuite être traduit au moyen d'un langage objet, lequel doit permettre :

[5] La profondeur du sol est par exemple réduite de 1000 à 200, soit 2 mètres.

[6] Ce mode est utilisé pour réduire la taille des matrices du système racinaire, 731×1000 dans le code Fortran.

1. l'intégration des fonctions compilées de bibliothèques au moyen de méthodes dites natives, pour pouvoir exécuter le code C natif ;
2. la production d'un code exécutable performant du fait de la complexité calculatoire ;

Java a finalement été retenu pour réaliser les prototypes de ces applications.

Java Le langage *Java* est largement utilisé pour des applications en mode interprété (bytecode) mais le code peut aussi être compilé au moyen du compilateur *gcj*, élément de la *GNU Compiler Collection (GCC)*, pour produire un code exécutable. Il permet de construire des classes pour intégrer des bibliothèques de fonctions C au moyen de méthodes dites natives, lesquelles gèrent les correspondances entre variables dans les deux langages pour les types numériques et les tableaux de ces types, donc les vecteurs.

C'est un pur langage objet au sens où tout est objet et appartient à une classe (instance), à une nuance près pour les types numériques, utilisables comme tels ou dans leurs classes enveloppes : c'est un moyen pour garantir la conformité des règles et algorithmes et produire des applications de qualité. Il est largement présent dans les formations universitaires. De grands ensembles de classes ont été développées et sont accessibles sous forme de paquetages : interfaces graphiques (`java.swing.*`), bases de données...

Java intègre aussi la synchronisation des processus et le parallélisme.

Notations

Ce document reprend les règles d'écriture *Java*.
Un identificateur est déclaré derrière son type, avec en suffixe les crochets pour les table et les parenthèses pour les fonctions.
L'usage est de noter les classes avec la première lettre majuscule, les variables et fonctions avec une minuscule.
La double barre oblique // place le reste de la ligne en commentaires.

1.3 Fabrication logicielle

La chaîne de traitement démarre avec le code informatique des chercheurs qui, dans le cadre d'un groupe de référents STICS, introduisent de nouveaux formalismes de modélisation de processus pour la plante ou le système sol-plante. Ces formes sont définies dans des équations avec les noms symboliques partagés par la communauté STICS, équations immédiatement traduites sous la forme d'expressions arithmétiques et logiques dans le langage Fortran 77. La gestion de ces noms symboliques utilise d'anciennes structures, les communs Fortran qui regroupent toutes les déclarations des

variables informatiques dans une zone mémoire commune, accessible à tous les modules en lecture et écriture.

C'est ce langage partagé par les référents STICS qui est traité par le compilateur *CStics* pour :

1. fabriquer les vecteurs à partir des communs Fortran 77 ;
2. construire les fonctions vectorielles par reconnaissance syntaxique ;

La fabrication logicielle va ensuite s'effectuer au moyen de ces composants, structures de données et fonctions restituées dans le langage C et mobilisables dans des applications, procédurales ou objets, et depuis les logiciels scientifiques.

1.3.1 Validation du code C

Le logiciel STICS est maintenu et évolue avec des ajouts, et périodiquement de nouvelles versions. La validation du code C produit s'effectue avec les outils de développement classiques de la communauté LINUX, dans un environnement d'informatique scientifique dans le monde du logiciel libre.

L'étape de validation s'effectue avec une version STICS C rigoureusement équivalente à la version Fortran du point de vue de toutes les évaluations numériques en cours de simulation, le débogage s'effectuant par des exécutions en parallèle C et Fortran pour localiser les écarts éventuels.[7]

La fabrication de la version C utilise :

1. les productions de *CStics*, soit
   - le code C généré sous la forme de fonctions dans des fichiers sources pour chaque unité de programme Fortran ;
   - des fichiers d'en-tête de déclarations C pour la gestion des structures de données, dont le dictionnaire des noms STICS ;
2. des fonctions C indépendantes des versions Fortran, développées pour piloter l'exécution et valider le code généré, avec
   - un programme principal C et sa suite qui reprennent l'architecture logicielle Fortran d'exécution de l'unité de simulation ;
   - des fonctions d'accès aux formalismes STICS et utilitaires de mise au point pour gérer la vectorisation ;

Les exécutions C et Fortran utilisent les mêmes fichiers de paramètres et éditent les mêmes bilans.

[7] Les écarts liés à la gestion de la précision par les compilateurs C et Fortran demeurent inférieurs à 10^{-6} en fin de simulation.

1.3.2 Construction des classes des langages objets

Les applications objets utilisent :

1. les productions de *CStics*, soit
   - les fonctions C compilées, archivées dans des bibliothèques accessibles dans des répertoires avec les chemins d'accès du système hôte ;
   - les structures de données produites par *CStics* pour gérer les formalismes STICS, restituées dans des fichiers ;
2. le code C d'application des méthodes natives pour l'exécution sous le système d'exploitation hôte des fonctions de bibliothèque STICS, soit
   - le fichier d'en-tête qui déclare les prototypes des fonctions natives (produit en langage *Java* par la commande `javah`) ;
   - les fonctions C natives développées pour la communication entre les formes procédurales et objets ;
   - deux fonctions C supplémentaires appelées par les fonctions natives pour reconnaître les fichiers de paramètres STICS ;

Interface graphique de restitution des formalismes STICS

Une classe est définie pour :

1. visualiser l'ensemble des noms des paramètres, variables, fonctions dans des boîtes de listes pour pouvoir les sélectionner ;
2. retrouver les équations dans un pseudo-code C qui restitue les algorithmes d'évaluation du logiciel source ;

Cette classe utilise les structures de données produites par *CStics* et la liste des fonctions natives. Le pseudo-code est généré pour chacune des fonctions par une option de *CStics*, et restitué sous un éditeur à la demande.

2 Classe EvalStics

La classe *EvalStics* définit les méthodes qui fournissent une interface pour exécuter le code natif des fonctions de la bibliothèque. Elle va permettre de construire un objet au moyen de la méthode *loadLibrary()* de la classe *System* du paquetage `java.lang`, objet statique qui charge la bibliothèque dans l'environnement d'exécution du système d'exploitation hôte.

2.1 Méthodes natives

EvalStics déclare un ensemble de méthodes natives d'accès aux fonctions C et charge la bibliothèque dans un bloc d'initialisation statique au moyen de la méthode

```
System.loadLibrary("bibStics");
```

La bibliothèque *bibStics* est créée dans l'environnement de développement GNU sous LINUX sous le nom `LibbibStics.so`[8] dans un répertoire d'accueil des bibliothèques, le chemin d'accès étant ajouté dans la variable d'environnement LD_LIBRARY_PATH : *bibStics* contient le code compilé d'un ensemble de quelque 200 fonctions C, générées par *CStics* ou développées.

Les méthodes proposées reprennent les étapes de la simulation et les modules d'évaluation à l'issue de quelques restructurations minimales dans le code Fortran.[9] Les noms de ces méthodes sont préfixés par le radical *stics*.

2.1.1 Arguments des méthodes

Les méthodes sont déclarées avec les vecteurs du modèle comme arguments soit la liste <*listeVecteurs*>

```
float contexte[] ; // K
float sol[] ; // S
float iniParcelle[] ; // X
float plante[] ; // V
float travail[] ; // W
float meteo[] ; // M
float etatCulture[] ;// Y
```

Ces arguments sont des tableaux à une seule dimension déclarés avec les types primitifs, `float` ou `double` pour les nombres réels.

[8] Sous WINDOWS, cette bibliothèque serait créée sous le nom `bibStics.dll`.

[9] Annexe A.3 de la documentation *CStics* (note 1).

Les arguments textuels, nécessaires pour l'accès aux fichiers, sont des objets de la classe *Java* `String`.

Toutes ces méthodes sont définies avec le type primitif entier `int` comme valeur de retour pour pouvoir communiquer avec les objets qui les utilisent.

2.1.2 Acquisition des paramètres et vectorisation

Le code C de lecture des paramètres de l'*usm* dans les sept fichiers WINSTICS peut s'exécuter dans deux méthodes qui vont permettre d'initialiser les premiers vecteurs, ensuite gérés dans l'application objet. Le protocole d'acquisition de l'application Fortran est conservé à l'identique pour pouvoir intégrer l'existant sous WINSTICS :

1. `native int` *sticsIniUsm*(`String` *appli*, <*listeVecteurs*>) ;
   lit successivement les fichiers `travail.usm`, `param.par`, `paramv6.par`, `param.sol`, les paramètres de la plante et techniques avec les noms lus dans `travail.usm`, enfin l'entête du fichier climatique `stat.dat`.
2. `native int` *sticsIniMeteo*(`String` *appli*, <*listeVecteurs*>) ;
   lit dans le fichier `stat.dat` les 5 ou 7 séries météorologiques annuelles ou bisannuelles pour les périodes fournies.

Ces deux fonctions natives utilisent les fonctions *IniUsm*() et *IniMeteo*() pour piloter l'acquisition des paramètres avec les fichiers STICS tels que gérés dans le logiciel : un journal de lecture est édité dans le fichiers `winstics`.

Notons que ce code C de lecture gère la multivectorisation pour des valeurs lues comme paramètres (K, S, V, W) ou données initiales (X) de l'*usm* puis reprises comme variables du modèle (Y) lors des évaluations, gérées dans une même variable informatique dans le code Fortran.

2.1.3 Initialisation de la culture

Le code d'initialisation de la culture avant les évaluations quotidiennes est repris dans deux méthodes. Une autre méthode reprend le code C développé pour reconstituer l'état initial pour une nouvelle simulation du fait de la multivectorisation, soit affecter aux variables concernées dans le vecteur Y les valeurs conservées dans les autres vecteurs (K, S, V, W, X).

1. `native int` *sticsVariables*(<*listeVecteurs*>) ;
   initialise le vecteur Y des variables du modèle lues comme paramètres.
2. `native int` *sticsParcelle*(`String` *fichier*, <*listeVecteurs*>) ;
   initialise le sol de la parcelle et peut éditer des avertissements.

