Finalisation d'un modèle de bilan de gaz à effet de serre : FarmSim

Ecrah Hoba Ulrich Eza

▶ **To cite this version:**

Ecrah Hoba Ulrich Eza. Finalisation d'un modèle de bilan de gaz à effet de serre : FarmSim. Milieux et Changements globaux. 2014. hal-02799811

HAL Id: hal-02799811
https://hal.inrae.fr/hal-02799811

Submitted on 5 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Institut **S**upérieur d'**I**nformatique de
Modélisation et de leurs **A**pplications

Campus des Cézeaux
24 avenue des Landais
BP 10125
63173 Aubière cedex

UREP - INRA

Site de Crouël
5 chemin de Beaulieu
63039 Clermont-Ferrand cedex 2

Rapport d'ingénieur
stage de 3e année
Filière génie logiciel & systèmes informatiques

Finalisation d'un modèle de bilan de gaz à effet de serre : FarmSim

Présenté par : **Ulrich Eza**

Responsable ISIMA : **M. Claude Mazel**
Responsable INRA : **M. Raphaël Martin**

1er Avril au 30 septembre 2014
Durée du stage : 6 mois

Remerciements

Je ne saurais commencer ce rapport sans tout d'abord remercier mon tuteur de stage Raphaël Martin pour la disponibilité dont il a fait preuve. Malgré le manque d'informations en amont de mon stage, il a toujours su m'aiguiller dans mon travail. Il n'a pas hésiter lorsque j'avais des questions à prendre de son temps, malgré son agenda chargé, pour m'éclairer davantage.

Je tiens également à remercier M Claude Mazel qui s'est assuré du bon déroulement de mon stage et qui a pu participer lors de sa visite de stage à des réflexions autour des objectifs à atteindre.

Pour finir je voudrais remercier l'équipe de l'UREP pour son accueil chaleureux et pour l'ambiance conviviale qui y régnait. Un remerciement tout particulier va à l'encontre des membres de mon bureau, d'une part pour mes questions auxquelles ils ont su répondre et d'autre part pour tous les conseils concernant ma vie professionnelle qu'ils ont su me donner. Je n'oublie pas les matchs de football ou de ping-pong ou encore les tables rondes après la pause déjeuner qui ont apporté plus de gaieté à mon stage.

Résumé

La réduction des émissions de **gaz à effet de serre** (GES) constitue un enjeu majeur dans la recherche agroenvironnementale. C'est dans ce but que l'INRA a développé un modèle de simulation, **FarmSim**, qui permet de simuler une exploitation agricole en agrégeant plusieurs autres modèles dont **PaSim** pour les prairies, et **CERES** pour les cultures. Les bilans en sortie de ces différents modèles sont ensuite agrégés afin d'avoir un bilan de GES au niveau de l'exploitation agricole.

Cependant, du fait du traitement indépendant des composants de la ferme (prairies, cultures) par ces modèles, il était impossible de simuler des **rotations** prairies/cultures.

J'ai ainsi dans un premier temps modifié FarmSim en vue de le rendre stable puis ai implémenté le mécanisme de rotations. Ceci fait, j'ai développé une plateforme afin de lancer des simulations spatialisées avec ce modèle.

Mots-clés : gaz à effet de serre, FarmSim, PaSim, CERES, rotations

Abstract

Reduction of **greenhouse gases** (GHG) **emissions** is at stake in the food-environmental research field. In order to act complementary, INRA scientists developed a simulation model, **FarmSim** which is able to simulate a complete farm by making use of other models: **PaSim**, for grasslands and **CERES** for crops. Outputs of all these nested models are then merged in order to produce the final result.

However, because each of the farm components (grasslands, crops) was only processed by its corresponding model, it was impossible to simulate grasslands/croplands **rotations**.

In a first time, I upgraded FarmSim to make it stable, then I implemented plots rotations. Once done, I developed a platform in order to launch large-scale simulations with this model.

Keywords: greenhouse gases emissions, FarmSim, PaSim, CERES, rotations

Table des matières

Table des figures

Liste des tableaux

Liste des Algorithmes

Glossaire

API	Ensemble de fonctionnalités qui permettent à un logiciel donné d'utiliser les services d'un autre
DTD	**D**omain **T**ype **D**efinition
GES	**G**az à **E**ffet de **S**erre
IDE	**I**ntegrated **D**evelopment **E**nvironment
INRA	**I**nstitut **N**ational de **R**echerche **A**gronomique
Indice de surface foliaire	Rapport de la superficie du feuillage vert à la superficie du sol.
IPCC	**I**ntergovernmental **P**anel on **C**limate **C**hange
PaSim	**Pas**ture **Sim**lation model
SMTP	**S**imple **M**ail **T**ransfert **P**rotocol.
Thread	Processus léger s'éxecutant sur un processeur
UREP	**U**nité de **R**echerche sur l'**E**cosystème **P**rairiral
Widget	Composant graphique de la bibliothèque Qt

Introduction

L'un des principaux enjeux des recherches menées en agroenvironnement est la réduction des émissions des gaz à effet de serre (GES) liées à l'agriculture. Cette dernière est à l'origine d'environ **20%** des émissions de GES en France du fait de la production de **trois** principaux composés : le dioxyde de carbone (CO_2), le méthane (CH_4) et le protoxyde d'azote (N_2O). Pour mener à bien ses recherches dans ce domaine, l'INRA a mis en place un modèle de simulation : **FarmSim**, qui permet d'effectuer un bilan des GES à l'échelle d'une exploitation agricole.

Pour assurer son fonctionnement, FarmSim réalise le couplage de plusieurs modèles dont **PaSim** pour les prairies, et **CERES** pour les cultures. Les bilans en sortie de ces différents modèles sont ensuite agrégés afin d'avoir un bilan au niveau de l'exploitation agricole.

Cependant, cette méthode présente un inconvénient du fait du traitement indépendant des composants de la ferme. En effet, dans l'état actuel du modèle, il est impossible de changer la pratique d'une parcelle d'une année à l'autre ; pratique pourtant courante dans l'agriculture.

C'est dans ce cadre que s'inscrit mon stage, dont l'objectif principal sera de finaliser FarmSim et ainsi de mettre en place les pratiques de rotations entre parcelles.

Afin de présenter le travail réalisé, ce rapport sera constitué de **trois** parties. Dans la première partie, il s'agira de présenter le contexte d'étude. La seconde partie présentera les corrections ainsi que les nouvelles fonctionnalités apportées au modèle permettant de satisfaire la problématique du stage. Enfin, je détaillerais les résultats obtenus suite aux simulations que j'aurais lancées à l'échelle de l'Europe.

1
Contexte d'étude

1.1 Présentation de l'institut

1.1.1 Vocation et implantation

L'**I**nstitut **N**ational de **R**echerche **A**gronomique (INRA) est un organisme public Français de recherche fondé en **1946**. Cet institut produit des connaissances scientifiques et accompagne l'innovation dans les domaines de l'alimentation, de l'agriculture et de l'environnement. C'est le premier institut de recherche agronomique en Europe et le deuxième mondial.

Ses activités s'organisent au sein de **treize** départements et sont repartis dans toute la France sur **21** centres.

1.1.2 Centre d'accueil

Mon stage s'est déroulé au centre de Clermont-Ferrand au sein de l'**U**nité de **R**echerche sur l'**É**cosystème **P**rairial (UREP) qui est rattachée au département "Écologie des forêts, des prairies et des milieux aquatiques".

L'UREP possède une expertise internationale dans le domaine de l'écologie prairiale et plus particulièrement sur l'impact du changement climatique, les bilans de gaz à effet de serre, la séquestration de carbone, les cycles carbone et azote [1]. L'unité a développé une démarche originale en se concentrant sur les interactions plantes-sol (microorganismes) et herbe-animal, et en considérant explicitement les effets des pratiques de gestion sur la dynamique prairiale.

1.2 Contexte scientifique

L'un des principaux enjeux des recherches menées en agroenvironnement est la réduction des émissions de ces gaz liées à l'agriculture. Cette dernière est à l'origine d'environ **20%** des émissions de GES en France du fait de la production de **trois** principaux composés :

1. Source : https://www1.clermont.inra.fr/urep/presentation/index.htm, consulté le 14/04/2014

– le dioxyde de carbone (CO_2) qui résulte de la combustion des énergies fossiles ainsi que du changement d'utilisation des sols
– le méthane (CH_4) issu des déjections animales ou encore de l'élevage des ruminants
– le protoxyde d'azote (N_2O) produit par l'épandage d'engrais azotés sur les sols

L'UREP étudie plus particulièrement la contribution des prairies, ces dernières occupant un quart du territoire européen, au flux de GES.
En plus des moyens expérimentaux, les chercheurs de l'INRA dispose d'un panel de modèles informatiques leur permettant de simuler ces émissions. L'un d'entre eux modélise une exploitation agricole et constitue le cœur de mon stage : **FarmSim**.

1.3 Présentation de FarmSim

FarmSim (figure 1.1) est un modèle d'exploitation agricole qui a été développé au sein de l'INRA dans le cadre du projet GREENGRASS (2002-2004). Depuis, il a subi plusieurs modifications comme c'est le cas, par exemple, du portage du modèle, du Visual Basic au Java (en 2007).
Il permet de modéliser une ferme dans son intégralité ainsi que ses flux de GES. Pour cela, il se base sur **deux** autres modèles : **PaSim** et **CERES**, ainsi que sur la méthodologie **IPCC.**

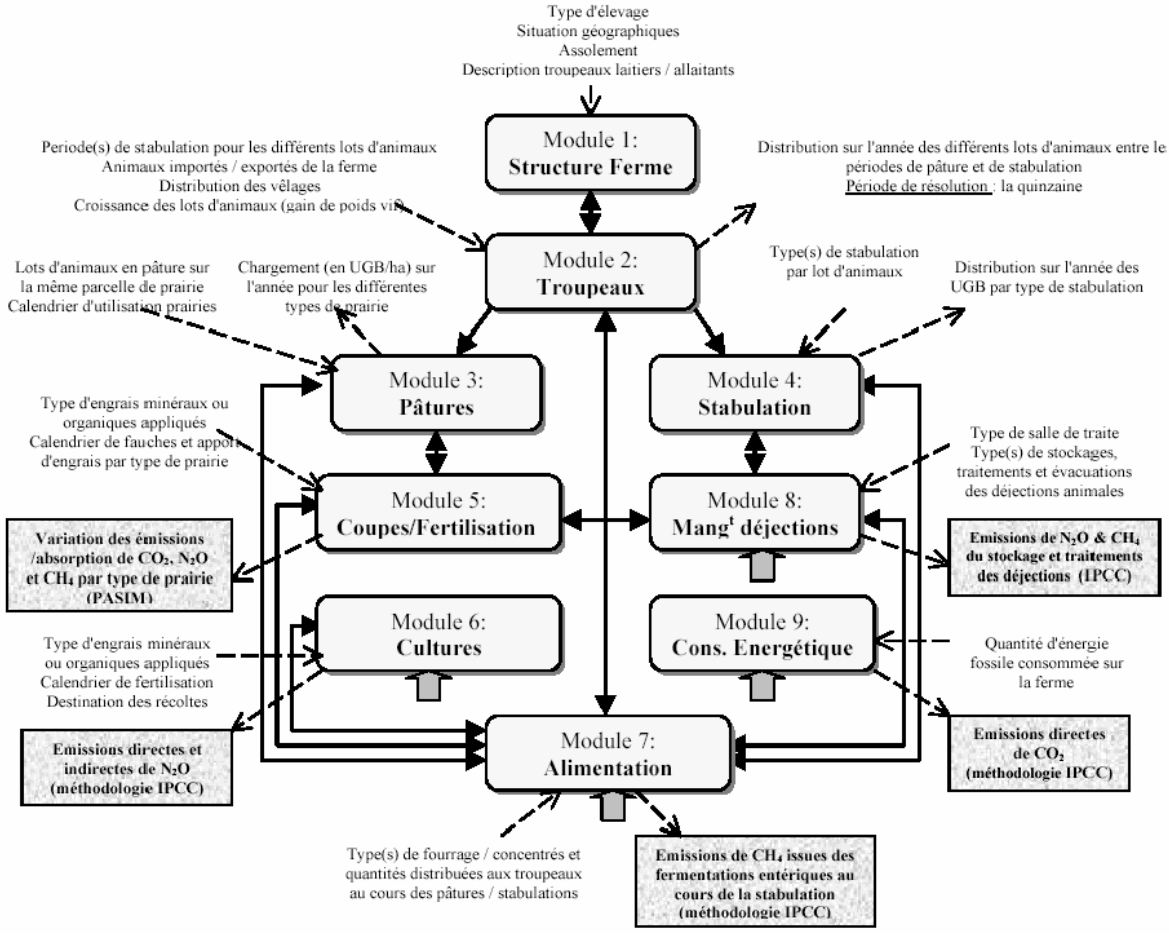


FIGURE 1.1 – Modèle FarmSim (Source [Duretz, 2007])

PaSim (figure 1.2) est un modèle de simulation d'un écosystème prairial. Écrit en FORTRAN 90, il permet de simuler, à l'échelle d'une parcelle, les flux de carbone, d'azote, d'eau et d'énergie entre le sol, la végétation, les animaux et l'atmosphère. Il nécessite pour cela d'avoir des informations sur la météo, la composition du sol, le couvert végétal et la gestion au quotidien (coupe, fertilisation, chargement).

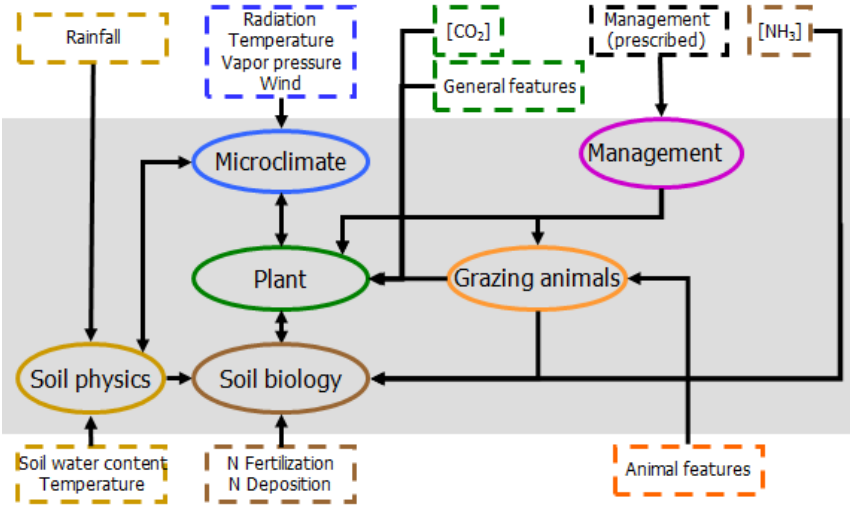


FIGURE 1.2 – Modèle PaSim

CERES (figure 1.3), deuxième modèle sur lequel se base FarmSim, est un modèle de cultures. Il permet de modéliser la croissance et le développement de la plante, les transferts de chaleur, d'eau et de solutés et les différents flux de carbone et de nitrates lors des phases de minéralisation, immobilisation par la plante, nitrification et dénitrification à partir d'informations sur l'atmosphère, le sol et le sous-sol et la gestion du couvert végétal.