3. `native int` *sticsCampagne*(`String` *fichier*, <*listeVecteurs*>) ;
   initialise la campagne sur la période de simulation et peut éditer des avertissements.

2.1.4 Evaluation quotidiennes

Ces méthodes implémentent les fonctions vectorielles φ évaluées quotidiennement : elle reprennent le code C des fonctions générées par *CStics*, d'abord une fonction d'initialisation pour actualiser les séries temporelles gérées dans le mode *Retro* du langage procédural (1.2.1), ensuite toutes les fonctions issues des sous-programmes Fortran associés aux modules. Toutes ces méthodes sont définies avec comme argument la période, rang de l'itération dans la boucle des évaluations quotidiennes.

0. `native int` *sticsRetro*(`int` *jour*, `float` *etatCulture*[]) ;
   décale les périodes des variables temporelles gérées en mode *Retro*.
1. `native int` *sticsDeveloppement*(`int` *jour*, <*listeVecteurs*>) ;
   évalue le développement de la plante du semis à la récolte.
2. `native int` *sticsPhotosynthese*(`int` *jour*, <*listeVecteurs*>) ;
   traite la photosynthèse, évalue la surface foliaire, intègre les opérations qui réduisent cette surface (effeuillage) ;
3. `native int` *sticsCroissance*(`int` *jour*, <*listeVecteurs*>) ;
   évalue la croissance aérienne et racinaire.
4. `native int` *sticsEtatParcelle*(`int` *jour*, <*listeVecteurs*>) ;
   traite le tassement du sol et évalue les décisions de semis et récolte.
5. `native int` *sticsApports*(`int` *jour*, <*listeVecteurs*>) ;
   intègre les apports, irrigation et fertilisation, et les résidus de cultures dans les opérations de traval du sol.
6. `native int` *sticsOrganique*(`int` *jour*, <*listeVecteurs*>) ;
   évalue l'état de surface et la minéralisation de la matière organique.
7. `native int` *sticsBesoinEau*(`int` *jour*, <*listeVecteurs*>) ;
   évalue les besoins en eau.
8. `native int` *sticsAzoteEau*(`int` *jour*, <*listeVecteurs*>) ;
   effectue les bilans azote et eau.
9. `native int` *sticsChaleur*(`int` *jour*, <*listeVecteurs*>) ;
   effectue le bilan thermique en surface et dans le sol ;
10. `native int` *sticsFinal*(`int` *jour*, <*listeVecteurs*>) ;
    évalue des variables récapitulatives finales.

11. `native int` *sticsCoupe*(`String` *fichier*, `int` *jour*, <*listeVecteurs*>) ; traite les coupes périodiques des cultures fourragères fauchées et édite un bilan pour ces coupes.

2.1.5 Edition des bilans après récoltes

Un bilan récapitulatif est édité en fin de simulation, sauf pour les prairies fauchées pour lesquelles ces bilans sont édités à l'issue des coupes.

`native int` *sticsBilan*(`String` *fichier*, `int` *jour*, <*listeVecteurs*>) ;

effectue les sorties sous la forme listing.

2.1.6 Rotation des cultures

L'état de la parcelle à l'issue de la culture courante est repris pour initialiser l'unité de simulation suivante.

`native int` *sticsRotation*(`int` *campagne*, <*listeVecteurs*>) ;

réinitialise le vecteur Y avec les valeurs des paramètres requis et effectue les transferts des données de la culture courante pour la culture suivante dans les vecteurs X et Y.[10]

2.2 Implémentation des méthodes natives

Le langage *Java* offre des bibliothèques avec des fonctions qui permettent d'exécuter du code natif développé en langage C, ensemble de fonctions désigné sous le terme JNI (Java Native Interface). Les méthodes natives étant déclarées, la commande *javah* appliquée à la classe *EvalStics* génère un fichier entête en langage C, `EvalStics.h`, qui contient les trames des fonctions C à développer pour implémenter les méthodes natives.

Les noms des méthodes sont repris comme noms des fichiers pour cette implémentation : à *sticsXxx()* va correspondre le fichier `sticsXxx.c`. Ces noms sont utilisés pour identifier les fonctions C associées dans `EvalStics.h`, soit *Java_EvalStics_sticsXxx()* pour appeler les fonctions natives qui contiennent le code natif à exécuter.

2.2.1 Correspondance entre les types Java et C

Aux types primitifs *Java* sont associés des types *C* préfixés par la lettre `j`, soit `jint` dans le code *C* pour le type *Java* `int`. Pour les tableaux à une

[10] Les fichiers temporaires Fortran sont remplacés par un vecteur transmis en paramètre.

dimension de ces types primitifs est ajouté le suffixe `Array`, soit `jfloatArray` dans le code C pour le type *Java* `float[]`, `jdoubleArray` pour `double[]`... Quant aux chaînes de caractères requises pour les accès fichiers, elles sont dans le code C de type `jstring`.

Les fonctions qui implémentent ces méthodes natives utilisent ces types pour récupérer des pointeurs sur les tableaux *Java*, et les transmettre comme paramètres effectifs des fonctions natives, les vecteurs du modèle. Ces pointeurs sont fournis et libérés par les fonctions JNI, soit pour le type `float` :

- `float*` *GetFloatArrayElements*(`JNIEnv*`,`jfloatArray`, `jboolean*`) pour l'acquisition du pointeur ;
- `void` *ReleaseFloatArrayElements*(`JNIEnv*`,`jfloatArray`, `float*`,`int`) pour sa libération ;

Soit par exemple le vecteur des paramètres de la plante

- `float` $V[]$ dans le code C,
- `float` *plante*[] dans le code *Java*,

et les variables dans la fonction *Java_EvalStics_sticsXxx()*

- `JNIEnv*` *env*, structure qui gère les références vers les fonctions JNI,
- `jboolean` **copie* et `jint` *mode = 0* des variables locales.

Le pointeur sur le tableau *Java plante*[] est récupéré par l'affectation

$$*V = (*env) -> GetFloatArrayElements(env,plante,\&copie) ;$$

et libéré par l'appel

$$(*env) -> ReleaseFloatArrayElements(env,plante,V,mode) ;$$

Cette solution présente l'avantage d'être relativement générique, une application pouvant intégrer d'autres vecteurs, en simple ou double précision.

2.2.2 Fichiers informels

Les fonctions C issues des sous-programmes Fortran exécutent les opérations de lecture et d'écriture dans les fichiers au moyen de formats descriptifs de champs dans des enregistrements, tables de caractères. Ces opérations, traditionnellement appelées entrées-sorties dans le langage Fortran, sont empiriques, les enregistrements n'étant pas des structures de données générées par une grammaire formelle : n'exécutant pas un algorithme prouvable, un risque d'erreur, aussi faible soit-il, demeure. Cela se traduit lors de l'exécution de l'unité de simulation de différentes façons.

1. L'erreur est reconnue et traitée dans le code natif : des avertissements sont édités en interactif ou dans le fichier historique comme pour l'application Fortran.

2. Des anomalies liées au contexte d'utilisation (système hôte pour l'interprétation des formats, paramètres d'une ancienne version...) font que les valeurs ne sont pas affectées aux bons paramètres : l'unité de simulation s'exécute néanmoins et fournit des résultats aberrants sans qu'il soit possible de localiser ces erreurs initiales.
3. Une erreur à l'exécution non contrôlée survient dans le code natif : l'objet de la classe *java.lang.Runtime* qui pilote l'application perd le contrôle, ce qui se solde par une suite d'avertissements du système hôte avec une copie de l'image mémoire.

Le débogage du code natif est alors à effectuer depuis une application C pour le jeu de paramètres concerné, le code Fortran pouvant être exécuté simultanément dans l'environnement de développement procédural.

Initialisation des applications Java Il résulte de ce qui précède que l'utilisation de ces méthodes qui encapsulent les fonctions d'entrées-sorties *Stics* n'est pas recommandée pour du calcul intensif, l'acquisition des paramètres et les restitutions des variables devant s'effectuer avec d'autres méthodes, disponibles dans d'autres classes. Elles sont cependant nécessaires pour initialiser les applications à partir de l'existant et visualiser les bilans.

Gestion des fichiers de paramètres Les fichiers WINSTICS sont gérés dans le code C natif : les fonctions appelées dans les méthodes *sticsIniUsm()* et *sticsIniMeteo()* reçoivent comme paramètre le chemin d'accès au répertoire contenant ces fichiers, avec les noms réservés ou lus.

Edition des bilans Les fichiers de sortie sont gérés dans les fonctions appelantes associées aux méthodes *sticsBilan()* et *sticsCoupe()* : les noms sont transmis comme arguments des méthodes. Ces fichiers, associés aux unités de simulations, sont écrasés s'ils existent.

Edition des avertissements Le fichier dit historique est conservé pour l'étape d'initialisation et géré comme précédemment dans les fonctions appelantes associées aux méthodes *sticsCampagne()* et *sticsParcelle()*.

Pour les évaluations quotidiennes, c'est la sortie standard du langage C *stdout* qui est transmise en paramètre aux fonctions natives : les messages sont en conséquence affichés dans la fenêtre d'exécution.

2.2.3 Appels des fonctions natives

Les variables C étant initialisées, une ou plusieurs fonctions natives sont appelées. La liste suivante reprend les fichiers sources `sticsXxx.c` et énumère les appels : les noms des sous-programmes Fortran sont conservés et toutes les fonctions sont déclarées avec la première lettre majuscule.

- `sticsVariables.c` : *IniY()* ;
- `sticsParcelle.c` : *Initsimul()* ;*Initsol1()* ;*Initsol2()* ; *Initnonsol()* ;*Initcycle* ;
- `sticsCampagne.c` : *Iniclim()* ;
- `sticsDeveloppement.c` : *Develop()* ;
- `sticsPhotosynthese.c` : *Photosynthese()* ;
- `sticsCroissance.c` : *Croissance()* ;
- `sticsEtatParcelle.c` : *Etatparcelle(()* ;
- `sticsApports.c` : *Apports()* ;
- `sticsOrganique.c` : *Etatsurf()* ; *Mineral()* ;
- `sticsBesoinEau.c` : *Beaupl()* ;
- `sticsAzoteEau.c` : *Azoteeau()* ;
- `sticsChaleur.c` : *Microclimat()* ;
- `sticsFinal.c` : *Final()* ;
- `sticsCoupe.c` : *Coupe()* ;
- `sticsBilan.c` : *Bilan()* ;
- `sticsRotation.c` : *CultSuiv()* ;*IniY()* ;*CultPrec()* ;
- `sticsIniUsm.c` : *IniUsm()* ;
- `sticsIniMeteo.c` : *IniMeteo()* ;

Toutes ces fonctions sont utilisées dans les applications C, sauf les deux dernières *IniUsm()* et *IniMeteo()* avec les appels

- `IniUsm.c` : *Lecusm()* ;*Lecparam()* ;*Lecparamv6()* ;*Lecsol()* ; *Lectech()* ; *Lecplant()* ;*Lecconscli()* ;
- `IniMeteo.c` : *Lecstat()* ;

3 Classe SymboleStics

La classe *SymboleStics* gère les formalismes du modèle *Stics* au moyen d'un dictionnaire qui reprend l'ensemble des variables informatiques communes utilisées dans le logiciel. Les structures reconnues par *CStics* sont restituées dans des fichiers repris pour initialiser les objets.

3.1 Données

Constantes

Les fichiers utiles produits par *CStics* sont recopiés dans le répertoire *SticsAlgo*, d'où les chemins d'accès sous LINUX :

- `String` *fichierEspace* = "../SticsAlgo/Stics.espace" ;
- `String` *fichierDic* = "../SticsAlgo/Stics.dictionnaire" ;

Dictionnaire

`VarStics` *dico*[] est le dictionnaire *Stics*, tableau avec pour élément un structure simple qui regroupe les attributs des variables informatiques définis dans la classe interne `VarStics`. Ces attributs sont :

- `String` *nom* : identificateur de la variable ;
- `int` *type* : entier (1), réel (2), logique (3), code textuel (4) ;
- `int` *dim* : variable simple (0), vecteur (1), matrice (2) ;
- `int` *inf1* : borne inférieure dimension 1 ;
- `int` *sup1* : borne supérieure dimension 1 ;
- `int` *inf2* : borne inférieure dimension 2 ;
- `int` *sup2* : borne supérieure dimension 2 ;
- `int` *n* : numéro du vecteur qui gère la variable ;
- `int` *p* : position (première composante) dans ce vecteur ;
- `int` *nn* : numéro du second vecteur si double vectorisation ;
- `int` *pp* : position dans ce vecteur ;
- `int` *nnn* : numéro du troisème vecteur si triple vectorisation ;
- `int` *ppp* : position dans ce vecteur ;
- `int` *retro* : si > 0 nombre de périodes pour le mode *Retro* ;

`int` *nstics* contient le nombre de variables.

Vectorisation

- *K_,S_,V_,M_,W_,X_,Y_* désignent les vecteurs, numérotés de 1 à 7 ;
- *CAMPAGNE_ = 366* est la durée maximale d'une campagne ;
- `int` *espaceVec*[] gère les tailles des vecteurs.
- `int` *retro*[] contient le rang des variables gérées en mode *Retro*.
- `int` *nretro* dénombre ces variables.

3.2 Gestion du dictionnaire

`void` *lireDico()* initialise le dictionnaire par lecture des fichiers puis appelle la méthode *listeRetro*() pour gérer le mode *Retro*.

Des méthodes permettent d'accéder aux variables informatiques *Stics* et aux composantes des vecteurs dans lesquelles ces variables sont gérées.

1. **Position dans le dictionnaire**
   - `int` *rangDicoDepuis*(`String` *nom*,`int` *avant*) examine si la variable de rang *avant* est la variable recherchée, sinon effectue la recherche dichotomique entre le début et la fin du dictionnaire, et rend le rang de la variable ou 0 si elle n'existe pas.
   - `int` *rangDico*(`String` *nom*) appelle *rangDicoDepuis(nom,1)*.
2. **Position dans le vecteur**
   - `int` *pVariable*(`int` *idic*,`int` *ivecteur*) rend la position de la variable de rang *idic* dans le vecteur numéro *ivecteur*.

Ces méthodes gèrent la multivectorisation, un identifiant *Stics* pouvant désigner, selon le contexte, un paramètre ou une variable.[11]

3.3 Accès aux vecteurs

Une grammaire simple a été adoptée dans le contexte procédural pour représenter les vecteurs sous la forme tabulaire et accéder aux paramètres :

$< nomvariable >< valeur >$ pour les variables *Stics* simples ;
$< nomvariable > [< indice >] < valeur >$ pour les vecteurs ;
$< nomvariable > [< indice >][< indice] < valeur >$ pour les matrices ;

Afin de gérer les plages d'indices avec des valeurs constantes, $< indice >$ peut se dériver sous la forme $< inf >–< sup >$.[12]

Tous les vecteurs sont nuls par défaut.

Lecture des vecteurs

`void` *lireVecteur*(`String` *fichier*, `int` *ivecteur*, `float` *v*[])
traite le fichier désigné. Les variables sont recherchées dans le dictionnaire et, après vérification de leur vecteur d'appartenance numéro *ivecteur*, les valeurs sont affectées dans *v*.

`float` *lireValeur*(`String` *nombre*) reconnait les formats usuels des nombres en mode décimal ou flottant.

[11] Des "paramètres" du sol (S) sont modifiés au cours des évaluations quotidiennes (Y) lors des opérations culturales, et les valeurs finales restituées avec l'état de la parcelle (X).

[12] "`Infil[1-5] 50.0`" signifie que ce paramètre vaut 50 pour les 5 horizons du sol.

Copie des vecteurs

`void` *lireVecteur*(`String` *fichier*, `int` *ivecteur*, `float` *v*[])
édite dans le fichier désigné les variables vectorisées en position *ivecteur* dans v : les valeurs non nulles sont copiées par application des règles de la grammaire adoptée.

4 Classe CultureStics

La classe *CultureStics* construit des objets qui gèrent l'unité de simulation (*usm*) : l'exécution d'une instance fournit des résultats équivalents aux exécutions Fortran ou C.

CultureStics offre des méthodes pour pouvoir démarrer avec l'existant, les fichiers WINSTICS, et poursuivre avec les structures de données vectorielles. La dynamique de la simulation est traduite au moyen des méthodes natives qui exécutent le code natif C des fonctions de bibliothèques. Des méthodes utilitaires sont également fournies pour accéder aux formalismes, et exploiter les tables de paramètres existantes (variétés et types de sols).

4.1 Objets et données

Pour pouvoir effectuer les évaluations et reconnaître les formalismes, tout objet de la classe *CultureStics* doit disposer des fonctions d'évaluation et du dictionnaire, soit les objets

- `EvalStics` *usmStics* ; // fonctions d'évaluation
- `SymboleStics` *dicStics* ; // dictionnaire STICS

Les structures de données de base sont les vecteurs (K, S, V, M, W, X, Y) de la culture, représentés dans le code *Java* par les tableaux :

- `float` *contexte*[] ; //paramètres généraux (pseudo-constantes) K
- `float` *sol*[] ; // paramètres du sol de la parcelle S
- `float` *plante*[] ; //paramètres de la plante V
- `float` *meteo*[] ; // séries météorologiques M
- `float` *travail*[] ; // itinéraire technique W
- `float` *iniParcelle*[] ; // état initial de campagne de la parcelle X
- `float` *etatCulture*[] ; // état quotidien de la culture Y

Les tailles de ces tableaux sont notées dans les variables

- `int` $dimK, dimS, dimV, dimM, dimW, dimX, dimY$;

Plusieurs paramètres *Stics* sont repris pour contrôler l'exécution :

- `int` *jourInitial* ; //jour début simulation (*Iwater*)
- `int` *jourFinal* ; //jour fin simulation (*Ifwater*)
- `int` *finCulture* ; // rang de la dernière simulation (*Maxwth*)

– `int` *finRecolte* ; // ultime période de récolte (*Nrecbutoir*)
– `int` *codeFauche* ; // contrôle coupes cultures fauchées

Une table gère les listes de variables *Stics* à restituer après simulation :
– `String` *sortieStics*[] ;

Les constantes climatiques lues en entête sont conservées pour construire les vecteurs M des campagnes, soit
– `int` *nometp_*, image numérique d'un code textuel ;[13]
– `float` *alphapt,parsurrg,latitude* ;

Des identifiants des paramètres sont gérés dans des chaînes textuelles :
– `String` *nomVariete* gère la liste des groupes variétaux ;
– `String` *nomSol* contient la liste des types de sols ;
– `String` *nomClimat* est le nom lu avec les paramètres climatiques ;

Initialisation des vecteurs L'allocation des vecteurs est suivie d'une mise à zéro, opération effectuée par application de la méthode :
– `void` *zeroStics*() ;

4.2 Acquisition des paramètres

La création des objets de la classe `CultureStics` démarre avec les sept fichiers WINSTICS de la version Fortran en cours dans un répertoire : les vecteurs, paramètres (K, S, V, M, W) et état initial de la parcelle (X), sont immédiatement édités avec la grammaire de la classe `SymboleStics` (3.3). L'option de prise en compte de la culture précédente avec lecture d'un fichier issu d'une autre simulation n'est pas reprise sous cette forme, les rotations étant gérées pour les cultures successives dans la classe `ParcelleStics`.[14]

4.2.1 Construction des vecteurs

Plusieurs méthodes sont proposées pour une reprise complète de l'existant, suivie d'éditions sous la forme vectorielle.

Paramètres de l'unité de simulation
– `void` *iniUsm*(`String` *chemin*)
contrôle l'appel de la méthode native *usmStics.sticsIniUsm()* ;
la fonction native *sticsIniUsm()* (2.1.2) appelle la fonction C *IniUsm()*, laquelle édite les codes textuels dans "`Code_`".
– `int` *existeUsm*(`String` *chemin*)
vérifie l'existence du répertoire pour la culture.