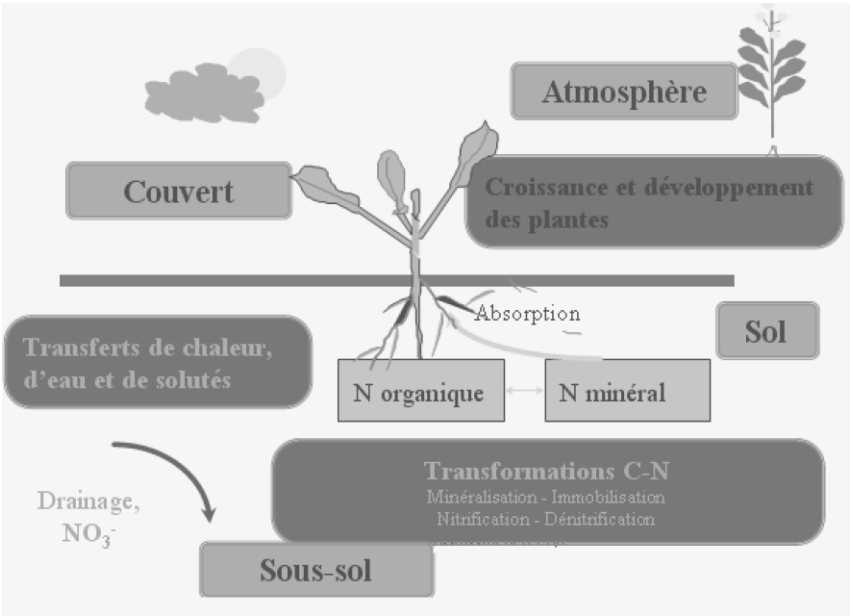


FIGURE 1.3 – Modèle CERES (Source [Duretz, 2007])

Enfin, la méthodologie **IPCC** (**I**ntergovernmental **P**anel on **C**limate **C**hange) est une

méthodologie basée sur des facteurs d'émissions. Ces derniers sont issus des publications scientifiques mondiales. Par exemple, pour déterminer l'énergie de maintenance d'une vache, on utilise la formule ci-dessous :

$$NE_m = Cfi * (Poids)^{0.75}$$

$$Cfi = \begin{cases} 0.386 & vaches\,allaitantes \\ 0.322 & vaches\,non\,allaitante \end{cases}$$

Cette méthodologie a donc l'avantage d'être facilement utilisable, mais elle est fournie en contrepartie des estimations globales. Elle est utilisée dans FarmSim afin de calculer :

1. les émissions de N_2O et de CH_4 issues du stockage des déjections animales
2. les émissions de CH_4 issues de la fermentation entérique au cours de la période de stabulation
3. les émissions directes et indirectes de N_2O issues des cultures.

1.4 Objectif du stage

Mon stage présente **trois** principaux volets.

Tout d'abord, j'ai du me familiariser avec FarmSim. J'ai pour cela :

1. Corrigé les anomalies récentes relevées dans FarmSim
2. Mis à jour les coefficients IPCC
3. Intégré la version tropicalisée de PaSim à FarmSim

Dans un second temps, j'ai implémenté dans FarmSim le mécanisme de changement de pratique (rotation prairie/culture). En effet, au début du stage, les composants de la ferme (prairies, cultures, bâtiments) étaient traités indépendamment et les résultats agrégés par la suite. Il était, par exemple, impossible de passer d'une prairie à une culture d'une année à l'autre.

Enfin, mon dernier objectif était de lancer des simulations à l'échelle de l'Europe afin d'avoir des estimations des bilans de GES à l'échelle du territoire.

1.5 Planning prévisionnel

Le stage s'étend sur une période de 6 mois (24 semaines). Le diagramme de la figure 1.4 représente le temps passé sur chaque fonctionnalité. Il présente le temps planifié en début de stage avec mon responsable.

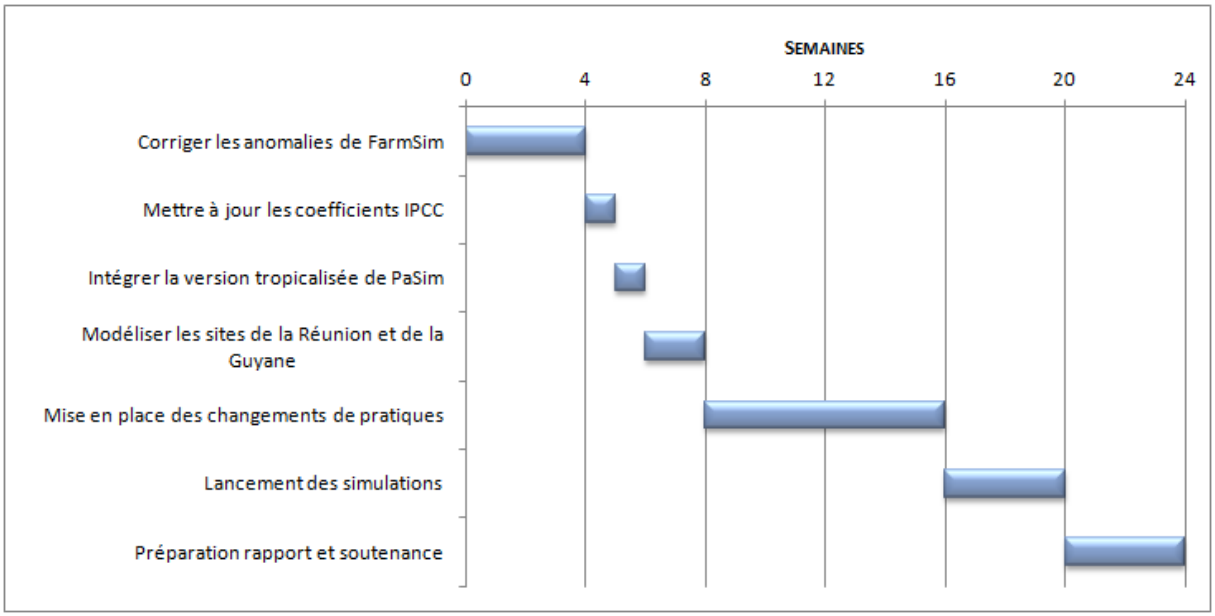


FIGURE 1.4 – Planning prévisionnel

2
Améliorations de FarmSim

Comme indiqué dans la section 1.4, le premier objectif de mon stage a été de me familiariser avec FarmSim. Ce chapitre sera donc consacré à présenter les travaux que j'ai effectué durant cette période.

2.1 Corrections des anomalies de FarmSim

Le code de FarmSim qui m'a été fourni en début de stage contenait des anomalies. Il m'a donc été demandé de les corriger. Cette section listera donc les erreurs les plus significatives, leur causes ainsi que la solution adoptée.

2.1.1 Fichier de log

La première anomalie rencontrée dans le code initial était causée par un problème de droits utilisateurs. En effet, le fichier de log initial de FarmSim était **C:\log_farmsim.log**. Cependant, un utilisateur ne peut créer de dossier dans ce lecteur, à moins d'être un administrateur. Un lancement du programme provoquait l'erreur de la figure 2.1.
Pour résoudre ce problème, j'ai décidé de modifier cette destination. En procédant ainsi, un dossier **logs** se crée automatiquement dans le répertoire d'exécution de FarmSim. La contrainte restante est que FarmSim ne doit pas être dans le répertoire **C:\.**

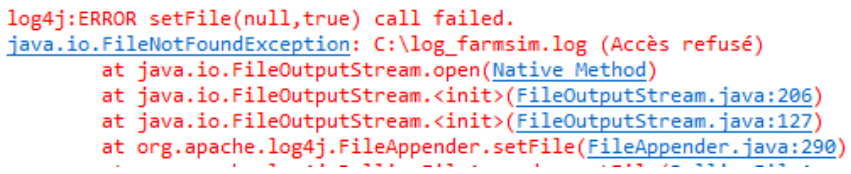
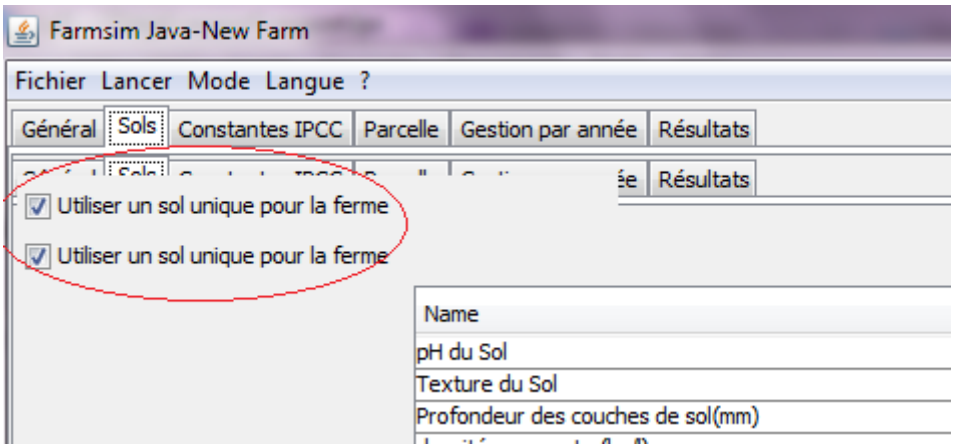


FIGURE 2.1 – Erreur avec le fichier log

2.1.2 Interface graphique

2.1.2.1 Problème d'affichage

Sur certains onglets de l'interface graphique, il arrivait de constater des problèmes d'affichage. Ainsi par exemple certaines informations apparaissaient en double sur l'interface comme le montre la figure ci-dessous :



Le problème venait des méthodes **paintComponent** des conteneurs associés aux onglets en question. D'après [1], cette méthode sert à ajouter des instructions de dessin à un composant. Elle est appelée à chaque fois que ce dernier nécessite d'être redessiné (défilement barre verticale par exemple). Cependant, dans le programme initial de FarmSim, l'arrière plan du composant n'était jamais effacé. C'est ce qui entrainait ces affichages multiples. J'ai corrigé cela en appelant explicitement la méthode **super.paintComponent()**.

2.1.2.2 Sorties des exécutions de PaSim et CERES

Pour pouvoir lancer un programme extérieur, obtenir ses sorties et les afficher dans l'interface, le code de FarmSim fait usage de la méthode **waitFor** de la classe **java.lang.Process.** Cependant, cette méthode cause l'arrêt du thread d'exécution de l'application.
Toutefois, dans un programme SWING classique en Java, les mises à jour des interfaces se font via le Event Dispatch Thread (thread traitant les évènements reçus). Étant donné que celui ci était stoppé, aucune mise à jour n'était effectuée.

Pour résoudre ce problème, il a fallu faire en sorte que l'appel à **Process.waitFor()** se fasse dans un thread différent de l'Event Dispatch Thread. Ainsi, avant le lancement de PaSim ou de CERES, un thread distinct est initialement créé. C'est ensuite lui qui se charge d'exécuter la commande externe. Cela permet par la même occasion, de garder l'interface réactive.

2.1.3 Récupération des données dans les fichiers XML

Dans FarmSim, la récupération des données des fichiers XML se fait avec la bibliothèque **StAX**. Elle présente l'avantage de rendre aisée cette opération. Un problème se posait toutefois avec l'utilisation qui était faite de cette bibliothèque. En effet, les valeurs des attributs des balises étaient récupérées via leur index. En considérant par exemple la balise : **<LATITUDE Degree="45" Min="31" Sec="31"/>**, les données étaient récupérées avec les méthodes Java ci-dessous :

```
reader.getAttributeValue(0); //Degree
reader.getAttributeValue(1); //Min
reader.getAttributeValue(2); //Sec
```

où **reader** représente l'objet **XMLStreamReader** permettant de parser le fichier XML.

Cette façon de procéder n'aurait pas posé de problèmes si la sauvegarde des fichiers XML conservait les attributs dans le bon ordre, or cela n'était pas le cas. Les données sauvegardées n'étaient donc pas chargées correctement.

Pour résoudre ce problème, j'ai décidé de baser la récupération des valeurs des attributs présents dans les fichiers XML sur leur noms. En reprenant la balise exemple présentée un peu plus haut, le code d'extraction des valeurs devient :

```
reader.getAttributeValue(reader.getNamespaceURI(),"Degree")));
reader.getAttributeValue(reader.getNamespaceURI(),"Min")));
reader.getAttributeValue(reader.getNamespaceURI(),"Sec")));
```

Le premier paramètre de cette nouvelle méthode est le nom du namespace dans lequel est défini le nom de la balise . Sa valeur a été laissée à **reader.getNamespaceURI()** (et non à **null**), pour apporter plus de généricité au programme, au cas où la structure de ces fichiers venait à évoluer.

2.1.4 Suppression des fichiers temporaires de sauvegarde

Pour effectuer la sauvegarde des données de la ferme ayant été modifiées par l'utilisateur, FarmSim utilisait des fichiers temporaires. Cela permettait d'éviter de maintenir ouvert les fichiers de configuration trop longtemps. En effet, les données étaient d'abord sauvegardées dans ces fichiers temporaires puis rapatrier vers les fichiers de configuration. Cependant, ils n'étaient pas supprimés après l'opération de sauvegarde, bien que le code pour le réaliser était présent dans FarmSim.

La source du problème était la non fermeture des buffers de lecture des fichiers après le rapatriement des données. Après avoir veillé à fermer ces buffers, la suppression des fichiers temporaires a pu être effectuée.

2.1.5 Mise à l'équilibre de PaSim

Avant que PaSim simule une prairie, il réalise une **mise à l'équilibre** de celle ci afin d'initialiser certaines de ses propriétés(pools de carbone par exemple) [Graux et al., 2013]. A l'issue de cette mise à équilibre, il produit un fichier « **restart** » qui contient toutes les valeurs des propriétés qu'il aurait initialisé. Ce fichier peut ensuite être utilisé pour démarrer la simulation.

Dans FarmSim, il est possible de paramétrer les prairies en spécifiant un fichier restart en entrée ou en indiquant qu'une mise à l'équilibre doit être préalablement réalisée. Si l'utilisation de fichiers restart ne pose pas de problèmes, la mise à l'équilibre pour certains fichiers de paramètres fournis avec FarmSim ne marchait pas.
Pour m'assurer que c'était bien FarmSim qui était à l'origine de l'erreur, j'ai récupéré d'autres données valides et j'ai créé mes propres fichiers d'entrée. Je me suis alors rendu compte qu'avec ces fichiers, la mise à l'équilibre se déroulait correctement.

J'ai donc déduit que ce problème de mise à l'équilibre provenait surement d'une mauvaise configuration de fichiers. En effet, aucune vérification de la cohérence des données n'est effectuée dans PaSim. Des valeurs d'entrées absurdes seraient donc vraisemblablement à l'origine de ces erreurs.

2.2 Schémas XML

2.2.1 Généralités

Bien souvent, il arrive qu'on veuille s'assurer que le contenu d'un fichier XML en entrée d'un programme suive une syntaxe précise (ordre d'apparition des balises, présence d'attributs, etc...). Pour y parvenir, on fait usage de technologies dites de **définition d'un document XML**. Celles ci permettent de définir un ensemble de règles qu'un fichier XML doit respecter afin d'être considéré comme **valide**.

Il existe plusieurs technologies de définition de documents XML, chacune ayant son lot d'avantages et d'inconvénients. Les deux technologies les plus utilisées restent les « **D**ocument **T**ype **D**efinition » (DTD) et les schémas XML.
Afin de savoir laquelle de ces technologies choisir, je me suis documenté sur elles et j'ai pu dresser la table comparative 2.1.

Critères	DTD	Schémas XML
Langage utilisé pour l'écriture	Langage et syntaxe propres	XML
Typage des données	Non. On se contente de dire qu'une balise doit contenir des données mais impossible de spécifier leur types.	Oui
Contraintes sur les données	Non	Oui. Il est possible d'utiliser des expressions regulières, enumérer les valeurs possibles d'un attribut, etc...

TABLE 2.1 – Comparaison des DTD et des schémas XML

On peut voir dans ce tableau que non seulement les DTD apportent moins de fonctionnalités que les schémas XML, mais elles nécessitent en plus l'apprentissage d'un nouveau langage et donc l'utilisation d'une nouvelle API pour les exploiter. J'ai donc décidé de retenir les schémas XML car j'étais familier avec le langage XML et que ceux-ci offrent plus de possibilités (type de données, contraintes sur les valeurs). Cela me permettra de déporter plusieurs vérifications relatives aux données d'entrées au niveau des schémas XML plutôt que dans le code métier de FarmSim.

2.2.2 Syntaxe de déclaration des éléments

Cette section présentera les principes de base au travers d'un exemple. On s'attache ainsi à écrire le **xsd** (schéma XML) permettant de valider le fichier XML suivant :

```xml
<?xml version="1.0"?>
<FARMSIM>
  <YEAR Value="1994">
    <METEO>
      <CO2 Value="380.0" Unity="ppm"></CO2>
      <NH3 Value="2.0" Unity="unknown"></NH3>
      <FILE_ASSOCIATE type="Hourly">/test/1994.csv</FILE_ASSOCIATE>
    </METEO>
  </YEAR>
  <YEAR Value="1995">
    <METEO>
      <CO2 Value="380.0" Unity="ppm"></CO2>
      <NH3 Value="2.0" Unity="unknown"></NH3>
      <FILE_ASSOCIATE type="Hourly">/test/1995.csv</FILE_ASSOCIATE>
    </METEO>
  </YEAR>
<FARMSIM>
```

Selon le contenu d'un élément présent dans le fichier XML, celui-ci peut être considéré comme simple ou complexe.

2.2.2.1 Éléments simples

Un élément simple contient une unique valeur dont le type est **simple.** C'est par exemple le cas de la balise :

```xml
<nom>John</nom>
```

.

Pour déclarer un tel élément dans un schéma XML, il faut utiliser le mot clef **element** :

```xml
<xsd:element name="nom" type="xsd:string" />
```

2.2.2.2 Éléments complexes

Un élément complexe est un élément qui contient d'autres éléments ou attributs. C'est le cas de tous les éléments présents dans le fichier XML exemple. Pour définir un élément complexe, on utilise les mots clef **element** et **complexType :**

```xml
<xsd:element name="METEO">
    <xsd:complexType>
        <!-- contenu ici -->
    <xsd:complexType>
</xsd:element>
```

On distingue trois types de contenu possible pour un élément complexe :
- Les contenus **simples**
- Les contenus « **standards** »
- Les contenus **mixtes** (qui ne seront pas discutés ici)

Le contenu simple désigne le contenu d'un élément complexe composé uniquement d'attributs et d'un texte de type simple. C'est le cas de la balise **FILE_ASSOCIATE** du fichier XML exemple. Pour déclarer un contenu simple, il faut utiliser le mot clef **simpleContent** combiné avec les balises **attribute** (pour la déclaration d'attributs) et **extension** (pour le type du texte).

```
    <xsd:simpleType name="meteoFileDataType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="Daily" />
            <xsd:enumeration value="Hourly" />
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:element name="FILE_ASSOCIATE">
        <xsd:complexType>
            <xsd:simpleContent>
                <xsd:extension base="xsd:string">
                    <xsd:attribute name="type" type="meteoFileDataType" use="required" />
                </xsd:extension>
            </xsd:simpleContent>
        </xsd:complexType>
    </xsd:element>
```

Le contenu standard désigne le contenu d'un élément complexe qui n'est composé que d'autres éléments ou uniquement d'attributs. C'est le cas des balises **FARMSIM**, **YEAR**, **METEO**, **CO2** et **NH3**.
Dans le cas où l'élément ne contient que des attributs, il suffit d'imbriquer les balises **attribute** dans une balise **complexType** :

```
    <xsd:element name="CO2">
        <xsd:complexType name="data">
            <xsd:attribute name="Unity" type="xsd:string" use="required"/>
            <xsd:attribute name="Value" type="xsd:double" use="required"/>
        </xsd:complexType>
    </xsd:element>
```

Par contre, si d'autres éléments sont contenus dans l'élément complexe, il convient de définir l'ordre dans lequel on souhaite les voir apparaitre. On peut pour cela utiliser :
- Le mot clef **all** afin d'indiquer que l'ordre de déclaration des balises enfants n'importe pas
- Le mot clef **sequence** pour spécifier que les balises enfants doivent obligatoirement apparaitre dans le même ordre que celui du schéma XML
- Le mot **choice** pour déclarer des balises enfants exclusives

En prenant le cas de la balise **METEO**, on obtient :

```
    <xsd:complexType name="data">
        <xsd:attribute name="Unity" type="xsd:string" use="required"/>
        <xsd:attribute name="Value" type="xsd:double" use="required"/>
    </xsd:complexType>

    <xsd:element name="METEO">
        <xsd:complexType>
            <xsd:all>
                <xsd:element name="CO2" type="data"/>
                <xsd:element name="NH3" type="data"/>
                <xsd:element ref="FILE_ASSOCIATE"/>
            </xsd:all>
        </xsd:complexType>
    </xsd:element>
```

Enfin, lorsque l'élément complexe contient des éléments et des attributs, il suffit de combiner les deux informations précédentes :

```
38      <xsd:element name="YEAR">
39          <xsd:complexType>
40              <xsd:sequence>
41                  <xsd:element ref="METEO"/>
42              </xsd:sequence>
43              <xsd:attribute name="Value" type="xsd:nonNegativeInteger" use="required"/>
44          </xsd:complexType>
45      </xsd:element>
```

2.2.3 Mise en place de la validation dans FarmSim

Le principe était très simple : avant d'ouvrir un fichier XML, il fallait d'abord le valider. J'ai donc créer la classe **ValidateXML.** Celle ci stocke une table d'association qui, pour chaque type de fichier XML d'entrée possible, associe son schéma XML de validation.
Le point d'entrée avec le reste de l'application se résume à son unique méthode publique :

```
public static void validate(String fileName, XMLContent content);
```

Celle-ci prend en paramètre le nom du fichier d'entrée et un entier (plus précisément la valeur d'une énumération) représentatif du contenu de ce fichier. Via cet entier et la table d'association, il est possible de récupérer le chemin du schéma XML correspondant et de valider le fichier d'entrée. Puis, en cas d'erreur de validation, cette méthode lance une exception. Il devient ainsi possible, quand cela est nécessaire, d'empêcher la simulation de se lancer.

2.3 Mise à jour des coefficients IPCC

Comme mentionné dans la partie 1.3, la méthodologie IPCC est une méthodologie relativement simple qui se base sur des facteurs d'émissions. On estime par exemple qu'une vache laitière produit une certaine quantité de CH_4 par an, et suivant les années de simulation à lancer, on peut déduire la quantité de CH_4 totale produite. Cette méthodologie a toutefois ses limites. Ainsi, elle ne prend pas en compte les variations intra et inter annuelles (météo, variabilité au sein du troupeau) pouvant influer sur les quantités émises. On peut toutefois donner une estimation globale des émissions.

Dans ces formules apparaissent généralement des coefficients dont la valeur varie en fonction du type d'animal. Par exemple, l'énergie de maintenance (en abrégé NE_m) est déterminée à partir de la formule :

$$NE_m = Cfi * (Poids)^{0.75}$$

où le coefficient **Cfi** peut prendre différentes valeurs :

$$Cfi = \begin{cases} 0.386 & vaches\ allaitantes \\ 0.322 & vaches\ non\ allaitantes \end{cases}$$

Pour parvenir à mettre à jour ces coefficients IPCC, mon travail s'est subdivisé en deux parties. J'ai d'abord réalisé une documentation (au format PDF) des formules IPCC actuellement présentes dans FarmSim ainsi que des valeurs des coefficients IPCC s'y afférant. Ce document a ensuite été transmis à un scientifique pour validation.

A la suite de cette validation, j'ai pu récupérer les nouvelles valeurs des coefficients et les intégrer dans le modèle. Pour améliorer la lisibilité du code et faciliter les futures mises à jour de ces coefficients , j'ai regroupé toutes les formules IPCC dans une unique classe **IPCCConstant** dont l'instanciation est assurée par le biais d'un design pattern **Singleton**.

2.4 Intégration version tropicalisée de PaSim

PaSim avait été initialement développé pour des climats tempérés. De récentes demandes ont nécessité le développement d'une version tropicalisée de ce modèle. Afin que FarmSim profite de cette version, il a été décidé de l'intégrer au modèle.

Toutefois, cette intégration introduit deux modèles pour simuler les prairies. Il a donc été décidé, après discussion avec mon maitre de stage et mon responsable ISIMA, d'ajouter une fonctionnalité permettant de rendre possible le choix des modèles permettant de simuler les prairies, les cultures et les bâtiments. Il deviendra par la même occasion aisé d'intégrer ultérieurement de nouveaux modèles.

Dans cette section, je vais donc commencer par présenter l'architecture que j'ai utilisée, puis je vais présenter les modifications apportées à l'interface graphique afin de permettre à l'utilisateur de changer facilement de modèles.

2.4.1 Architecture de gestion des modèles

Lorsqu'il s'agit de réaliser une architecture modulaire, il convient de se tourner en premier lieu vers des **patrons de conception**. Il en existe de plusieurs catégories. C'est l'objectif à atteindre qui détermine vers quelle catégorie se tourner. Dans notre cas, le patron de conception qui apparaissait le plus adapté est le pattern **Stratégie** [Gamma et al., 1994]. C'est donc lui que j'ai utilisé.

Le fonctionnement de ce pattern est résumé dans la figure 2.2.

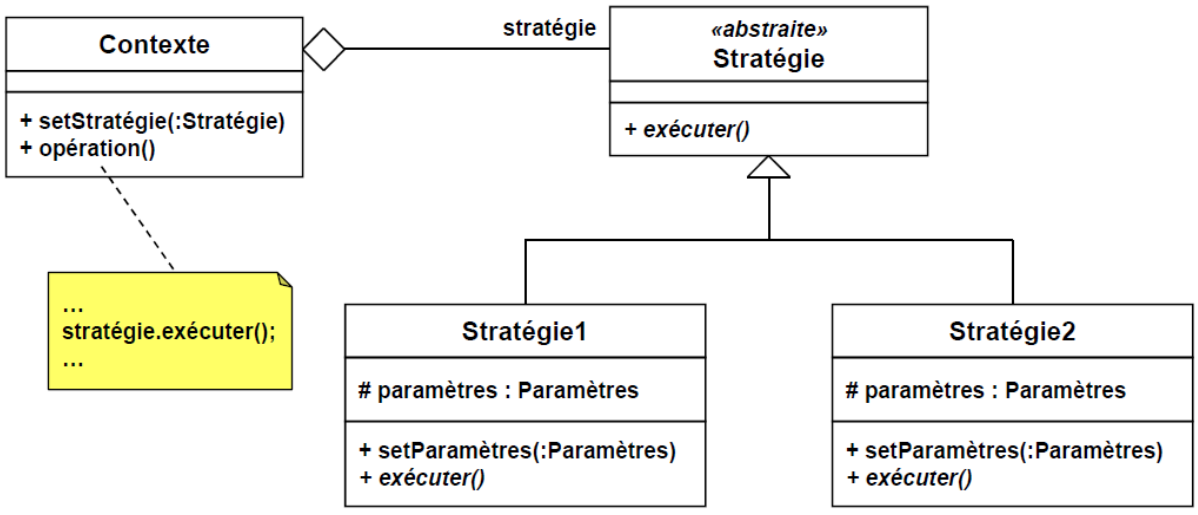


FIGURE 2.2 – Design pattern stratégie

Dans cette figure, on peut voir que, juste avec la méthode **setStratégie**, il est possible de changer complètement le contexte utilisé par l'application (passé de la Stratégie1 à la Stratégie2).

Afin de mettre en place cette architecture dans FarmSim, il fallait d'abord déterminer les méthodes qui allaient permettre de manipuler les modèles (à l'image de la méthode **exécuter** de la classe abstraite **Stratégie**). Cela nécessitait d'avoir une certaine compatibilité entre les modèles du même type.
C'est une problématique qui avait été longuement discutée lors d'une réunion de miparcours fin mai. En effet, concevoir un programme complètement indépendant des modèles utilisés n'est pas une chose aisée. Chacun de ces modèles peut nécessiter une adaptation particulière en terme de variables à paramétrer, modèles annexes à utiliser, etc...

Pour le moment, je me suis positionné par rapport aux modèles disponibles. J'ai ainsi pu dégager **cinq** méthodes afin de les manipuler. Ces méthodes ainsi que leur rôles sont consignés dans la table 2.2.

Méthodes	**Rôle**
getModelName	Retourne le nom du modèle manipulé
run	Exécuter le modèle
generateTableResults	Consigne les résultats de la simulation dans un tableau d'une feuille Excel
generateGraphicalResults	Génère des résultats graphiques de la simulation dans une feuille Excel
getTotalCH4Emissions	Retourne les émissions de CH_4
getTotalCO2Emissions	Retourne les émissions de CO_2
getTotalN2OEmissions	Retourne les émissions de N_20

TABLE 2.2 – Méthodes de gestion des modèles utilisés par FarmSim

2.4.2 Modifications apportées à l'interface graphique

Ayant mis en place l'architecture présentée ci-dessus, il fallait permettre à l'utilisateur de changer facilement de modèles. Pour cela j'ai mis à jour l'interface graphique en rajoutant des **Combobox** pour chaque type de modèles comme le montre la figure 2.3.

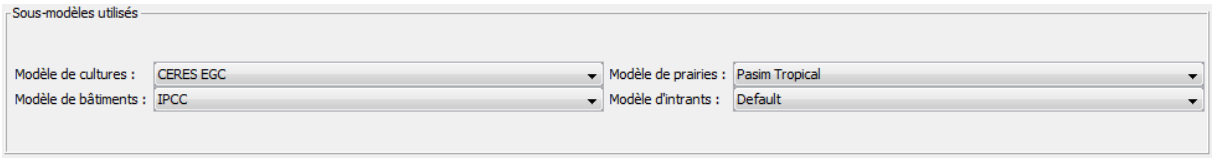


FIGURE 2.3 – Possibilité de choisir les modèles via l'interface graphique

Aussi, étant donné que certains composants de l'interface graphique dépendent du modèle utilisé, une fonctionnalité qui s'avère utile est la désactivation des informations non utiles pour l'utilisateur. Ainsi, l'utilisateur ne se préoccupe que des informations indispensables à l'exécution des modèles qu'il choisit.

Pour se faire, j'ai ajouté une nouvelle classe à FarmSim : **FieldsVisibilty**. Au lancement de l'application, cette classe charge des données dans un fichier XML nommé **guiFieldsAndModelsCompatibility**. Dans ce fichier, on retrouve pour chaque composant (spécifique à un ou plusieurs modèles) de l'interface les modèles avec lesquels il est compatible. Ainsi, lorsque l'utilisateur choisit un nouveau modèle dans l'interface, cette classe récupère tous les composants de l'interface, détermine ceux qui doivent être désactivés ou réactivés à l'aide des données préalablement chargées et réalise l'opération correspondante.

Cette opération dépend du type de composants. J'ai répertorié **trois** grandes catégories de composants : les **onglets**, les **tables** et les **composants standards** (les autres composants).
Pour un composant standard, il suffit de faire appel à la méthode **setVisible** de la classe **java.awt.Component** afin de rendre ce dernier visible ou non.
Par contre, utiliser la méthode **setVisible** sur un onglet n'a aucun effet. Il faut utiliser la méthode **setEnabledAt(index,status)** dans le classeur père contenant l'onglet en question.
Enfin, le dernier cas particulier concerne les tables. Dans le cas où l'on veut désactiver la table en entier, on peut la considérer comme un composant standard. Cependant, rendre invisible une seule ligne ou colonne d'une table est impossible. En effet, les tables suivent le design pattern Modèle-Vue-Contrôleur (MVC). De ce fait, on ne peut changer le modèle (les données) qu'à leur construction (instanciation). Afin de contourner cette limite, j'ai décidé, lorsque les informations demandées dans un tableau ne sont pas nécessaires, de les rendre non éditables et sous un fond noir comme le montre la figure 2.4. Les tables ne sont donc plus paramétrables uniquement à l'aide de leur noms mais également à l'aide des index des lignes et colonnes.