[13] Ce code est nul dès lors que la valeur n'est pas citée dans le code source Fortran.
[14] L'option CODESUITE du logiciel source est ici inopérante.

– `void` *iniMeteo*(`String` *fichier*)
contrôle l'appel de la méthode native *usmStics.sticsIniMeteo()* ; l'argument est le chemin d'accès avec le nom du fichier climatique.[15]
– `void` *vectoCulture* (`String` *chemin*)
édite les vecteurs dans les fichiers "`K_`","`S_`","`V_`","`M_`", "`W_`", "`X_`" du répertoire de la culture.

Sous-ensembles de paramètres des unités de simulation

Les fichiers *winstics* contiennent des ensembles de groupes variétaux pour les paramètres de la plante, une liste de types de sols pour les paramètres du sol, une ou deux années climatiques

1. **Variétés**
   – `void` *varieteUsm*(`String` *chemin*)
   reprend la liste de groupes variétaux lus ($\leq$ 12), et copie les paramètres dans des fichiers identifiés par les noms variétaux suffixés par "`.v`".
   La liste des noms est copiée dans la variable textuelle *nomVariete*, et éditée dans le fichier `V_variete`.
2. **Sols-types**
   Les paramètres du sol sont lus dans le fichier "`param.sol`".
   – `int` *typeSolUsm*(`String` *chemin*)
   lit le fichier des paramètres du sol et copie les sols-types dans des fichiers identifiés par les noms des sols suffixés par "`.s`".
   La liste des sols est copiée dans la variable textuelle *nomTypeSol* et éditée dans le fichier `S_type`.
3. **Station météo**
   – `int` *meteoUsm*(`String` *chemin*)
   relit les données climatiques et copie les séries annuelles dans des fichiers identifiés par le nom de la station *nomMeteo* suffixé par l'année lue, soit un fichier pour les cultures annuelles et deux pour les bisannuelles. *Julfin*, variable *Stics* qui code le dernier jour lu, est ajoutée en fin de fichier.
   La méthode effectue des contrôles de formats effectifs.[16]

[15] Le fichier réservé "`stat.dat`" peut ainsi être renommé pour les années climatiques.
[16] L'interprétation des formats est ambiguë, tout écart pouvant aboutir à une valeur erronée sans la moindre alerte.

4.2.2 Acquisition des vecteurs

Les vecteurs sont lus au moyen de la méthode *lireVecteur()* de la classe `SymboleStics` (3.3), appliquée dans des méthodes définies pour chacun des vecteurs avec un nom de fichier en argument. Ces vecteurs sont remis à zéro avant acquisition des valeurs par analyse syntaxique du contenu du fichier.

- `void` *lireContexte*(`String` *fichier*) ;
- `void` *lireSol*(`String` *fichier*) ;
- `void` *lirePlante*(`String` *fichier*) ;
- `void` *lireTechnique*(`String` *fichier*) ;
- `void` *lireMeteo*(`String` *fichier*) ;
- `void` *lireEtatParcelle*(`String` *fichier*) ;

4.2.3 Gestion des listes de paramètres et variables

La table *sortieStics*[] est initialiée au moyen de la méthode qui lit les noms dans un fichier et rend le nombre, soit

- `int` *sortieVariables*(`String` *fichier*) ;

4.3 Exécution de l'unité de simulation

Les méthodes natives de la classe `EvalStics` sont appliquées pour enchaîner les autres étapes de l'unité de simulation STICS.

1. **Initialisation de l'unité de simulation** (2.1.3)
   - `void` *iniCulture*(`String` *journal*) ;
     contrôle l'étape d'initialisation et récupère les paramètres de contrôle : *finCulture*, *finRecolte*, *codeFauche* ;
2. **Simulations quotidiennes** (2.1.4)
   - `int` *evalCulture*() ;
     pilote la boucle des évaluations quotidiennes :
     - `int` *evalQuotidien*(`int` *jour*) ;
       exécute les méthodes natives liées aux modules à la période *jour*, et rend 0 en cas d'erreur ;
   - `int` *evalCulture*(`String` *bilan*) ;
     pilote cette boucle pour les cultures fourragères fauchées, et traite les fauches avec sortie de bilans aux récoltes :
     - `int` *evalQuotidien*(`int` *jour*,`String` *bilan*) ;
       exécute les méthodes à la période jour ;
3. **Restitution du bilan en fin de simulation** (2.1.5)

– `void` *bilanCampagne*(`String` *bilan*) ;
édite le bilan à la récolte, sauf pour les cultures fauchées, et indique le rendement évalué par la méthode :
– `float` *rendementRecolte*(`float` *magrainc*,`float` *h2orecc*)
qui reprend les formules de calcul du bilan ;

4. **Initialisation de la culture suivante** (2.1.6)
– `void` *suivantCampagne*(`int` *campagne*) ;
exécute la méthode native qui transfère les données relatives à l'état final de la parcelle X et à l'état initial de la culture suivante Y ;

4.4 Consultation et mise à jour des paramètres et variables

Des méthodes sont disponibles pour consulter les valeurs *Stics* et effectuer des mises à jour.

Consultation

– `float` *valeurStics*(`String` *nom*)
rend la valeur de la variable *Stics* désignée.
– `float` *valeurStics*(`String` *nom*,`int` *indice*)
rend la valeur associée à l'indice pour une table *Stics*.

Mise à jour

– `void` *affecterStics*(`String` *nom*, `float` *valeur*)
affecte la valeur dans la variable *Stics* désignée.
– `void` *affecterStics*(`String` *nom*,`int` *indice*, `float` *valeur*)
affecte la valeur pour l'indice cité d'une table *Stics*.

Méthodes temporelles

– `int` *periodeSimul*(`int` *jour*)
rend le rang de la simulation pour le jour calendaire.
– `int` *serieStics*(`String` *nom*)
rend 1 si le nom désigne une série temporelle, 0 sinon.
– `int` *gregoire*(`int` *an*)
rend le nombre de jours de l'année.

Restitution d'informations textuelles

– `String` *choixVariete*() rend le nom du groupe variétal ;
– `String` *choixSol*() rend le nom du type de sol ;

5 Classe ParcelleStics

La classe *ParcelleStics* gère le processus de conduite des cultures sur l'unité spatiale la parcelle. Pour simuler les systèmes de cultures d'un territoire agricole, petite région (pays) à laquelle est associé un climat local, il suffit de créer des objets de cette classe. Chaque instance est initialisée avec deux arguments fournis au constructeur, un identifiant du territoire, et l'année de récolte de la campagne initale, soit

– *ParcelleStics*(`String` *pays*, `int` *anDebut*)

Le processus spatial peut alors être simulé en continu pour les années successives, avec des campagnes annuelles qui enchaînent éventuellement plusieurs unités de simulation, dont les intercultures. Il peut aussi être répété sur une ou plusieurs périodes pour évaluer de grands ensembles d'itinéraires techniques sur un territoire.

Arborescence des objets L'initialisation du territoire et la gestion des processus sont organisés dans une arborescence de racine $< pays >$. Des objets `CultureStics` sont d'abord créés dans des branches $< culture >$ pour produire les vecteurs des cultures standards à partir de l'existant STICS. Ces vecteurs sont ensuite copiés dans des branches $< parcelle >$ pour pouvoir initialiser les objets `CultureStics` qui vont permettre de simuler les processus sur chaque parcelle.

5.1 Objets et données

Plusieurs ensembles d'objets et variables des types de base sont définis pour gérer la dynamique des cultures et évaluer les itinéraires techniques pour les parcelles d'un territoire identifié par l'attribut

– `String` *pays* ;

5.1.1 Campagne culturale

L'état quotidien de la culture sur la parcelle est géré dans l'objet

– `CultureStics` *maCulture* ;

Les paramètres lus sont conservés dans des vecteurs homonymes des attributs de la classe `CultureStics`.[17]

– `float` *contexte*[] pour les paramètres généraux, pseudo-constantes K définies pour une période et la région ;

[17] Les vecteurs de `CultureStics` contiennent en sus les paramètres évalués dans l'étape d'initialisation.

– `float` *sol*[], paramètres du sol S de la parcelle ;
– `float` *plante*[], paramètres de la plante V pour la culture courante ;
– `float` *travail[]*, itinéraire technique W proposé ;
– `float` *meteo*[], paramètres climatiques M pour la campagne courante ;
– `float` *iniParcelle*[], état initial X pour la parcelle ;
Deux tables sont définies pour conserver l'état de la culture, soit
– `float` *etatCultureIni*[] après initialisation avant la première évaluation ;
– `float` *etatCultureJour*[] avant la simulation du jour
  – `int` *jourDebut*, début d'un calendrier d'opérations ;
Des attributs sont ajoutés pour le repérage chronologique, soit
– `int` *anCampagne*, année de récolte, initialisée avec *anDebut* ;
– `int` *finCampagne*, rang de l'ultime simulation d'une campagne ;
– `int` *rangCulture*, rang de la culture sur la parcelle ;
– `int` *annuel*, 1 si culture annuelle, 2 si bisannuelle ;

Culture précédente Les successions de cultures sont traitées par une méthode native qui gère la transmission des données entre les unités de simulation (2.1.6). Les résidus après récoltes, enfouis dans une opérations de travail du sol simulée pour la culture suivante, sont décrits par des paramètres de la culture précédente non transmis. Ces valeurs sont reprises ici dans des attributs afin de pouvoir les restituer dans les tables de paramètres techniques W de la culture en cours, soit :
– *qResSuite*, quantité en tonnes par hectare ;
– *qnResSuite*, teneur en azote minéral ;
– *csurnResSuite*, rapport C/N ;

5.1.2 Calendrier des opérations techniques

Les interventions culturales *Stics* sont décrites avec les paramètres techniques, dans des tables pour les opérations répétitives, avec des dates qui définissent les calendriers de campagnes. Une forme générique est adoptée pour représenter ces calendriers sous une forme simple afin de pouvoir les gérer dans des attributs, une valeur étant une liste structurée au moyen d'une grammaire simple par des séparateurs, soit :
– les deux-points (:) pour les valeurs des paramètres liés à l'opération ;
– le point-virgule (;) pour les opérations, quotidiennes ;
Ces listes sont des chaînes de caractères gérées ici dans des tables pour représenter des ensembles d'itinéraires techniques (5.4).