FIGURE 2.4 – Désactivation des lignes des tables

Au vu des différences énumérées ci-dessus, il me fallait adapter le fichier **guiFieldsAndModelsCompatibility** afin de savoir le type de composant paramétré. Ce fichier a donc été subdivisé en trois grandes balises : **tabs** (pour les onglets), **tables** (pour les tables) et **others** (pour les autres composants). Puis à l'intérieur de ces balises on retrouve les informations concernant les compatibilités entre modèles et composants, comme le montre la figure 2.5.

Ce fichier a été conçu de sorte à pouvoir facilement s'adapter à l'évolution de FarmSim. Ainsi, que ce soit l'ajout d'un nouveau composant ou la prise en compte d'un nouveau modèle, seul ce fichier aura besoin d'être modifié afin de réaliser la mise à jour de l'interface (désactivation/activation des composants). Cela est rendu possible grâce aux attributs **name** et **compatibleModels** et à l'architecture des modèles présentée dans la partie précédente.

En effet, la valeur de l'attribut **name** correspond à la valeur retournée par la méthode

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ROOT>
    <tabs>
        <var name="buildingTab" compatibleModels="IPCC" />
        <var name="croplandTab" compatibleModels="CERES EGC" />
        <var name="grasslandTab" compatibleModels="Pasim Tropical;Pasim Tempere" />
    </tabs>
    <tables>
        <var name="soilNonExpertValues">
            <rows>
                <row id="1" compatibleModels="IPCC" />
            </rows>
            <columns>
                <column id="1" compatibleModels="Pasim Tropical;Pasim Tempere" />
            </columns>
        </var>
    </tables>
    <others>
        <var name="FfeedHousConc" compatibleModels="IPCC" />
    </others>
</ROOT>
```

FIGURE 2.5 – Fichier de configuration des compatibilités entre modèles et composants

java.awt.Component.getName(). Elle sert à identifier un composant de manière unique dans toute l'application. Les composants de l'interface étant récupérés de façon dynamique, il suffit que le nom d'un composant nouvellement ajouté à l'interface soit présent dans le fichier **guiFieldsAndModelsCompatibility** pour que le traitement de celui-ci soit réalisé.
Quant à l'attribut **compatibleModels**, il contient le nom des modèles dont dépend un composant. Ces noms sont identiques à ceux retournés par la méthode **getModelName()** décrite dans la table 2.2. Ainsi quelque soit le modèle rajouté par la suite, son nom sera toujours accessible dans la classe **FieldsVisibilty** et peut donc être directement utilisé dans le fichier **guiFieldsAndModelsCompatibility**.

2.5 Intégration de FarmSim à la plateforme SourceSup

SourceSup est une plateforme d'hébergements de projets informatiques de l'Enseignement Supérieur et de la Recherche. Elle fournit un environnement technique permettant aux chercheurs de travailler en communauté et aussi de mieux gérer l'activité autour de ces projets.
Durant mon stage, j'ai eu à me documenter sur les fonctionnalités offertes par cette plateforme, et ce afin de pouvoir mettre en place un environnement de travail type pouvant être réutilisé par la suite dans d'autres projets.

Dans cette partie, je vais uniquement présenter les outils techniques que j'ai eu à utiliser (pour plus de renseignement, se référer à [2]).

2.5.1 Intégration continue avec Jenkins

2.5.1.1 Intégration continue ?

L'intégration continue est un ensemble de pratiques utilisées en génie logiciel. Elles consistent à vérifier **automatiquement** après chaque **modification de code source** que le résultat des modifications ne produit pas de **régression** de l'application en cours de développement [1].
Pour cela, cette technique doit être en mesure de :
– Contrôler l'intégrité unitaire et fonctionnelle
– Mesurer la couverture du code
– Générer des rapports synthétiques utiles
C'est à ce niveau qu'intervient les serveurs d'intégration continue. Leur objectif sera d'automatiser l'ensemble de ces tâches.

2.5.1.2 Jenkins

Jenkins (figure 2.6) est un serveur d'intégration continue écrit en Java. Il s'interface avec des systèmes de gestions de version (Subversion, Git). Son fonctionnement passe par la création de jobs. Le but d'un job est de construire ou '**builder**' un projet et d'effectuer un ensemble d'actions **post-construction**.
Construire un projet ne se restreint pas uniquement à sa compilation. C'est un terme qui désigne toutes les actions que l'utilisateur souhaite automatiser. Parmi ces actions, on peut citer par exemple :
– Exécuter un script Ant
– Exécuter un script shell
– Exécuter une ligne de commandes batch windows
Pour la construction du projet FarmSim, j'ai décidé d'utiliser des scripts **Ant**. Ceux ci étant destinés à des projets de type Java, ils avaient l'avantage de faciliter les tâches communes (compilation des sources, génération de la javadoc, lancement des tests, etc...). J'ai également rajouté une autre action dont le but est de lancer une analyse de code avec SonarQube (voir partie 2.5.2).

Les actions post-construction quant à elles s'exécutent à la suite de l'étape de construction et se basent (généralement) sur les résultats produits par celle-ci (javadoc, résultats des tests unitaires). On peut par exemple trouver comme action post-construction :
– La publication de la Javadoc (dans l'interface de Jenkins)
– La notification par email (en cas d'échec de construction)

Lors de l'écriture du script Ant pour la construction du projet FarmSim, je me suis confronté à des problèmes d'encodage. En effet, au départ du stage, après avoir importé le code source initial de FarmSim dans mon **IDE** de développement (Eclipse), je me suis rendu compte que certains caractères (lettres accentuées notamment) ne s'affichaient pas correctement. J'ai donc configurer, dans mon IDE, l'encodage des fichiers sources en **UTF-8** qui est un codage permettant de gérer de nombreux alphabets (dont le Français). Cependant, cela n'avait pas été fait (au départ) dans mon script Ant. Il conviendra donc de veiller, lors d'améliorations futures de FarmSim, que l'encodage des fichiers sources soit le « **UTF-8** » ou si cet encodage n'est pas disponible sur le système hôte, de changer

1. Source Wikipédia

FIGURE 2.6 – Job de construction de FarmSim

l'encodage défini dans le script Ant de sorte qu'il corresponde à celui des fichiers sources. J'ai rendu cette dernière tâche moins ardue en spécifiant une propriété **src.encoding** dans le fichier **build_properties** associé à mon script Ant comme le montre la figure 2.7. L'annexe A montre comment importer correctement FarmSim dans l'IDE Eclipse.

Script Ant

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="farmsim" default="testReports" basedir=".">
    <description>Farmsim build file</description>
    <!-- Properties file -->
    <property file="build_properties.properties" />
    ...
    <target name="compile" depends="init-compile">
        <javac srcdir="${src.dir}" destdir="${bin.dir}"
               encoding="${src.encoding}" ... >
            <classpath refid="project.buildpath" />
        <javac>
        <copy todir="${bin.dir}">
            <fileset dir="${src.dir}" excludes="**/*.java"/>
        </copy>
    </target>
    ....
</project>

```

Fichier build_properties

```
src.encoding=UTF-8
src.dir=src
bin.dir=bin
external.lib.dir=library
lib.dir=dist/lib
tests.reports.dir=dist/tests-reports
javadoc.dir=dist/documentation/api
main.class=fr.inra.farmsim.Principal

```

FIGURE 2.7 – Extrait du script Ant

2.5.2 Analyse de code avec SonarQube

SonarQube est un outil qui permet de réaliser une analyse qualitative du code source. Il couvre **7** axes de qualité du code [3] : l'architecture, les commentaires, le respect des conventions de codage, les anomalies potentielles, la complexité, les tests unitaires et les duplications.
Un exemple de sortie possible de Sonar est représenté sur la figure 2.8.

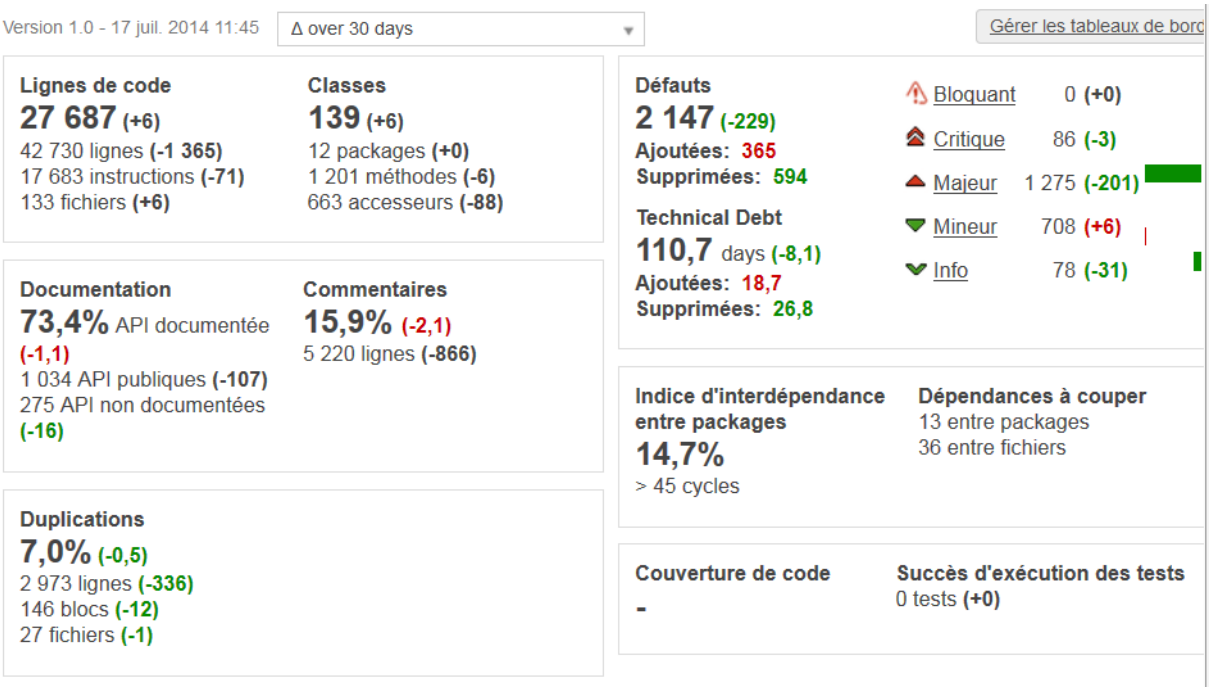


FIGURE 2.8 – Exemple de sortie de SonarQube

Il convient toutefois d'avoir une analyse critique des améliorations suggérées par cet outil. Un cas que j'ai rencontré lors de mon stage était la suggestion du non usage de la sortie console standard : **System.out** pour l'affichage d'informations à l'utilisateur. Elle préconise plutôt l'utilisation des **loggers**, ceux-ci étaient configurables. Si une telle suggestion à tout son intérêt dans un programme graphique, elle perd totalement sa pertinence dans un programme console nécessitant d'interagir avec l'utilisateur (attente de données saisies au clavier).

3
Mise en place de la rotation entre parcelles

Comme indiqué dans la partie 1.3, la problématique principale du stage concerne les pratiques de rotations pluriannuelles, c'est à dire l'implémentation des changements d'une année sur l'autre de la répartition des cultures sur le parcellaire de l'exploitation. En effet, FarmSim considère actuellement une parcelle de façon atomique et fait abstraction de ses propriétés antérieures. On ne peut choisir pour une année et une parcelle données, qu'un unique type de pratique : prairie ou culture. Ensuite, la simulation s'effectue par année puis par parcelle avec absence de liens entre années consécutives.

Cette situation pose deux problèmes.
Tout d'abord, il est uniquement possible d'associer une pratique à une parcelle par an or un exploitant peut modifier la pratique de sa parcelle en cours d'année.
De plus, l'historique et donc les effets des pratiques sur le sol ne sont pas pris en compte d'une année sur l'autre.

Ce chapitre sera donc consacré à présenter la manière dont j'ai intégré ces modifications à FarmSim.

3.1 D'un modèle orienté année à un modèle orienté parcelle

3.1.1 Mise à jour du modèle

3.1.1.1 Problématique

Dans le modèle initial, deux classes étaient chargées de stocker les données de la ferme :
- La classe **FarmManager** pour toutes les données statiques, c'est à dire qui ne changent pas d'une année sur l'autre. Il s'agit des données concernant les parcelles, le sol (initial) et les constantes IPCC.
- La classe **RotationManager** pour toutes les données dynamiques, c'est à dire qui changent d'une année sur l'autre : prairies, cultures, jachères, troupeaux, intrants, données météo.

Si le fait de passer d'un modèle orienté année à un modèle orienté parcelle ne suscitait pas d'interrogations par rapport à la classe FarmManager, ce n'était pas le cas pour la classe RotationManager. En effet, pour plus de clarté, il aurait été judicieux de répartir les données dynamiques non plus uniquement en fonction de l'année, mais également en fonction de la parcelle. Il serait ainsi possible, pour une parcelle donnée, de récupérer automatiquement toutes les données la concernant et ce, pour toutes les années de la simulation.

Un deuxième problème qui se posait concernait la configuration des modèles de prairies et de cultures. En effet, dans le code initial de FarmSim, il n'était possible de lancer ces derniers que sur une année. Les pratiques pouvant être maintenant modifiées au cours de l'année, la durée des pratiques devient indéterminée (inférieure ou supérieure à une année). Les fichiers de configuration de ces modèles devaient donc être modifiés de sorte à intégrer ce changement.

Enfin, le dernière problème concernait le bilan que doit effectuer FarmSim pour chaque année de simulation. Étant donné que nous devrions être en mesure de configurer une parcelle sur un temps indéterminé, les fichiers de résultats produits à l'issue des simulations des modèles de prairies et/ou cultures contiendront des données sur des périodes d'une année ou plus. En effet, un modèle, qu'il soit lancé sur une ou plusieurs années, ne produira qu'un ou plusieurs fichiers de sortie contenant les résultats cumulés de toutes les années simulées. L'objectif principal de FarmSim étant de pouvoir faire un bilan chaque année, il fallait trouver un moyen de scinder ces fichiers de résultats globaux en plusieurs sous fichiers qui contiendront les résultats pour chaque année simulée.

3.1.1.2 Solution mise en place

3.1.1.2.1 Architecture

Du fait de l'historique de FarmSim (5 ans de développement), réaliser un changement d'architecture (modification de l'existant) pourrait être la source de nombreuses incohérences et m'obligerait à une restructuration du code couteuse en temps. C'est pour cette raison que j'ai décidé de conserver l'architecture actuelle, puis de rajouter des informations supplémentaires qui me permettront, quand cela est nécessaire, de transformer la vision annuelle que FarmSim a des données de la ferme en vision par parcelle. Cela s'est fait en deux étapes.