1. **Travail du sol et fertilisation organique**
   Les opérations de travail du sol sont caractérisées par une profondeur

de travail, paramètre qui définit une zone du sol dans laquelle sont incorporés les résidus organiques.
- `String` *cultiv*[] gère ces opérations culturales dans une liste
  <jour> :<profondeur> :<quantite> ;

2. **Semis**
   Le calendrier est réduit à une date.
   Les paramètres du semis sont le groupe variétal et la densité.
   La variété est identifiée par son nom auquel sont associés 17 paramètres variétaux acquis avec les paramètres de la plante et repris au moyen de la méthode *varieteUsm()* de la classe `CultureStics`.
   - `String` *semis*[] gére des stratégies de choix variétaux sous la forme
     <jour> :<variete> :<densiteSemis> ;
3. **Irrigation**
   Les paramètres *Stics* retenus sont la date de l'apport et la quantité.
   - `String` *irrig*[] gère les apports du jour désigné, soit pour un calendrier d'irrigation une liste d'éléments
     <jour> :<quantité> ;
4. **Fertilisation azotée**
   - Les paramètres sont la date, la quantité, ainsi qu'un type d'engrais.[18]
     `String` *fertil*[] gère les apports du jour désigné d'un type d'engrais azoté, soit pour un calendrier de fertilisation une liste d'éléments
     <jour> :<quantité><type> ;

5.2 Initialisation des cultures d'un territoire

Les vecteurs nécessaires pour simuler les cultures du territoire sont produits par application des méthodes de la classe `CultureStics` qui reconnaissent les fichiers WINSTICS. Ils sont ensuite répartis dans l'arborescence du pays, avec des paramètres définis pour le territoire, les parcelles et les systèmes de cultures.

La liste des variables STICS de sortie est définie pour le territoire.

5.2.1 Reprise de l'existant WINSTICS

L'initialisation démarre dans les branches *< culture >* pour les cultures standards avec l'objet *maCulture*.
Les processus sont pilotés dans le répertoire *culture* par la méthode

[18] Le paramètre technique *Engrais* code 8 types d'engrais azotés, décrits dans des tables avec les paramètres généraux *Stics*.

– `void` *iniWinStics*(`String` *culture*) ;

laquelle exécute le code C natif au moyen de deux méthodes :

1. **Acquisition des paramètres et fabrication des vecteurs**
   – `void` *vectoWinStics*(`String` *culture*) ;
   applique à l'objet *maCulture* les méthodes (4.2.1) :
   – *iniUsm()* et *iniMeteo()* pour l'acquisition des paramètres ;
   – *vectoCulture()* pour la recopie des vecteurs ;
   – *varieteUsm()*,*typeSolUsm()* et *metoUsm()* pour l'extraction des paramètres variétaux, types de sols et climats des stations ;
   L'existence du chemin d'accès au répertoire est vérifié par
   – `String` *cheminUsm*(`String` *culture*),
   méthode qui clôt l'exécution si ce répertoire n'existe pas ;
2. **Exécution de l'unité de simulation et édition du bilan**
   – `void` *evalWinStics*(`String` *culture*, `int` *campagne*) ;
   applique à l'objet *maCulture* les méthodes d'évaluation (4.3) :
   iniCulture(), *evalQuotidien()*, *bilanCampagne()* ;

On retrouve alors les résultats des applications procédurales, C et Fortran.

5.2.2 Paramètres du territoire

A la région pour une période sont associés l'ensemble des paramètres généraux, pseudo-constantes K, et les climats M des campagnes culturales, sous la forme de séries quotidiennes relevées dans des stations météorologiques locales $<$ *station* $>$. Les vecteurs associés, fabriqués par reconnaissance des fichiers WINSTICS dans les répertoires des cultures, sont transférés dans l'arborescence $<$ *pays* $>$ au moyen de méthodes :

– `void` *contexteStics*(`String` *culture*) ;
  recopie le fichier "`K_`" du répertoire "`culture`" dans "`contexte.K`" dans le répertoire "`pays`" ;
– `void` *climatStics*(`String` *culture*, `int` *an1*,`int` *an2*) ;
  dans le répertoire "`culture`" pour *an* depuis `an1` jusqu'à `an2` :
  – applique la méthode *meteoUsm(culture,"`stat.an`")* à l'objet *maCulture* pour créer les fichiers "`station`.*an*" ;
  – lit ces fichiers et copie dans le répertoire "`pays`" les séries :
    – "`meteo`$<$ *an* $-$ 1 $><$ *an* $>$`.M`" pour les campagnes bisannuelles ;
    – "`meteo`$<$ *an* $>$`.M`" pour toute campagne ;
    Les constantes climatiques et descriptifs des séries sont ajoutés.

5.2.3 Paramètres des parcelles

Les parcelles, gérées dans les branches $< sol >$, sont caractérisées par les paramètres du sol S, et un état initial X_{n-1} pour la campagne n, soit X_0 en début de période avant le démarrage des cultures (1.1.3). Les fichiers sont copiés entre répertoires depuis "`culture`" vers "`sol`".

Choix du sol

Le fichier WINSTICS de paramètres regroupe un ensemble de types de sols, identifiés par des noms et des rangs,[19] répartis dans des fichiers par les méthodes de reprise de l'existant (4.2.1). Une méthode appliquée avec différents arguments permet de choisir un sol-type, les paramètres étant copiés dans le fichier "`sol.S`" :

- `void` *solStics*(`String` *culture*,`String` *sol*) ;
  recopie le fichier "`S_`" :
- `void` *solStics*(`String` *culture*,`String` *sol*, `String` *nom*) ;
  recopie le fichier "`nom.s`" ;
- `void` *solStics*(`String` *culture*,`String` *sol*, `int` *numsol*) ;
  recopie le sol-type de rang *numsol* ;

Etat initial de la parcelle

L'initialisation WINSTICS fabrique le vecteur X_0 :

- `void` *iniParcelleStics*(`String` *culture*, `String` *sol*) ;
  copie le fichier "`X_`" dans le fichier "`iniParcelle.X`" ;

Les conditions initiales, lues dans le fichier de simulation "`travail.usm`", dépendent de la culture initiale choisie, avec des valeurs relatives aux états des horizons du sol (humidité, azote, densité).

Ces méthodes sont applicables pour l'ensemble des sols de la région.

5.2.4 Paramètres des cultures

Les cultures des rotations sur les parcelles sont décrites par les paramètres de la plante V. A ces cultures sont associés des itinéraires techniques W, avec le choix variétal et des calendriers d'opérations prédéfinis pour la campagne. Les fichiers sont copiés depuis "`culture`" vers "`sol`" :

- `void` *planteStics*(`String` *culture*, `String` *sol*) ;
  recopie le fichier "`V_`" dans le fichier "`culture.V`" ;
- `void` *travailStics*(`String` *culture*, `String` *sol*) ;
  recopie le fichier "`W_`" dans le fichier "`culture.w`" ;

[19] Paramètres *Numsol* dans S et *Ichsl* dans X.

Ces méthodes sont appliquées pour l'ensemble des cultures des rotations. Si V demeure invariant, les groupes variétaux étant contenus dans une table, W est un itinéraire technique par défaut pour le territoire, nécessaire pour démarrer le processus mais modifiable, lors de l'initialisation pour prendre en compte les résidus de la culture précédente (5.3.1), ou en cours de simulation.

5.2.5 Variables de sortie

Une liste commune de paramètres et variables est définie pour la restitution des valeurs sous forme vectorielle sur les parcelles du territoire, et copiée sous le nom "`Stics.sortie`" dans le répertoire "`pays`". La méthode

– `int` *listeVariables*() ;

applique à l'objet l'objet *maCulture* la méthode *sortieVariables()* (4.2.3) pour lire cette liste.

5.3 Chronologie des cultures

Les simulations peuvent désormais s'exécuter pour les cultures standards sur les parcelles types du territoire.

5.3.1 Simulation d'une campagne culturale

La campagne est identifiée par l'année de récolte, valeur affectée dans l'attribut *anCampagne* : les étapes de l'unité de simulation *Stics* sont reprises par application à l'objet *maCulture* des méthodes de la classe `CultureStics`.

1. **Acquisition des paramètres et données initiales**
   (a) `void` *paraParcelle*(`String` *sol*)
   lit les paramètres généraux K, les paramètres du sol S, l'état initial de la parcelle X ;
   (b) `void` *paraCampagne*(`String` *culture*, `String` *sol*)
   lit les paramètres de la plante V, l'itinéraire technique W, les données climatiques de campagne M ;
2. **Initialisation de la campagne**
   `void` *iniCampagne*(`String` *culture*,`String` *sol*)
   applique à l'objet *maCulture* la méthode *iniCulture()*,
   le journal étant restitué dans un fichier suffixé par '`ini`' ;
   Les valeurs des attributs *Qressuite*, *Qnressuite*, *Csurnressuite*, pour la culture précédente, sont restituées dans les paramètres techniques ;
   L'état initial de la culture Y_0 est recopié dans *etatCultureIni*[] ;

3. **Evaluations quotidiennes de l'état de la culture**
   `int` *evalCampagne*()
   enchaîne les évaluations sur la période de simulation par application à l'objet *maCulture* de la méthode *evalQuotidien()*, laquelle rend le code de retour des fonctions natives, une valeur non nulle traduisant une erreur ou demande de fin d'exécution.[20]
   L'état de la culture Y_{t-1} avant évaluation du jour $t = jourDebut > 0$ est copié dans *etatCultureJour*[] ;
4. **Clôture de la campagne**
   (a) **Rotation des cultures**
   `void` *cloreCampagne*(`String` *culture*,`String` *sol*)
   effectue les mises à jour de l'état de la parcelle X et de l'état initial de la culture suivante Y.
   Qressuite, *Qnressuite* et *Csurnressuite* sont mis à jour.
   (b) **Edition du bilan**
   `void` *bilanCampagne()* édite le bilan de l'unité de simulation ;

5.3.2 Suivi et déboguage d'une unité de simulation

Les méthodes précédentes permettent d'enchaîner les étapes de la simulation. Des méthodes sont proposées pour suivre la chronologie des évaluations quotidiennes, et accessoirement repartir sur une période antérieure.

1. `int` *evalEtatCulture*(`int` *jour*)
   effectue les évaluations du jour ;
   Si ($jour < 0$) l'état de la culture Y avant simulation est copié dans *etatCultureJour[]* et *jourDebut* reçoit la valeur *jour* ;
2. `int` *reevalEtatCulture*()
   restitue l'état antérieur *etatCultureJour[]* ;
   refait l'évaluation du jour *jourDebut* ;

5.3.3 Décisions culturales

Les opérations culturales sont programmées dans des calendriers de campagnes prédéfinis, et décrites par les valeurs de paramètres techniques, dont les dates d'interventions. Ces calendriers sont gérés sous la forme de listes textuelles dans les attributs associés à ces opérations (5.1.2). Des méthodes sont définies pour modifier ces calendriers.

[20] Le STOP Fortran est repris en C avec un message d'erreur et ce code de retour.

1. **Cultiver la parcelle**
   - `void` *nonCultiver()*
     neutralise les opérations de travail du sol ;
   - `void` *cultiver*(`String` *travailSol*,`int` *n*)
     reconnaît la chaîne <*jour*> :<*profondeur*>
     et met à jour les paramètres pour l'opération de rang n ;
   - `void` *cultiver*(`String` *travailSol*)
     applique la méthode précédente avec l'argument $n = 1$;

   Incorporation d'une fumure organique
   Les différents types d'amendements organiques sont traités par la méthode
   - `void` *cultiver*(`String` *travailSol*, `int` *n*, `int` *typeResidu*, `float` *qres*)
     qui met à jour les paramètres pour l'opération de rang n :

2. **Semer une culture standard**
   Stics acquiert les paramètres de la plante pour la culture standard et un ensemble d'au plus 12 groupes variétaux gérés dans des tables.
   - `void` *nonSemer()*
     neutralise l'opération de semis ;
   - `void` *semer*(`String` *semis*)
     reconnaît la chaîne <*nom_variete*> :<*jour*> :<*densite*> ;
     nom de la variété suivi des date et densité de semis,
     et met à jour les paramètres *Stics* :
     - *Variete*, numéro du groupe variétal, indice dans les tables ;
     - *Densitesem*, densité du semis en plantes par par m^2 ;

3. **Fertiliser (azote minéral)**
   - `void` *nonFertiliserN()*
     neutralise les opérations de fertilisation minérale ;
   - `void` *fertiliserN*(`int` *n*, `String` *apport*)
     reconnaît la chaîne <*jour*> :<*quantité*><*type*> ;
     et met à jour les paramètres *Stics* pour l'opération de rang n, soit :
     - *Julapn[n]*, date de l'apport d'azote minéral ;
     - *Dosen[n]*, quantité en kilogrammes par hectare ;

     Si <type> est présent, il fournit une nouvelle valeur du paramètre :
     - *Engrais*, type d'engrais azoté utilisé pour la culture.
   - `void` *fertiliserN*(`String` *apport*)
     applique la méthode précédente avec l'argument $n = 1$.

4. **Irriguer**
   - `void` *nonIrriguer()*
     neutralise les opérations d'irrigation ;
   - `void` *irriguer*(`String` *apport*)

reconnaît la chaîne <*jour*> :<*quantité*> ;[<*jour*> :<*quantité*>] ;
liste de couples de paramètres *Stics* décrivant l'opération d'irrigation du jour, soit, si n est le rang du couple :
- *Julapi[n]*, date de l'apport d'azote minéral ;
- *Dosei[n]*, quantité en millimètres ;

5. **Faucher des cultures fourragères**
   - `void` *nonFaucher()*
     neutralise les dates prévues des coupes fourragères ;
   - `void` *faucher*(`int` *n*, `String` *coupe*)
     reconnaît la chaîne <*jour*> :<*hauteur*> ;
     et met à jour les paramètres *Stics* pour l'opération de rang n, soit :
     - *Julfauche[n]*, date de la coupe ;
     - *Hauteurcoupe[n]*, hauteur de la coupe en mètres ;

   Pâturage
   La méthode est redéfinie avec deux arguments supplémentaires.
   - `void` *faucher*(`int` *n*, `String` *coupe*, `float` *laiFinal*, `float` *msFinal*)
     effectue en sus la mise à jour deux paramètres techniques, soit :
     - *Lairesiduel[n]* = *laiFinal*, indice foliaire du jour après pâturage ;
     - *Msresiduel[n]* = *msFinal* , matière sèche restante ;

5.3.4 Gestion du temps calendaire et chronologie des campagnes

Les cultures sont simulées sur une période prédéfinie, reprise avec les paramètres techniques W, l'installation de la culture et la récolte se situant dans cet intervalle, avec éventuellement des plages vides en début et fin avant la première opération de travail du sol et après l'ultime opération culturale. La période de simulation est désignée par les rangs des jours calendaires, de 1 à 366 pour les cultures annuelles, 731 pour les bisannuelles. Pour les cultures pérennes, les campagnes étant annuelles, la simulation des années postérieures à l'installation sont faites après réinitialisation à partir de valeurs choisies de paramètres.

Les intercultures, avec des opérations de travail du sol pour l'enfouissement des résidus organiques, sont simulées au moyen d'une culture fictive, annuelle ou bisannuelle.

Contrôle des périodes de simulation

Plusieurs méthodes sont utilisables pour gérer ces cultures et périodes :
- `int` *debutSimul*() ;
  rend la date de début de *maCulture* ;

– `int` *finSimul*() ;
rend la date de fin de *maCulture* ;
– `int` *iniSimul*(`int` *debut*,`int` *fin*) ;
applique à l'objet *maCulture* la méthode *affecterStics()* pour mettre à jour les paramètres de contrôle des périodes ;
copie les valeurs dans les champs *jourFinal* et *anFinal* de *maCulture* ;

5.4 Evaluations d'itinéraires techniques

Le logiciel *Stics* évalue un itinéraire technique statique pour une campagne culturale ininterrompue. Des méthodes sont proposées pour pouvoir faire varier cet itinéraire pour la culture standard d'une campagne, en relation avec des modèles économiques : production de jeux de données pour évaluer des fonctions de production, recherche d'itinéraires techniques optimaux en fonction d'objectifs.

5.4.1 Gestion temporelle des états de la culture

La classe `EvalStics` gère l'exécution des fonctions vectorielles. Une culture étant initialisée, les méthodes précédentes mobilisent ces fonctions pour évaluer une suite d'états quotidiens Y, les autres vecteurs demeurant invariants. Ces états peuvent être conservés dans la classe pour permettre de redémarrer une simulation à une date antérieure avec d'autres décisions, conditionnelles en fonction de l'état de la culture, ou impératives pour des listes de stratégies produites par un autre modèle.

La restitution de ces états quotidiens est prise en charge par les méthodes de simulation de la campagne (5.3.1) :

1. *iniCampagne()* restitue dans *etatCultureIni[]* l'état initial de la culture avant la première évaluation quotidienne ;
2. *evalCampagne(jour)* appelée avec l'argument $jour > 0$
copie dans *etatCultureJour[]* l'état du jour avant évaluation ;
3. *evalEtatCulture()* appelée avec l'argument $jour < 0$
copie dans *etatCultureJour[]* l'état du jour avant évaluation ;

5.4.2 Production de jeux de données

Stics est utilisable pour simuler des stratégies culturales représentées dans des calendriers d'opérations, et récupérer les valeurs de variables observées à la récolte. Si certaines opérations sont uniques pendant une campagne, avec toutefois des variantes comme le choix variétal pour le semis, les

apports de facteurs s'effectuent en plusieurs opérations, ce qui peut aboutir à de grands nombres de plans possibles, et des temps de calcul conséquents.

Des méthodes sont définies pour évaluer ces plans à partir d'un état initial de la culture, date de début de la simulation ou d'une période, par exemple la saison d'irrigation. La campagne est initialisée pour la culture standard avec un itinéraire de référence puis évaluée (5.3.1), avec restitution des deux états de la culture :

- *etatCultureIni[]* avant la premier jour de la simulation ;
- *etatCultureJour[]* au début d'une saison de travaux ;

L'efficacité de ces plans en termes de résultats économiques et/ou environnementaux est liée aux climats des campagnes : cette dimension temporelle est prise en compte dans ces méthodes pour produire des jeux de données sur longue période en simulant l'ensemble des plans pour les campagnes culturales des années successives.

Variables restituées Les listes de variables à éditer sont à initialiser dans l'attribut *sortieStics[]* de l'objet *maCulture* par appel de la méthode *sortieVariables()* de la classe `CultureStics`.

Cultures irriguées

Le modèle *MoGIRE*[21] utilise *Stics* pour générer des données afin d'estimer des fonctions de production, soit 1000 calendriers pour 10 années climatiques pour les cultures irriguées des petites régions de bassins fluviaux.

1. **Acquisition des calendriers**
   Les plans d'irrigation sont gérés dans l'attribut *irrig[]* sous la forme de chaînes {<*jour*> :<*quantite*> ;}.
   Cette forme est lue dans le fichier "<`culture`>`.eau`" par la méthode
   – `int` *lireIrrig*(`String` *culture*,`String` *sol*) ;
   La forme numérique utilisée dans *MoGIRE*, avec n dates d'apports en entête puis n valeurs ≥ 0 par lignes pour les plans, est lue dans le fichier "<`culture`>`.neste`" par la méthode
   – `int` *lireNeste*(`String` *culture*,`String` *sol*) ;
2. **Evaluation des plans d'irrigation**
   La méthode *evalCampagne(jour)* initialise l'état de la culture *etatCultureJour[]* au début de la période d'irrigation : la boucle sur les calendriers pour l'année de récolte *anCampagne* s'exécute pour cette période jusqu'à la récolte par application de la méthode

[21] **Mo**dèle de **G**estion **I**ntégrée de la **R**essource en **E**au, INRA, Toulouse.