La première consistait à rajouter des informations de date aux pratiques de la ferme. On devait être en mesure d'indiquer qu'une pratique commençait à une date **d1** et se terminait à une date **d2**. J'ai donc ajouté un nouvel attribut à la classe **PlotManagement** (classe mère des classes relatives aux différentes pratiques) :

```
protected Date[] period; //period[0]:début; period[1]:fin
```

Ensuite, cette classe implémente l'interface **Comparable** afin de pouvoir ordonner les pratiques par ordre de début. Ainsi, il sera possible de lancer les simulations de façon chronologique sur les pratiques associées aux parcelles :

```
class PlotManagement implements Comparable<PlotManagement>{
    ...
    @Override
```

```
    public int compareTo(PlotManagement plotManagement) {
        return this.period[0].compareTo(plotManagement.period[0]);
    }
    ...
}
```

La deuxième étape de cette restructuration avait quant à elle pour but de mettre en place une routine me permettant de concaténer plusieurs pratiques successives. En effet, la classe **RotationManager** ayant conservée son architecture initiale, configurer une pratique sur trois ans revient à configurer la même pratique pour toutes les années de cette période. En d'autres termes, la classe **RotationManager** contiendra trois fois des informations sur cette pratique. Le but de cette routine est donc d'intégrer les informations des pratiques des deux dernières années dans celle de la première année de sorte à ce que FarmSim puisse, avec uniquement les données de la première année, savoir automatiquement que la pratique est en fait configurée pour trois ans.
J'ai pour cela rajouté deux nouveaux attributs à la classe **PlotManagement :**

```
protected List<PlotManagement> mergedPlots;
protected boolean active;
```

Le premier attribut indique les pratiques qui doivent être fusionnées avec la pratique courante et le second permet d'indiquer si une simulation doit être lancée pour cette dernière. En prenant l'exemple de la simulation de prairies sur trois ans présentée ci-dessus, la prairie de la première année intègrera les deux autres prairies dans l'attribut **mergedPlots** et l'attribut **active** sera à **true**. Les deux autres prairies auront une liste de **mergedPlots** vide et un attribut **active** à **false**, indiquant ainsi que les modèles ne doivent pas directement les simuler. FarmSim pourra ainsi savoir que la première prairie est associée à d'autres et pourra créer les fichiers de paramètres en conséquence (changer le nombre d'années de simulation par exemple).

L'algorithme de principe pour le lancement d'une simulation multi-années devient donc :

```
Fusionner les pratiques;
pour chaque année faire
    Recupérer les pratiques de l'année courante via la classe RotationManager;
    Trier ces pratiques par date de début croissant;
    pour chaque pratique dans la liste triée de pratiques faire
        si la pratique est activée alors
            Lancer la simulation de cette pratique (éventuellement sur plusieurs
            années);
        fin
    fin
    Réaliser le bilan d'émissions de l'année;
fin
Annuler la fusion de pratiques (pour d'éventuelles simulations sur une seule année);
```

Algorithme 1: Algorithme de principe d'une simulation multi-année

3.1.1.2.2 Fichier de configuration des modèles

Maintenant qu'il est possible de configurer les parcelles sur plusieurs années, il fallait adapter les fichiers de configuration des modèles. Tout d'abord, il fallait m'assurer que cela était possible pour tous les modèles présents dans FarmSim. A l'heure d'écriture du rapport, seuls PaSim et CERES étaient présents et ceux-ci pouvaient être lancés sur plusieurs années.
Pour PaSim, seuls les fichiers **météo, management** et **param** nécessitent une modification. Le fichier **météo** doit contenir les données météo pour toutes les années à simuler. Dans le fichier **management**, il faut renseigner les informations pour chaque année sur une ligne distincte. Enfin dans le fichier **param**, les paramètres **CYCLE_METEO** et **CYCLE_GESTION** doivent avoir une valeur égale au nombre d'années à simuler. Les paramètres **TSTART** et **TSTOP** sont ensuite utilisés pour indiquer respectivement les jours julien de début et de fin de la simulation [Graux et al., 2013].

Pour CERES, tout dépend de la longueur du fichier **météo** et de la dernière ligne du fichier **management** (fichier .itk). En effet, à la fin de ce fichier, on peut entrer la valeur **STOP** pour indiquer à CERES de s'arrêter après avoir simuler une année, ou **MONO** pour indiquer à CERES de continuer la simulation en fonction des données présentes dans le fichier **météo**. Par exemple, si le fichier **météo** contient des données sur deux ans et que le fichier management se termine par **STOP**, CERES tournera que sur un an. Alors que si le fichier management se termine par **MONO** alors CERES tournera sur deux ans.

Une fois ces informations en main, l'implémentation s'est faite aisément. J'ai ajouté une méthode **getNbYears()** à la classe **PlotManagement**. L'utilité de cette méthode est de retourner les années sur lesquelles une pratique existe. Attention toutefois car le résultat fourni par cette méthode ne correspond pas au nombre d'années en terme de jours. Par exemple, si une pratique débute le 31/12/2000 et se termine le 01/01/2001, alors le nombre d'années retourné sera de 2.
Les fichiers de configurations des modèles font donc usage de cette méthode pour s'adapter à tout type de pratiques (sur une année ou sur plusieurs).

3.1.1.2.3 Bilan par année

Les résultats produits par FarmSim chaque année, sont stockés dans des répertoires du même nom que l'année simulée. Puis dans ces fichiers, on retrouve les fichiers de résultats de chaque modèle comme le montre la figure 3.1.
Initialement, ces fichiers étaient obtenus par simple copie des fichiers produits par les modèles de prairies et de cultures. Or comme mentionné dans la partie 3.1.1.1, un modèle, qu'il soit lancé sur une ou plusieurs années, ne produira qu'un ou plusieurs fichiers de sortie contenant les résultats cumulés de toutes les années simulées. Conserver un tel fonctionnement aurait comme résultat le stockage de résultats multi-années dans un unique répertoire.

Pour régler ce problème, à la fin d'une simulation, les fichiers de résultats globaux sont scindés en plusieurs sous fichiers. En effet, il est toujours possible, via l'attribut **mergedPlots** de la classe **PlotManagement**, pour une prairie configurée sur plusieurs années, d'établir la correspondance entre année simulée et indice de lignes du fichier résultat global. Par exemple, si l'on lance PaSim sur trois années, on aura un fichier résultat global

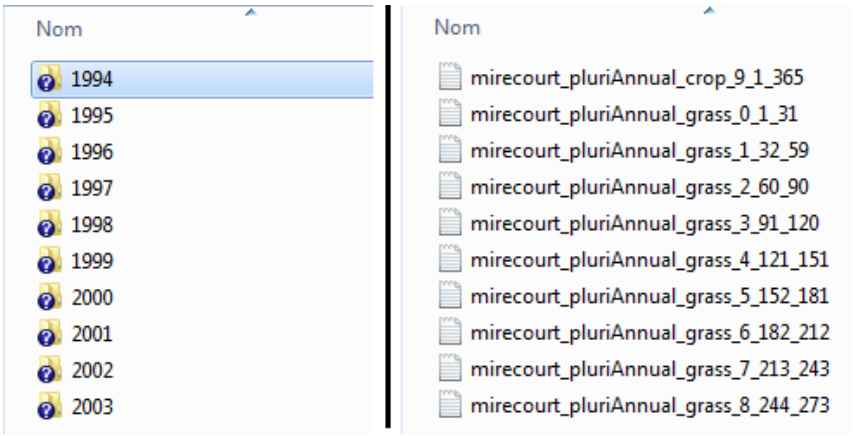


FIGURE 3.1 – Fichier résultat d'une simulation FarmSim

de 365*3 lignes. On peut donc déduire que les 365 premières lignes correspondent aux résultats de l'année n°1, les 365 suivantes à ceux de l'année n°2 et ainsi de suite.
Le traitement est encore plus facilité pour CERES car les fichiers résultats de ce dernier se présentent sous la forme d'une ligne par année.

Toutefois, pour plus de clarté et aussi pour faciliter l'intégration de ces données à l'onglet **Résultats** de l'interface graphique (voir partie 3.1.2.2), j'ai décidé de renommer les sous fichiers résultats de la manière suivante :
<nom de ferme>_<type de pratique>_<indice de parcelle>_<jour début>_<jour fin>.
Le type de pratique prend deux valeurs possibles :
– **crop** dans le cas d'une culture
– **grassland** dans le cas d'une prairie
L'indice de la parcelle est une donnée purement interne à FarmSim et sert à les identifier de façon unique.

3.1.2 Mise à jour de l'interface graphique

3.1.2.1 Onglets liés à la gestion des pratiques

Le fait de conserver l'architecture de base de FarmSim a également permis de conserver l'essentiel de l'interface graphique. Ainsi les onglets **Prairies**, **Cultures** et **Jachères** ne contiennent que deux composants en plus : une liste permettant de paramétrer leur période ainsi qu'un bouton « **Configurer** » permettant de configurer les autres informations relatives à la pratique (informations initialement présentes dans l'onglet **Gestion des Parcelles**) comme le montre la figure 3.2.
Par la suite, l'onglet **Gestion des Parcelles** a également été modifié afin d'apporter une aide visuelle à l'utilisateur. En effet, celui-ci aura l'opportunité de voir, pour une année donnée, la répartition des pratiques sur une parcelle comme le montre la figure 3.3.
Il pourra également via le bouton **Ajouter** de cette même interface, ajouter une nouvelle pratique sur les jours encore disponibles de l'année. Toutefois, une fois les pratiques créés dans cette interface, elles ne peuvent être modifiées que dans leur onglet correspondant (Prairies, Cultures ou Jachères).

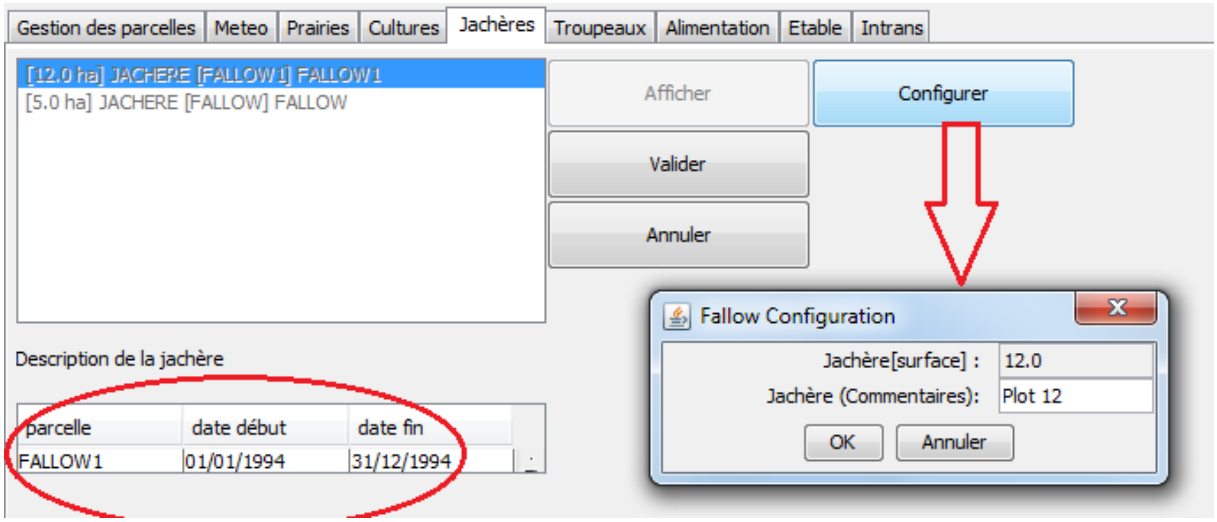


FIGURE 3.2 – Nouvel aspect de l'interface de configuration des jachères

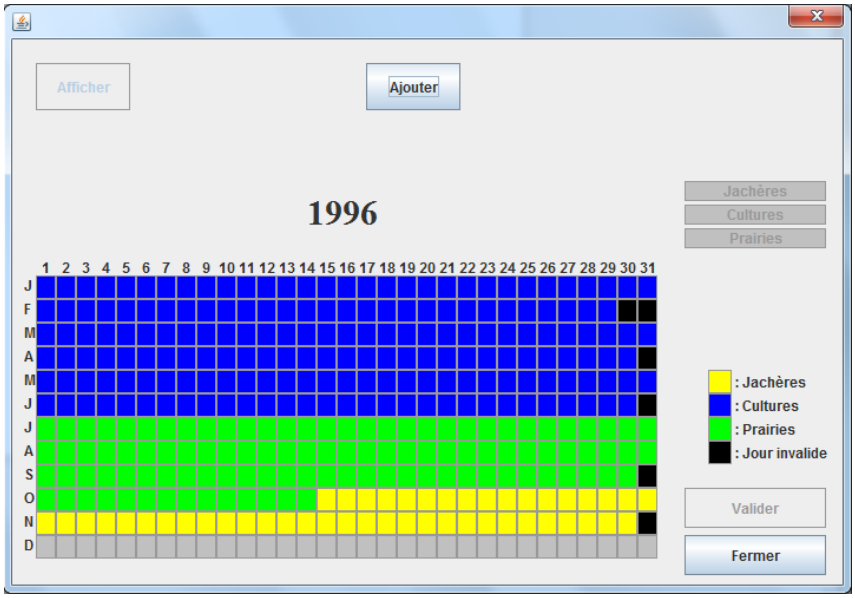


FIGURE 3.3 – Visualisation d'une parcelle

3.1.2.2 Onglet résultats

Aucune modification de l'aspect visuel de l'onglet Résultats n'a été effectuée. Cependant, cet onglet faisait deux hypothèses fortes quant aux données présentes dans les répertoires de résultats.
La première était qu'une culture ou une prairie pratiquée l'année N perdurait toutes les années suivantes de simulation. L'onglet s'attendait donc à avoir un nombre de fichiers résultats identiques pour chaque année.
La deuxième était que les fichiers de résultats des prairies contenaient toujours 365 lignes de données (simulation sur une année). Cependant, avec la période configurable, il est possible de simuler une prairie sur une période de 6 mois par exemple. De plus cette dernière ne débute pas forcément le 1er Janvier.
Ainsi, les modifications apportées à cet onglet concernent uniquement le code métier de récupération des données. J'ai pour cela ajouté la méthode **initialiserDonnees** à la classe **InterfaceResultats**. Cette méthode permet d'initialiser, pour toutes les parcelles et pour

chaque jour de l'année, la valeur des résultats des prairies et cultures à **zéro**. Ensuite, via les noms spécifiques des fichiers de résultats (présentés dans la partie 3.1.1.2.3), je suis en mesure de modifier la valeur pour un jour précis de l'année.

3.2 Rotations entre parcelles

3.2.1 Théorie

Cette section présentera les algorithmes de principes qui permettront de réaliser les rotations dans FarmSim. Elle reprend les éléments essentiels du document [Carozzi et al., 2014] qui m'a été fourni lors du stage.

3.2.1.1 D'une prairie à une prairie

Deux cas sont à distinguer :

1. Un changement de prairies en fin d'année
2. Deux prairies séparées par une période de labour

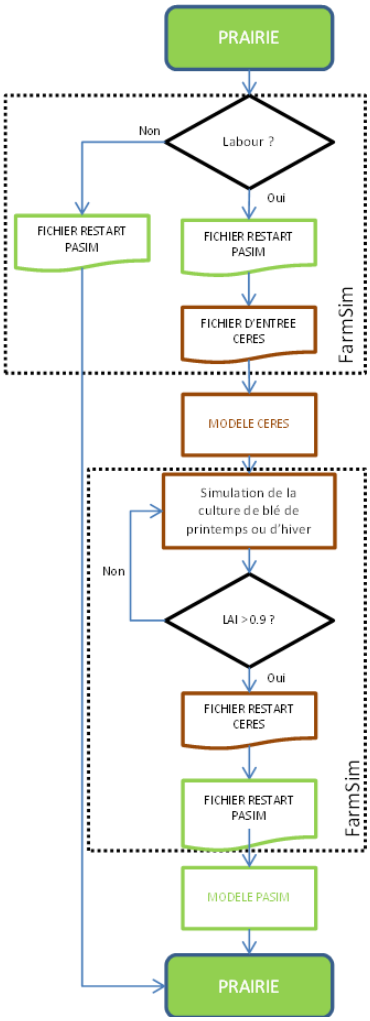


FIGURE 3.4 – Rotation prairie-prairie

Dans le premier cas, PaSim génère, à la fin de la simulation de la première année, un fichier **restart**. Ce fichier est ensuite utilisé en entrée de la simulation pour l'année suivante. Les propriétés du sol de la nouvelle prairie sont donc bien initialisées avec les propriétés du sol telles qu'elles étaient au 31 Décembre de la première année.

Dans le deuxième cas, FarmSim doit faire usage de PaSim et de CERES en coordination. En effet, l'une des insuffisances relevées dans PaSim est qu'il n'arrive pas à gérer correctement les prairies temporaires. L'idée sera donc de faire usage de CERES pour simuler la période de labour.
Ainsi, PaSim simule la prairie jusqu'à la date de labour. A cette date il génère un fichier restart. FarmSim utilise ce fichier restart pour générer des fichiers d'entrée de CERES qui simule la prairie en la considérant comme une culture de blé de printemps ou d'hiver avec une forte densité. Lorsque la valeur de l'indice de surface foliaire (LAI) dépasse **0.9**, CERES produit également un fichier restart. FarmSim transforme ensuite ce fichier restart en fichier d'entrée pour PaSim qui peut reprendre la simulation de la prairie.

3.2.1.2 D'une prairie à une culture

PaSim simule la prairie puis produit à la fin un fichier restart. Ce fichier restart est ensuite transformé par FarmSim en fichier d'entrée pour CERES.

FIGURE 3.5 – Rotation prairie-culture

3.2.1.3 D'une culture à une prairie

CERES simule la culture puis génère plusieurs fichiers de sorties. Ensuite FarmSim traite ces fichiers de sorties afin de générer un nouveau fichier d'entrée de CERES pour la simulation de la période de labour. Une fois que l'indice de surface foliaire (LAI) dépasse **0.9**, CERES produit un fichier restart qui est de nouveau traiter par FarmSim pour générer un fichier restart d'entrée pour PaSim. PaSim peut ensuite effectuer la simulation de la prairie.

3.2.1.4 D'une culture à une culture

Ce cas est déjà géré dans CERES. Il suffit de renseigner les différentes cultures à simuler dans le fichier management de CERES. Un exemple de fichier management configuré pour deux cultures figure en annexe B.

3.2.2 Implémentation

La première étape afin de pouvoir implémenter les rotations entre parcelles dans FarmSim était de raisonner en termes de parcelles plutôt qu'en terme d'années. C'est ce qui a été réalisé et présenté dans la partie 3.1.

La deuxième étape consistait à implémenter les algorithmes qui permettraient de générer les fichiers d'entrées des modèles à partir de fichiers de sorties d'autres modèles. Cependant, à l'heure d'écriture du rapport, ces algorithmes ne m'avaient pas encore été fournis. Ainsi, les rotations entre parcelles n'ont jusque là **pas encore été implémentées**. Toutefois, l'architecture ainsi que le flot d'exécution de l'application ont été clairement définis. Des méthodes vides ont été déclarées afin de simuler les algorithmes à venir. Une fois ceux-ci en main, l'implémentation devrait se faire aisément.

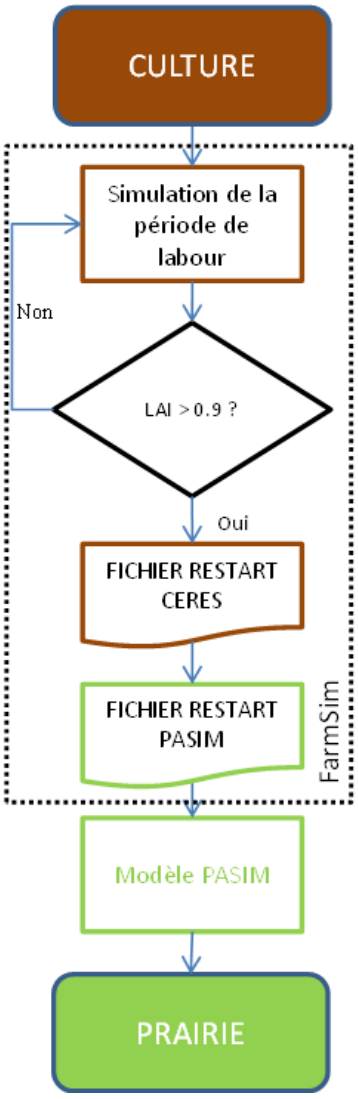


FIGURE 3.6 – Rotation culture-prairie

FIGURE 3.7 – Rotation culture-culture

4
Lancement des simulations spatialisées

Afin de pouvoir réaliser des simulations à plus grandes échelles, l'UREP a entrepris de concevoir une plateforme permettant le lancement de simulations spatialisées. Je devais utiliser cette dernière pour lancer FarmSim à l'échelle de l'Europe. Pour diverses raisons, cette dernière n'était pas fonctionnelle à la fin de mon stage. On me proposa donc de participer activement à son élaboration via un CDD de 6 mois et ce afin de pouvoir lancer les simulations comme prévu initialement.

Afin de réaliser divers tests sur cette plateforme, j'ai développé un modèle basique qui se contente de déterminer la température moyenne à partir des températures minimales et maximales.

4.1 Analyse fonctionnelle de la plateforme

Le diagramme de cas d'utilisation de la figure 4.1 résume toutes les opérations qu'un utilisateur doit être capable de réaliser sur la plateforme.

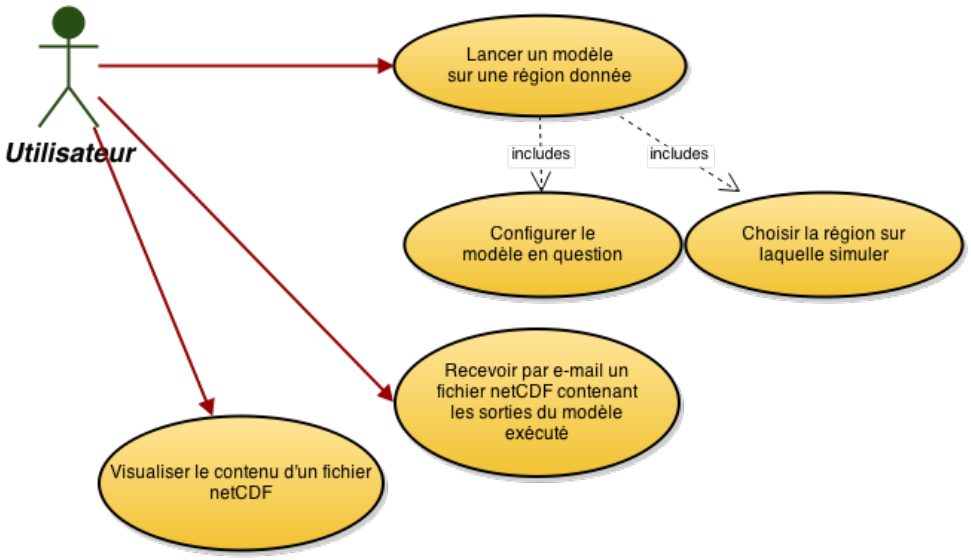


FIGURE 4.1 – Diagramme des cas d'utilisation de la plateforme

4.2 Outils utilisés pour le développement

4.2.1 Qt

Qt est une bibliothèque logicielle multi-plateforme développée en C++ par **Qt Development Frameworks**, entreprise d'informatique norvégienne filiale de **Digia** [4]. Elle peut être utilisée sous licence libre ou commerciale. Elle offre des composants d'interface graphique (widgets), d'accès aux données, de connexions réseaux, d'analyse XML, etc. . . [1]

Qt est fournie avec un environnement de développement intégré (IDE) : **Qt Creator**, dont une capture d'écran est présentée sur la figure 4.2. Cet IDE offre plusieurs fonctionnalités comme la coloration syntaxique, l'auto-complétion ou encore l'accès direct à la documentation de Qt via son interface.

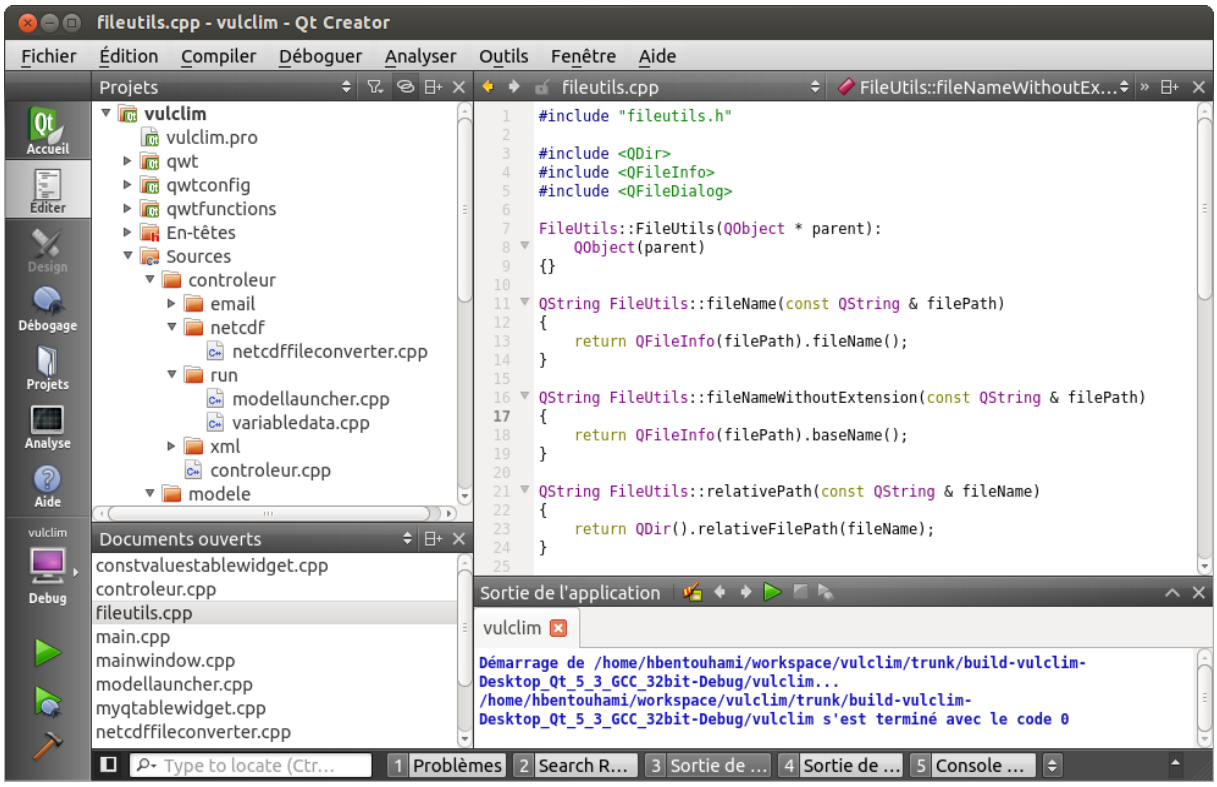


FIGURE 4.2 – Capture d'écran de Qt Creator

4.2.2 NetCDF

4.2.2.1 Présentation

NetCDF est une bibliothèque permettant la création et la manipulation de fichiers au format netCDF ; format de fichier indépendant de toute plateforme et très répandu dans le domaine scientifique. Il permet de stocker des données sous forme de tableaux connexes et est essentiellement basé sur **trois** concepts :

– **Les variables** : grandeurs physiques mesurées (température, précipitation)

1. Source Wikipédia

– **Les attributs** : informations supplémentaires des variables (unités)
– **Les dimensions** : dimensions des variables (temps, longitude, latitude)

Un fichier netCDF est encodé en binaire. Cela lui permet de stocker de grosses quantités de données tout en améliorant les performances d'accès en lecture. On peut toutefois, via des utilitaires fournis avec la bibliothèque NetCDF, exporter un fichier netCDF en fichier texte.

4.2.2.2 Fichiers manipulés par la plateforme

Les fichiers contenant les données en entrée pour les modèles à simuler sur la plateforme seront tous au format netCDF. Pour chaque point de l'espace (repéré par sa longitude et sa latitude) sur lequel on veut lancer une simulation, on doit disposer, pour chaque jour de la période à simuler, la valeur de toutes les variables nécessaires à la simulation.

Deux choix se présentent dès lors pour le contenu des fichiers netCDF :
– soit on stocke les données pour tous les points de l'espace (les valeurs des points dans la mer étant mises à zéro), auquel cas on obtient un fichier **lon-lat** (longitude-latitude)
– soit on stocke uniquement les données pour les points hors de la mer (points terre), auquel cas on obtient un fichier **NPTS** (nombre de points terres)

4.2.2.2.1 Fichiers lon-lat

Les variables stockées dans ce type de fichiers ont **trois** dimensions : le **temps**, la **latitude** et la **longitude** comme le montre la figure 4.3 dans laquelle est représentée l'entête d'un fichier de radiation sur cinq ans.

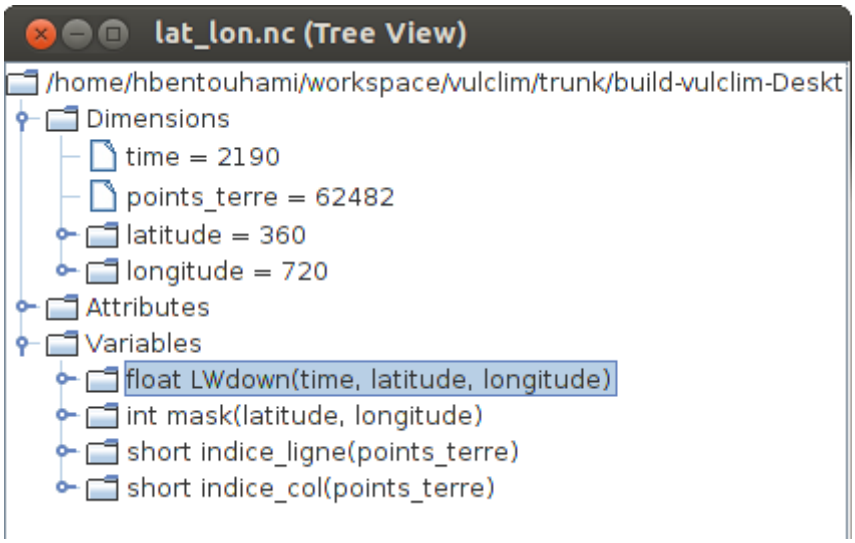


FIGURE 4.3 – Exemple d'entête d'un fichier lon-lat

On pourra remarquer dans cette figure que les dimensions de la latitude et de la longitude sont respectivement de 360 et 720. Cela est dû à la région couverte par les fichiers netCDF et à la taille de la **maille**. Cette maille permet d'indiquer en termes de **degrés**, la distance minimum qui sépare deux points de l'espace. Ainsi avec une maille de **0.5x0.5** (0.5° pour la latitude et 0.5° pour la longitude) et des fichiers de données mondiaux, la latitude dont

la valeur varie de -90 à +90 se retrouve bien avec 360 valeurs. De même avec la longitude qui varie de -180 à +180 et qui se retrouve à 720 valeurs.

4.2.2.2.2 Fichiers NPTS

Afin de stocker uniquement les données des points terres, deux variables supplémentaires (en plus de la variable représentant la quantité physique mesurée) sont utilisées :
– **Les indices lignes** : représentations des latitudes des points terres
– **Les indices colonnes** : représentations des longitudes des points terres

Ces deux variables ont comme unique dimension, les **points terres**. La variable mesurée quant à elle a deux dimensions : le **temps** et les **points terres**. La figure 4.4 montre l'entête du fichier NPTS correspondant au fichier lon-lat de radiation du paragraphe précédent.