– `void` *evalIrrig*(`String` *culture*,`String` *sol*) ;
laquelle évalue chaque calendrier *iplan* avec la méthode
– `String` *evalPlanIrrig*(`CultureStics` *maCulture*, `int` *iplan*) ;
Les résultats sont édités dans un fichier qui reprend le nom de la culture et l'année, suffixé par " `.irrig`", chaque ligne contenant :
– le numéro d'ordre du plan ;
– le total des apports ;
– la date de récolte ;
– les valeurs des variables de la liste *sortieStics*[] de l'objet `maCulture` ;
– le rendement évalué par la méthode :
– `float` *rendementPlan*(`CultureStics` *maCulture*, `int` *recolte*) ;
qui applique la méthode *rendementRecolte()* à *maCulture*.

Irrigation optimale La méthode *reevalEtatCulture()* (5.3.1) est utilisable pour estimer une irrigation optimale, l'évaluation du jour étant refaite avec un apport d'eau en cas de stress hydrique.

Fertilisations azotées

Des méthodes sont définies de la même façon pour la fertilisation azotée, avec des apports qui peuvent être répartis aux différents stades de développement.

1. **Acquisition des calendriers**
Gérés dans l'attribut *fertil[]*, les calendriers sont lus dans le répertoire de la parcelle au moyen de la méthode
– `int` *lireFertil*(`String` *culture*,`String` *sol*) ;
qui reconnaît ici la forme *<jour> :<quantite>[:<type>]* ;
le type, optionnel, étant par défaut celui de la simulation de référence.
2. **Evaluation des plans de fertilisation**
Les méthodes sont analogues aux précédentes pour l'irrigation, soit :
– `void` *evalFertil*(`String` *culture*,`String` *sol*, `String` *plan*) ;
– `void` *evalFertil*(`String` *culture*,`String` *sol*, `String` *plan*,
`int` *anDebut*, `int` *anFin*) ;
qui utilisent pour chaque plan :
– `String` *evalPlanFertil*(`CultureStics` *maCulture*, `int` *iplan*) ;
Les sorties sont également semblables, aux unités près.

Choix variétaux

L'objectif est de pouvoir simuler les choix des différentes variétés décrites avec les paramètres de la plante pour une culture standard, avec des périodes

de semis et des densités variables. Des méthodes sont définies pour évaluer ces variétés avec différentes dates et densités de semis.

1. **Acquisition des plans**
   Gérés dans l'attribut *semis[]*, les plans sont lus dans le répertoire de la parcelle au moyen de la méthode
   – `int` *lireSemis*(`String` *culture*,`String` *sol*) ;
   qui reconnaît ici la forme <*nom_variete*> :<*jour*> :<*densite*> ;
2. **Evaluation des stratégies variétales**
   Les méthodes sont définies avec les mêmes arguments, soit :
   – `void` *evalSemis*(`String` *culture*,`String` *sol*, `String` *plan*) ;
   – `void` *evalSemis*(`String` *culture*,`String` *sol*, `String` *plan*, `int` *anDebut*, `int` *anFin*) ;
   qui utilisent pour chaque plan :
   – `String` *evalPlanSemis*(`CultureStics` *maCulture*, `int` *iplan*) ;
   Les choix variétaux et dates et densités de semis sont édités en sortie.

Irrigation et choix variétal
Fertilisation azotée et choix variétal

Les stratégies d'irrigation comme celles de fertilisation azotée, peuvent être simulées pour différentes variétés et précédents culturaux.

1. **Acquisition des plans et calendriers**
   Plans et calendriers sont lus avec les méthodes
   – `int` *lireSemisIrrig*(`String` *culture*, `String` *sol*) ;
   – `int` *lireSemisFertil*(`String` *culture*, `String` *sol*) ;
   qui reconnaissent les formes
   <*nom_variete*> :<*jour*> :<*densite*> et <*jour*> :<*quantite*> ;
2. **Evaluation**
   – `void` *evalSemisIrrig*(`String` *culture*,`String` *sol*, `String` *plan*) ;
   – `void` *evalSemisIrrig*(`String` *culture*,`String` *sol*, `String` *plan*, `int` *anDebut*, `int` *anFin*) ;
   pilotent les appels des méthodes *evalPlanSemis*() et *evalPlanIrrig*() ;
   – `void` *evalSemisFertil*(`String` *culture*, `String` *sol*, `String` *plan*) ;
   – `void` *evalSemisFertil*(`String` *culture*, `String` *sol*, `String` *plan*, `int` *anDebut*, `int` *anFin*) ;
   pilotent les appels des méthodes *evalPlanSemis*() et *evalPlanFertil*() ;

Les choix variétaux, dates, densités de semis sont à ajouter dans la liste des variables *Stics* de l'objet *maCulture*.

6 Classe ModelStics

La classe `ModelStics` offre une interface graphique de restitution des algorithmes avec les formalismes du modèle numérique, reconnus dans le code informatique du logiciel. Dérivée de la classe `JFrame` du paquetage standard `javax.swing`, elle gère une fenêtre graphique et des évènements, actions de l'utilisateur pour selectionner des composants.

Les objets de cette classe sont créés avec une commande d'appel d'un éditeur en argument, soit :

– *ModelStics*(`string` *monEditeur*) ;

6.1 Données et objets graphiques

Les données, dictionnaires et graphes, sont les structures issues de *CStics*, auxquelles s'ajoutent les objets graphiques utilisés pour leur restitution.

6.1.1 Formalismes du modèle

Variables, fonctions et graphes sont gérés dans des tables.

1. **Variables informatiques**
   Les caractéristiques des variables *Stics*, anciennes variables communes Fortran, sont lues dans `Stics.variable`, et notées dans les attributs :
   – `String` *nomV*[], noms *Stics*, avec première lettre majuscule ;
   – `int` *typeV*[], entier, réel, logique, textuel ;
   – `int` *dimV*[], dimension, variable simple, vecteur, matrice ;
   – `int` *sup1V*[], taille vecteur ou nombre de lignes matrice ;
   – `int` *sup2V*[], nombre de colonnes matrice ;
   – `String` *vectoV*[], signature des vecteurs qui gèrent la variable ;
   – `int` *sommetV*[], numérotation sommets ;
   – `int` *nvStics*, nombre de variables ;
2. **Fonctions informatiques**
   Les noms et rôles des fonctions *Stics*, lus dans `Stics.fonction`,
   les noms des fonctions natives, lus dans `Stics.native`,
   les arbres des appels, reconnus dans `Stics.appel`,
   sont gérés dans les attributs :
   – `String` *nomF*[], noms des fonctions ;
   – `int` *dynamiqueF*[], 1 si évaluation quotidienne, 0 sinon ;
   – `int` *suivantF*[], chaînage alphabétique ;
   – `int` *nfStics*, nombre de fonctions *Stics* ;
   – `int` *nfNative*, nombre de fonctions associées aux méthodes natives ;

– `int` *nNativeAppel*, nombre d'appels *Stics* depuis ces fonctions ;
– `int` *appelF*[][], chaînes d'appels, jusqu'à la feuille terminale ;
– `int` *appelantF*[][], chaînes des appelants depuis un nœud ;

3. **Graphe des relations entre variables**
   Le processus d'évaluation des variables dans une fonction est représenté dans un graphe : $x \rightarrow y$ signifie que x est utilisé pour évaluer y. Fonctions et variables étant identifiés par des clés numériques, cet ensemble de graphes est lu dans `Stics.eval`, et restitué sous la forme :
   – `int` *arbreV*[][3], liste des triplets *(fonction,origine, extrémité)* ;
   Les arcs sont regroupés par fonctions, les ensembles issus de cette projection définissent un ou plusieurs arbres d'évaluation. Des tables sont produites pour décrire et explorer cette forêt :
   (a) pour chaque variable,
      – `int` *brancheV*[][], liste des fonctions qui les évaluent ;
   (b) pour chaque fonction,
      – `int` *brancheF*[][], liste des variables évaluées ;
      – `int` *nevalF*[][], nombre de variables évaluées ;
      – `int` *grapheF*[], rang du premier arc dans *arbreV*[] ;

6.1.2 Objets graphiques

La première fonction de l'interface graphique est l'accès à l'ensembles des noms symboliques décrits dans les formalismes *Stics*, utilisés dans le logiciel comme variables informatiques, auxquels s'ajoutent les noms des fonctions informatiques, unités de programmes Fortran appelées depuis les modules *Stics*, eux-mêmes appelés dans les fonctions natives. L'affichage de ces longues listes utilise un composant graphique classique, la boîte de liste dite combinée, classe `JComboBox` du langage *Java*, les listes étant affichées et déroulées à la demande dans ces boîtes, dotées d'un ascenseur si nécessaire.