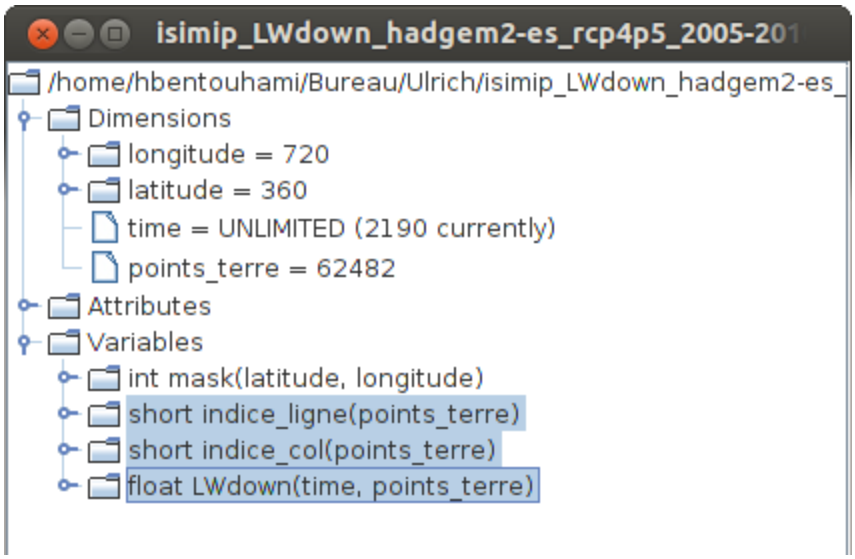


FIGURE 4.4 – Exemple d'entête d'un fichier npts

A l'heure actuelle, seul ce type de fichier est utilisé. Une fonctionnalité permettant la conversion en fichier lon-lat est toutefois disponible. Ainsi dans la suite de cette partie, tout fichier netCDF sera considéré comme étant au format npts.

4.3 Fonctionnalités développées

4.3.1 Interface dynamique

La plateforme ayant pour objectif de pouvoir lancer un nombre indéterminé de modèles, prévoir à l'avance tous les composants qui devront être présents dans l'interface graphique était impossible. L'interface a donc été conçue de telle sorte à pouvoir s'autogénérer via un fichier de configuration. Ainsi, l'ajout d'un nouveau modèle à la plateforme, en ce qui concerne l'aspect purement graphique, ne nécessitera que l'édition de ce fichier.

4.3.1.1 Contenu du fichier de configuration

J'ai choisi d'utiliser l'XML comme format pour le fichier de configuration, d'une part parce que c'est un format très utilisé et standardisé et d'autre part afin de pouvoir faire usage des composants fournis par Qt pour l'analyse XML.

On retrouve dans ce fichier la liste des modèles avec leurs entrées attendues et leurs sorties souhaitées. Pour le moment la question des sorties des modèles n'a pas encore été traitée ainsi l'on n'entre aucune information comme le montre la figure 4.5.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <models>
3    <modele1>
4      <input>
5        <meteo>
6          <Tamax description="Max air temperature" type="standard"/>
7          <Tamin description="Min air temperature" type="temporal"/>
8        </meteo>
9      </input>
10     <output>
11     </output>
12   </modele1>
13   <modele2>
14     ...
15   </model2>
16 </models>
```

FIGURE 4.5 – Fichier de configuration des modèles

Dans la balise **input** (entrées des modèles), on retrouve les variables nécessaires à l'exécution du modèle. L'attribut **description** sert, comme son nom l'indique, à donner une description de la variable et l'attribut **type** (qui est facultatif) indique comment les valeurs seront entrées pour la variable courante. Ce dernier attribut prend trois valeurs possibles :

- **temporal**, auquel cas les valeurs seront récupérées via des fichiers netCDF dont les variables ont comme dimensions les **points terres** et **le temps**. Il en résultera des valeurs (potentiellement) différentes par pixel (point de l'espace) et par année.
- **spatial**, auquel cas les valeurs seront également récupérées via des fichiers netCDF. Cependant, à la différence du type **temporal**, les variables de ces fichiers auront uniquement comme dimension **les points terres**. Il en résultera donc des valeurs (potentiellement) différentes par pixel mais restant constantes sur les années de simulation.
- **standard**, type regroupant les caractéristiques des deux types précédents avec en plus la possibilité d'entrer des valeurs constantes par année pour tous les pixels. C'est le type par défaut.

Dans le fichier de configuration ci-dessus et qui sera utilisé pour la suite de cette partie, la variable **Tamax** devrait, pour refléter plus fidèlement la réalité, être de type **temporal**. Elle a uniquement été déclarée **standard** à titre illustratif.

4.3.1.2 Construction de l'interface graphique

L'idée pour la construction automatique de l'interface via le fichier de configuration est simple : chaque balise sera représentée par un **QTabWidget** (composant pouvant contenir des fenêtres et les organiser sous forme d'onglets) et les widgets des balises enfants constitueront les onglets des widgets de leur balises parentes.

Étant donné le nombre indéterminé de modèles que la plateforme serait susceptible d'exécuter, rajouter tous ces **QTabWidget** directement dans l'interface graphique aurait comme inconvénient d'encombrer l'utilisateur avec des modèles dont il n'a pas besoin. En effet, le fichier de configuration présenté dans la section précédente n'est uniquement accessible qu'aux développeurs de la plateforme. Ainsi, l'interface ne doit uniquement être générée que pour le modèle nécessaire à l'utilisateur. Aussi, l'utilisateur doit avoir la possibilité de changer de modèle.

Pour satisfaire ces contraintes et aussi pour éviter de reconstruire l'interface en repartant de zéro (c'est à dire en parsant une fois encore le fichier de configuration) à chaque fois que l'utilisateur change de modèle, il s'est avéré indispensable de stocker les widgets représentant les modèles.

J'ai donc créé une structure de données sous forme d'arbre. Chaque nœud de cet arbre possède un **lien vertical** et un **lien horizontal**. Le lien vertical permet d'accéder au premier nœud enfant du nœud courant tandis que le lien horizontal permet d'accéder à son nœud « frère ». Les performances en temps d'insertion ou de recherche sont en **O(n)**, avec **n** le nombre de nœuds de l'arbre. La figure 4.6 donne une représentation visuelle de cette structure de données.

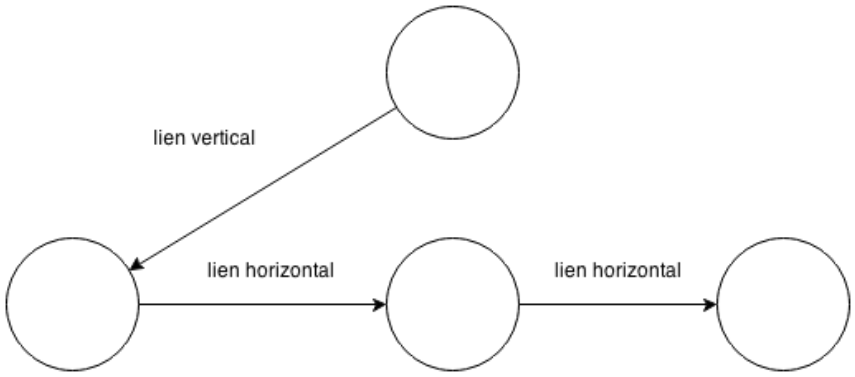


FIGURE 4.6 – Structure de données pour le stockage des widgets

Ainsi, lors de l'analyse du fichier de configuration, les pointeurs des widgets créés sont stockés dans la structure de données. Dans l'onglet **Overview** de l'interface graphique, l'utilisateur a ensuite la possibilité de choisir le modèle qu'il souhaite exécuter. Une fois que le modèle sélectionné change, les onglets de l'interface se mettent à jour avec le contenu du nœud enfant **input** du nœud représentant le modèle. La figure 4.7 montre l'interface graphique correspondant au fichier de configuration de la figure 4.5, le modèle sélectionné étant **modele1**.

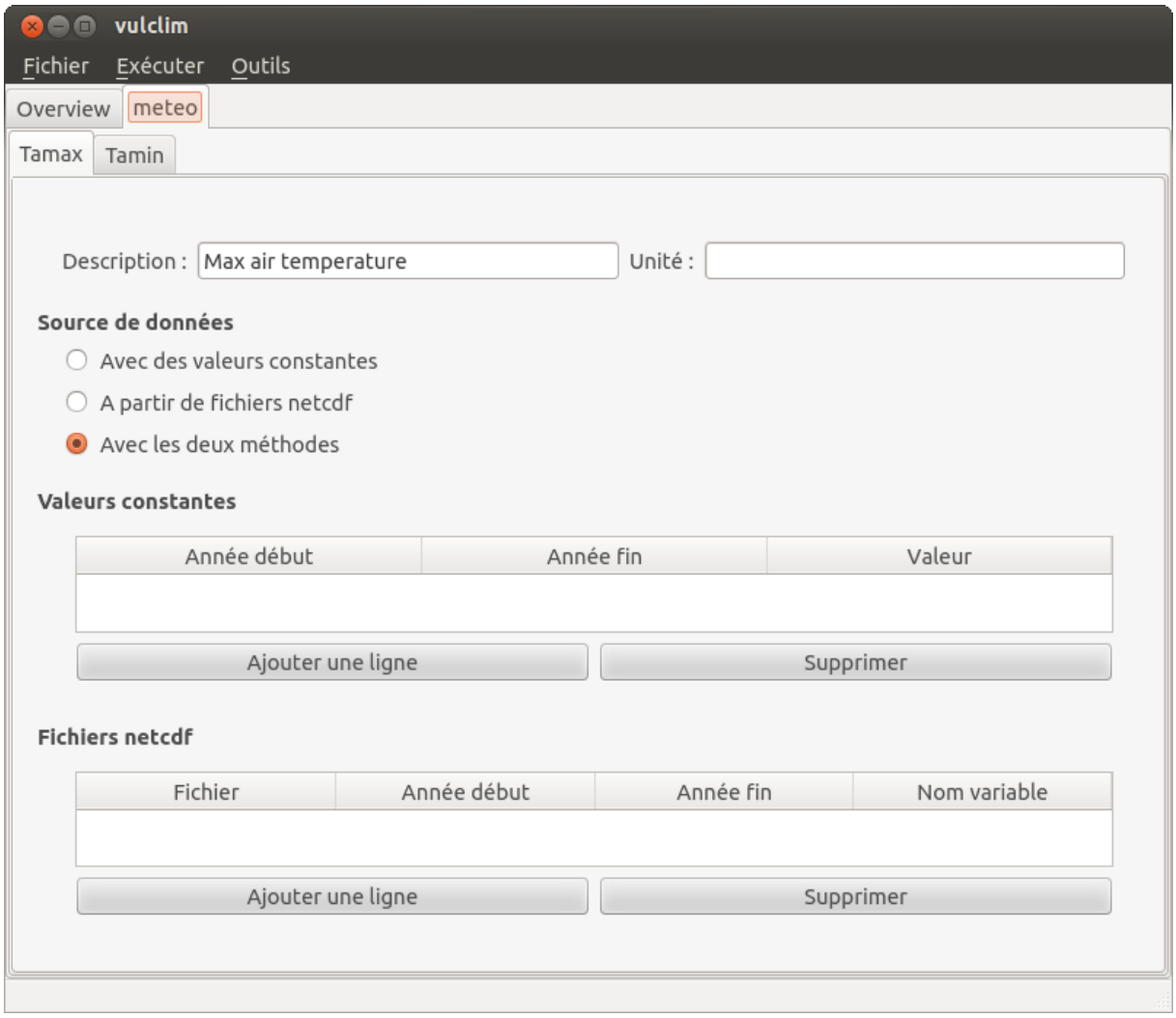


FIGURE 4.7 – Capture d'écran de l'interface graphique

4.3.2 Sauvegarde des données utilisateurs

Selon le modèle à simuler, l'utilisateur pourrait être amené à saisir un grand nombre de données pour plusieurs variables. Cette étape pouvant se révéler fastidieuse, j'ai décidé d'ajouter une fonctionnalité de sauvegarde (et de chargement) de données préalablement saisies dans l'interface.

La mise en place de cette sauvegarde a été grandement facilitée du fait de l'utilisation de la structure de données présentée dans le paragraphe précédent. Il fallait juste parcourir cette structure de sorte à pouvoir respecter l'imbrication des balises pères et enfants telles que définies dans le fichier de configuration. Pour cela, j'ai réaliser un parcours par lien vertical puis par lien horizontal comme le montre l'algorithme 2.

```
Initialisation d'une pile;
Initialisation d'une variable de statut : fin à faux;
Initialisation d'une variable de parcours : cour au lien vertical du noeud racine;
tant que non fin faire
    tant que cour est different du pointeur NULL faire
        Traitement de cour;
        Empiler cour;
        cour = lien vertical de cour;
    fin
    si la pile est non vide alors
        cour = depiler la pile;
        cour = lien horizontal de cour;
    sinon
        fin = vrai;
    fin
fin
```

Algorithme 2: Algorithme de parcours par lien vertical puis lien horizontal

En plus des données des widgets contenus dans l'arbre, il fallait également stocker les données contenues dans l'onglet **Overview**, celles-ci étant indispensables à l'exécution de la simulation. En effet, c'est dans cet onglet que l'utilisateur spécifie les années de simulation, la taille de la maille (paragraphe 4.2.2.2.1), la zone géographique, le modèle à exécuter ainsi que ses informations personnelles (pour la notification par email à la fin de la simulation). Ainsi dans le fichier de sauvegarde (nommé **description.xml**), on retrouve, en plus des balises du fichier de configuration, une balise Overview qui stocke les informations de l'onglet Overview.

4.3.3 Envoi d'emails

A la fin d'une simulation sur la plateforme, l'utilisateur doit recevoir un email de notification lui indiquant les liens vers ses fichiers résultats. L'envoi d'email nécessite l'utilisation d'un serveur SMTP. Ne disposant pas d'informations relatives au serveur qui sera utilisé par la plateforme, j'ai utilisé, durant la phase de test, le serveur SMTP de gmail (après avoir créé un compte utilisateur fictif). Ces informations seront modifiées par la suite lorsque la plateforme sera disponible en mode production.

J'ai ensuite cherché s'il existait des bibliothèques existantes pour l'envoi des mails avec Qt. De cette recherche est sortie **deux** librairies :

- **Qxt** : qui permet de rajouter plusieurs fonctionnalités à Qt. Elle est constituée de **sept** modules [5] dont le module **QxtNetwork** dans lequel se trouve la classe **QxtSmtp** qui implémente le protocole SMTP.
- **SmtpClient for Qt** : qui est une librairie développée uniquement dans le but d'envoyer des mails à l'aide du protocole SMTP.

Ces deux librairies avaient l'avantage d'être libres. J'ai toutefois décidé d'utiliser la seconde car celle-ci était uniquement dédiée à l'envoi de mails à l'inverse de Qxt qui m'aurait offert des fonctionnalités supplémentaires qui ne me sont pas utiles.

4.3.4 Lancement des modèles

4.3.4.1 Récupération des données des fichiers d'entrées

Avant de lancer un modèle il faut d'abord récupérer les données spécifiées dans l'interface graphique. Le principe est d'arriver à stocker dans un vecteur les données pour chaque pixel et pour chaque jour de la période à simuler.

Il faut également noter qu'on peut spécifier l'unité des données d'une variable. Cette information est utilisée afin de réaliser la conversion adéquate lors de la création des fichiers d'entrée des modèles à exécuter.

4.3.4.2 Création du masque de la région à simuler

Une fois les données récupérées, il faut déterminer la région à simuler ou en d'autres termes trouver les pixels pour lesquels le modèle devra être exécuté. Cela passe par la création d'un masque. Ce masque contiendra pour chaque pixel du monde **0** ou **1**. **0** indiquera que le pixel n'est pas à considérer et **1** le contraire.

Ce masque global résulte de l'application de l'opérateur **ET Exclusif** (&) entre plusieurs masques intermédiaires. Ces masques intermédiaires sont la combinaison des masques des pixels des fichiers netCDF utilisés et du masque de la région choisie par l'utilisateur (dans l'onglet Overview de l'application). Ainsi, en partant d'un masque initialisé à **1** pour tous les pixels du monde, on se retrouvera au fur et à mesure des opérations de Et Exclusif avec un masque restreint uniquement à la région à simuler.

4.3.4.3 Exécution

Lorsque l'utilisateur finit de saisir ses données dans l'interface, il peut lancer son modèle via l'action **Lancer** du menu **Exécuter**. L'exécution se fait ensuite par pixel en suivant l'algorithme de principe 3.