1. **Affichage et sélection des variables**
   – `JComboBox` *variablesStics*[], table de 15 boîtes pour la version 6, listes d'au plus 120 noms de paramètres ou variables *Stics* ;
   – `String` *listeVariables*[][], listes associées aux boîtes ;
   – `int` *indexVariables*[], rangs des têtes de listes des boîtes ;
   – `int` *maVariable*, dernière variable sélectionnée ;
2. **Affichage et sélection des fonctions**
   – `JComboBox` *fonctionEval*, fonctions *Stics* d'évaluation temporelles ;
   – `String` *listeFonctionsEval*[], noms de ces fonctions ;
   – `JComboBox` *fonctionNative*, appels depuis les méthodes natives ;

– `String` *listeFonctionsNative*[], chemins de ces appels ;
– `JComboBox` *fonctionAlgo*, fonctions qui évaluent une variable ;
– `int` *rangFonctionAlgo*[], rangs de ces fonctions ;

3. **Exploration des formalismes et suivi d'exécution**
Deux boutons radio permettent de basculer entre modes *explo* et *suivi* :
– `JRadioButton` *explo*, examen des variables et fonctions d'évaluation ;
– `JRadioButton` *suivi*, consultation des valeurs en cours d'exécution ;
La sélection de variables dans les listes des boîtes fournit des informations restituées dans une zone de texte :
– `JTextArea` *journal* ;

(a) **Examen des équations**
Le mode *explo* restitue les processus d'évaluations des variables sélectionnées par citation des variables des équations dans les fonctions. La selection d'une fonction affiche l'algorithme sous la forme d'un pseudo-code C sous l'éditeur GNU *Emacs*, ce qui requiert un accès à l'environnement d'exécution du système hôte au moyen de l'objet :
– `Runtime` *systemeHote* ;
Le chemin d'accès au répertoire du pseudo-code est géré dans :
– `String` *repAlgo* ;
La commande d'appel de l'éditeur est notée dans :
– `String` *ediAlgo* ;

(b) **Suivi des évaluations**
Le mode *suivi* permet de suivre l'exécution après initialisation et dans la boucle quotidienne au moyen de l'objet :
– `CultureStics` *maCulture* ;
La recherche des vecteurs est déclenchée avec le bouton :
– `JButton` *usm* ;
Le chemin d'accès au répertoire de la parcelle est noté dans :
– `String` *cheminUsm* ;
Les choix sont alors effectués au moyen de boîtes de listes :
– `JComboBox` *choixUsm*, pour la plante ;
– `JComboBox` *choixAnRecolte*, pour l'année de récolte ;
Le contrôle temporel de l'exécution est piloté depuis la boîte :
– `JComboBox` *piloteUsm* ;
Les états de la simulation sont suivis dans :
– `int` *etapeSimul*, rangs des étapes ;[22]

[22] L'initialisation des boîtes de listes génère les mêmes évènements que les sélections dans l'interface : *etapeSimul* est utilisée comme variable de contrôle pour prévenir les références vers les objets dynamiques créés à partir de l'interface ($etapeSimul < 0$).

– `int` *rangSimul*, période simulée ;
– `JTextField` *etatUsm*, champ d'affichage de la période ;

Les noms de répertoires et fichiers sont acquis au moyen de boîtes de dialogues, objets graphiques temporaires de la classe `JOptionPane`.

6.2 Reconnaissance des formalismes et initialisation

Les formalismes reconnus par *CStics* sont représentés dans des structures restituées dans des fichiers : des méthodes sont appliquées par le constructeur de classe pour leur acquisition et l'initialisation des composants graphiques.

6.2.1 Paramètres et variables du modèle

Les paramètres et variables sont désignés dans les listes par leurs noms propres, première lettre majuscule, suivi du ou des vecteurs d'appartenance :
– `void` *iniVariables()* ;
initialise les tables descriptives par lecture dans `Stics.variable` ;
– `void` *boitesVariables()* ;
fabrique les boîtes de listes de variables ;
initialise le composant *journal*, zone d'édition associée ;

6.2.2 Fonctions et variables évaluées

Les fonctions sont désignées par leurs noms d'unité de programme Fortran, ou de celui de modules précédés de la chaine `"Stics"`. Cet environnement est initialisé par les méthodes suivantes, appelées dans cet ordre :

1. `void` *iniFonctions()* ;
initialise les tables descriptives et génère les arbres des appels après lecture de `Stics.fonction` et`Stics.appel`, produits par *Cstics*, et `Stics.native`, créé ;
2. `void` *iniGraphe()* ;
produit les représentations des ensembles d'arbres d'évaluation à partir des structures lues dans `Stics.eval` ;
3. `void` *cheminsFonctions()* ;
produit les chemins des appels et les listes de fonctions ;
utilise la méthode récursive :
– `int` *moduleStics*(`int` *ilex*,`int` *ici*, `int` *pileAppelant[]*) ;
construit les chemins pour la fonction appelée ;

4. `void` *boiteFonctions()* ;
   fabrique les boîtes de listes de fonctions ;
   initialise l'environnement d'exécution ;

6.3 Exploration des formalismes

Cette interface permet de retrouver l'ensemble des formalismes dans le logiciel. Les actions sur les composants graphiques sont des sélections par de simples clics, lesquels génèrent des évènements gérés au moyen d'un écouteur, objet de la classe abstraite `ActionListener` qui met œuvre les méthodes pour les traiter.

6.3.1 Identification des composants actifs

Une classe abstraite ne fait que déclarer ses méthodes :

– `void` *actionPerformed*(`ActionEvent` *clic*) ;

est définie avec un objet de la classe `ActionEvent` en argument, laquelle offre une méthode, *getSource()*, qui rend une référence sur l'objet qui a déclenché l'évènement : c'est le moyen utilisé pour localiser les composants graphiques actifs dans des séquences d'instructions conditionnelles, et distribuer les tâches vers les méthodes spécialisées.

6.3.2 Sélection d'un variable informatique

Les variables sont classées dans l'ordre alphabétique, chaque boîte étant initialement repérée par le nom de la première variable de la liste associée, la sélection d'une boîte déroulant la liste associée. Ces variables sont identifiées par leurs noms, suffixés pour les tables par les tailles entre crochets, suivis de l'identification des vecteurs d'appartenance pour distinguer paramètres et variables du modèle.

La sélection d'un nom déclenche la copie dans *journal* des chemins d'appels des fonctions qui évaluent la variable ou initialisent le paramètre, sous la forme de chaînes jusqu'aux fonctions natives : pour chaque chemin figure l'ensemble des noms des variables informatiques utilisées dans la fonction d'évaluation, ces noms étant suffixés par les signatures des vecteurs. La mention `"invariant"` indique que le nom sélectionné est un paramètre lu, ou initialisé avec un nombre dans le code.

Les fonctions d'évaluation sont reprises dans la liste déroulante de la boîte *fonctionAlgo* qui apparaît alors avec le titre `"Algorithme ?"`.

6.3.3 Sélection d'une fonction

Les fonctions sont identifiées par leurs noms suivis des parenthèses (). Les fonctions *Stics* sont réparties après acquisition des formalismes dans deux boîtes de listes, *fonctionEval* pour les fonctions temporelles qui effectuent les évaluations quotidiennes, *fonctionNative* pour les fonctions associées aux méthodes natives, avec l'initialisation de la simulation puis l'enchaînement des modules pour une période.

La sélection d'une boîte déroule la liste associée : la sélection d'un nom visualise l'algorithme de la fonction sous la forme d'un pseudo-code C qui reprend les noms *Stics*, avec la première lettre majuscule pour les anciennes variables informatiques communes, minuscule pour les variables locales.

La restitution des algorithmes s'effectue à l'extérieur de l'application, avec la commande d'appel de l'éditeur *ediAlgo* et le répertoire *ediAlgo* contenant les sources, les valeurs étant reçues en argument du constructeur.

6.4 Suivi d'exécution

Le suivi va s'effectuer pour une unité de simulation dans le contexte d'utilisation des objets de la classe `SticsParcelle`. La sélection du mode `suivi` affiche le bouton *usm*. Un clic sur ce bouton ouvre une boîte de dialogue pour l'acquisition d'un chemin d'accès au répertoire d'une parcelle d'un pays. Si ce répertoire existe et si les fichiers contenant les vecteurs de paramètres sont présents, trois boîtes de listes apparaissent :

- *choixUsm* énumère les cultures décrites ;
- *anRecolte* restitue la liste des années climatiques ;
- *piloteUsm* permet de lancer et suivre l'exécution ;

etatUsm, champ contenant la période de simulation, s'affiche, vide ;
etapeSimul devient non négatif pour signifier que l'interface est active ;

6.4.1 Pilotage des simulations

Les boîtes *choixUsm* et *anRecolte* permettent de choisir la culture, puis la campagne identifiée par l'année de récolte : la boîte *piloteUsm* apparaît. Les options disponibles sont les suivantes :

0. `Campagne` ne déclenche rien : sa fonction est de signifier que la simulation peut démarrer par sélection des options qui suivent ;
1. `Acquisition` démarre la simulation avec la lecture des paramètres ; *etatUsm* affiche alors -1, la période n'étant pas définie ;

2. Initialisation exécute l'étape d'initialisation de la culture ;
   etatUsm affiche le temps 0, état initial $t =$*rangSimul* ;
3. $\underline{t+1}$, $\underline{t+5}$, $\underline{t+10}$, $\underline{t+50}$, $\underline{t+100}$, simulent le nombre de périodes ;
   etatUsm affiche le temps $t =$*rangSimul* ;
4. `Fin` termine la simulation ;

L'état de la culture pour la période courante est restitué par sélection dans les boîtes de listes des variables : les valeurs sont copiées dans *journal*.

6.4.2 Mise en œuvre

Les paramètres des cultures pour les campagnes sont lus dans les répertoires des pays, avec les climats, puis dans ceux des parcelles-types pour ces pays. L'interface utilise les boîtes de saisie interactives pour l'acquisition des chemins et de message pour signaler les problèmes, ceci au moyen de la méthode :

– `int` *chercherUsm(*`String` *chemin)* ;

La culture et la campagne étant choisies, la sélection d'une option de pilotage est traitée par la méthode

– `void` *suiviUsm()* ;

Les traitements effectués vont dépendre de l'état de la simulation :

1. Si (*étapeUsm* $= 0$), ou si acquisition est sélectionnée, la lecture des paramètres est pilotée par la méthode :
   – `void` *paraUsm()* ;
   acquiert les vecteurs pour l'objet *maCulture* ;
   étapeUsm reçoit la valeur 1 ;
2. Initialisation déclenche l'exécution de l'étape d'initialisation, même si des évaluations quotidiennes sont en cours ou une initialisation déjà faite (*étapeUsm* ≥ 2), par application de la méthode :
   – `void` *iniCulture()* ;
   initialise l'état de la culture à la période initiale *rangSimul* $= 0$;
   étapeUsm reçoit la valeur 2 ;

Table des matières