```
Création d'un fichier netCDF qui va contenir les sorties du modèle;
Création des vecteurs de données pour toutes les variables;
Création du masque global;
pour tous les pixels faire
    si la valeur du masque du pixel courant est à 1 alors
        Récupérer les données pour le pixel courant via les vecteurs de données;
        Créer les fichiers d'entrées pour le modèle;
        Exécuter le modèle pour le pixel courant;
        Agréger les sorties dans le fichier netCDF de sortie;
    fin
fin
Envoyer le lien du fichier netCDF de sortie par mail à l'utilisateur;
```

Algorithme 3: Algorithme de principe pour le lancement d'une simulation spatialisée

A l'heure d'écriture de ce rapport, le traitement des sorties des modèles n'a pas encore été abordé. On se contente d'envoyer par mail le lien vers le(s) fichier(s) résultat(s) du modèle.

4.3.5 Affichage des fichiers NPTS

La dernière fonctionnalité qui a été rajoutée à la plateforme est l'affichage des données des fichiers npts dans l'interface graphique. J'ai pour cela utilisé le code source de mon projet scolaire, au cours duquel j'ai également eu à afficher des fichiers netCDF. Je l'ai ensuite adapté pour les fichiers npts.
La figure 4.8 donne la capture d'écran d'un fichier npts de température. On peut choisir dans l'interface la variable à afficher, le temps ainsi que les valeurs minimum et maximum (qui peuvent être également détectées automatiquement).

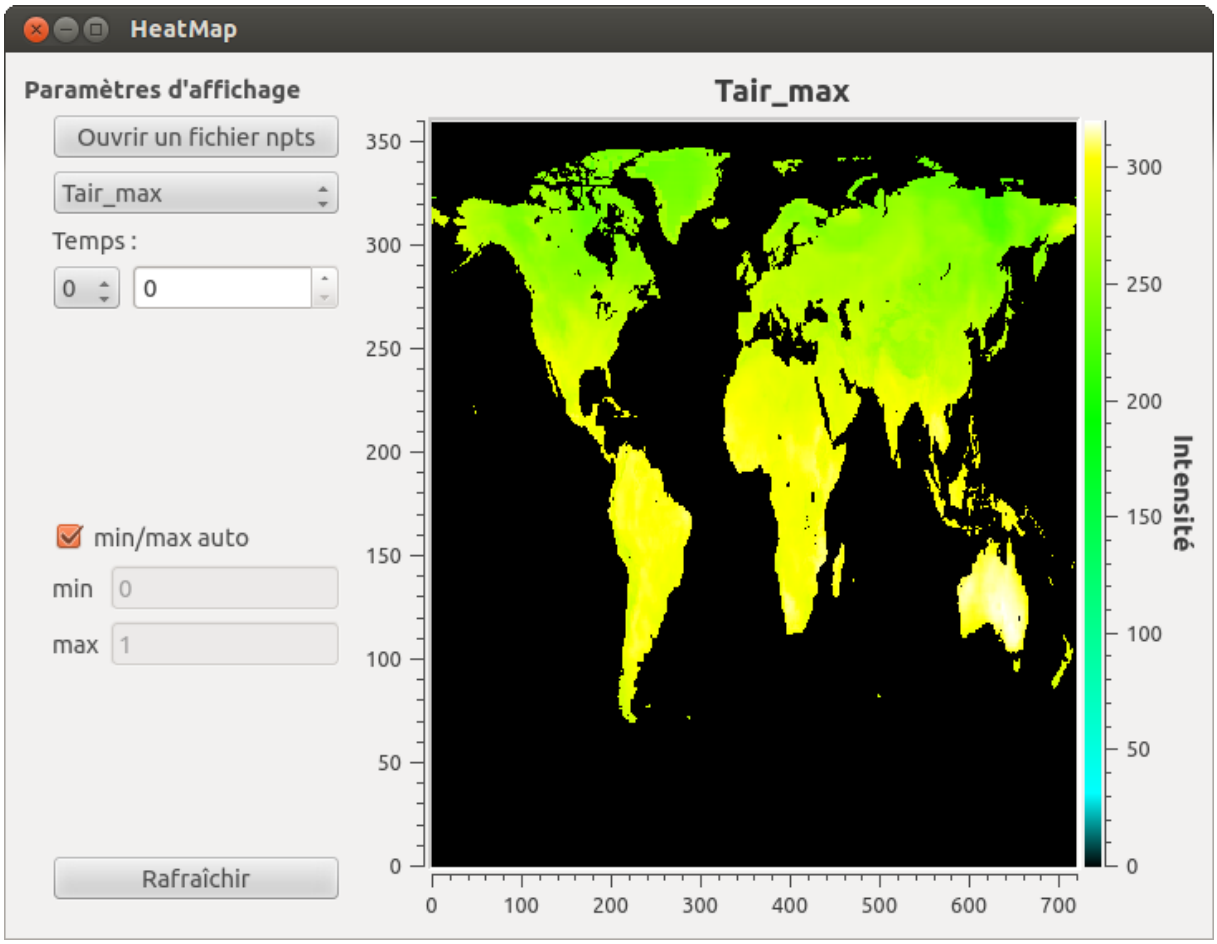


FIGURE 4.8 – Affichage des fichiers npts

4.4 Illustration avec un modèle basique

Comme indiqué en introduction de chapitre, à défaut d'utiliser FarmSim, je vais utiliser un modèle qui détermine la température moyenne à l'aide des températures minimales et maximales. Ce modèle attend des valeurs de température en degrés Celsius (°C) dans un unique fichier d'entrée.

Pour cet exemple, nous allons utiliser les données du fichier **description.xml** de la figure 4.9.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<simDesc>
    <Overview>
        <Years startYear="2008" endYear="2010"/>
        <Mesh size="0.5x0.5"/>
        <Location>
            <coordinates>
                <Latitude min="84" max="84"/>
                <Longitude min="-44.5" max="-44.5"/>
            </coordinates>
        </Location>
        <SelectedModel>modeleTest</SelectedModel>
        <User nom="Ulrich" email="ulsoneza@gmail.com"/>
    </Overview>
    <modeleTest>
        <input>
            <meteo>
                <Tamax unit="K" dataSource="both">
                    <Constant startYear="2008" endYear="2008">273.15</Constant>
                    <File startYear="2009" endYear="2010" variable="Tair_max">
                        isimip_Tair_max_2005-2010.nc
                    </File>
                </Tamax>
                <Tamin unit="K">
                    <File startYear="2008" endYear="2010" variable="Tair_min">
                        isimip_Tair_min_2005-2010.nc
                    </File>
                </Tamin>
            </meteo>
        </input>
        <output/>
    </modeleTest>
</simDesc>
```

FIGURE 4.9 – Fichier description exemple

On peut voir dans ce fichier que la simulation s'effectue sur **trois** ans et pour un **seul** pixel (balise Location). Les données de la température minimale sont récupérées uniquement via la variable **Tair_min** d'un fichier npts tandis que les données de la température maximale sont fournies à l'aide d'une valeur constante (pour l'année 2008) et la variable **Tair_max** d'un fichier npts. Enfin les unités sont spécifiées en **Kelvin** (K).

Une exécution du modèle produit alors la sortie console de la figure 4.10.
Une vérification a été réalisée afin de s'assurer que les données des fichiers netCDF sont bien récupérées en fonction de l'année spécifiée dans l'interface. En ce qui concerne la conversion des valeurs, on peut constater, en affichant par exemple les 3 premières lignes du fichier d'entrée du modèle test, que la valeur constante spécifiée à 273.15 dans le fichier de description, a bien été convertie en °C (K = °C + 273.15).
Aussi le nombre de pixels simulés est bien de **1**.

Enfin, à l'adresse mail spécifiée dans le fichier de description, on retrouve le mail indiquant la fin de la simulation ainsi que le lien vers le fichier résultat. On notera également que le fichier description est attaché au mail en pièce jointe (figure 4.11).

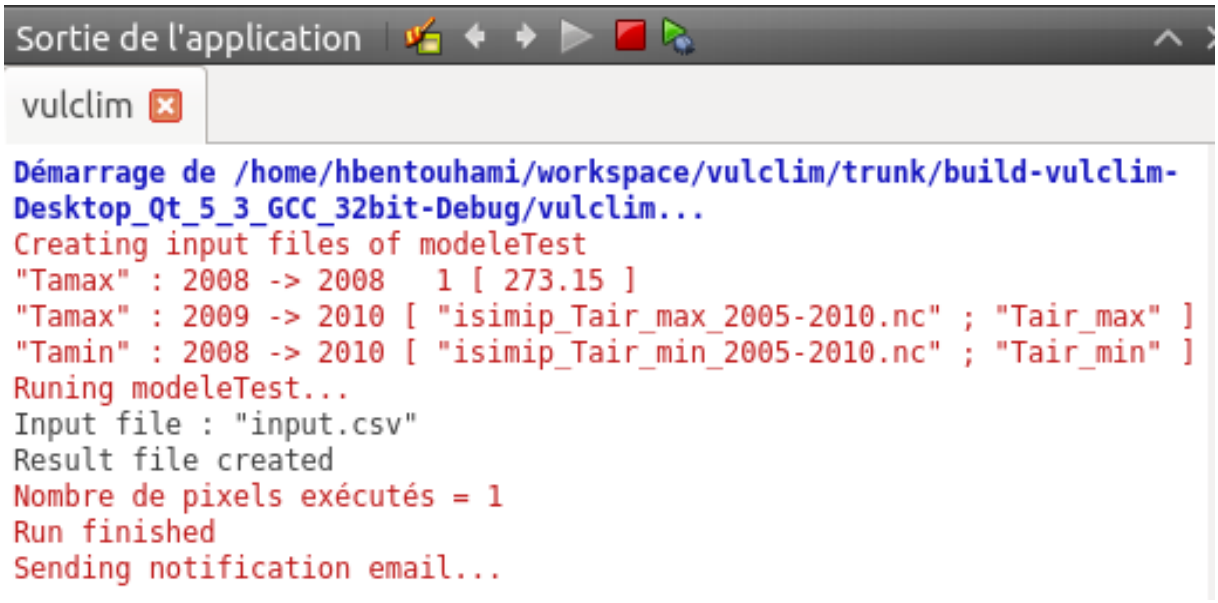
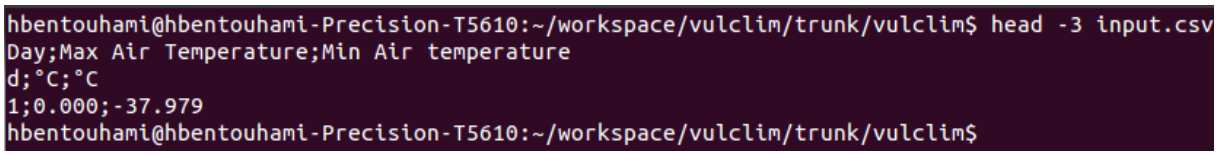


FIGURE 4.10 – Sortie console lancement modèle test



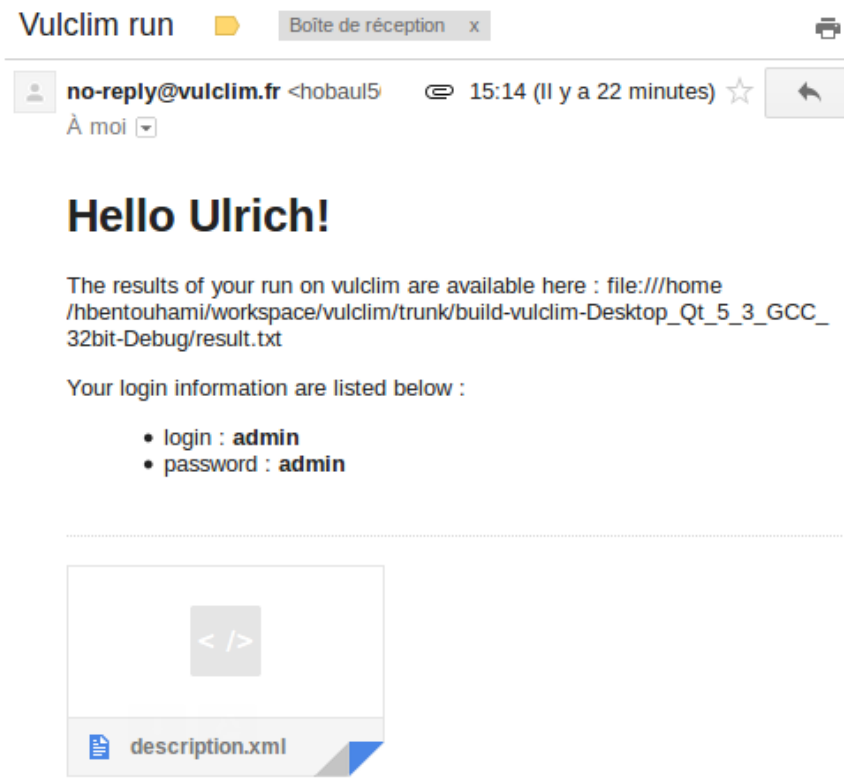


FIGURE 4.11 – Mail reçu à l'issue d'une simulation

Conclusion

FarmSim est un modèle de ferme permettant de simuler une exploitation agricole et d'en réaliser un bilan de GES. Il utilise pour cela deux autres modèles : **PaSim** pour les prairies, **CERES** pour les cultures ainsi que la méthodologie **IPCC** qui est une méthodologie permettant d'avoir des estimations grossières des émissions en N_2O et CH_4.

Ce modèle, avec un historique de 5 ans de développement, présentait une insuffisance majeure : il était impossible de réaliser des rotations prairies/cultures. Ainsi, l'objectif principal de mon stage était de remédier à cette insuffisance puis de lancer des simulations à l'échelle de l'Europe via une plateforme de lancement. Après une phase de correction d'anomalies relevées dans FarmSim, le mécanisme de rotations a été implémenté. Cependant, il n'a pu être mené à terme par manque d'informations concernant les algorithmes à utiliser pour réaliser ces rotations (répercussions sur le sol lors du passage prairie/culture). D'autre part, la plateforme n'étant pas encore opérationnelle, les simulations n'ont pu être réalisées. J'ai plutôt consacrer ce temps au développement de ladite plateforme.

Des améliorations ont toutefois été apportées à FarmSim. Il est désormais possible d'ajouter facilement de nouveaux modèles. L'utilisateur a même la possibilité de choisir les modèles qu'il souhaite utiliser et de ne remplir que les informations nécessaires à l'exécution de ceux-ci. FarmSim a également été intégré à une plateforme d'intégration continue.

Ce stage m'a permis de découvrir de nouveaux outils techniques. Je pense principalement à SonarQube pour l'analyse de la qualité du code source. Mes connaissances de la bibliothèque Qt se sont aussi étendues au module « réseau » pour l'envoi d'emails. J'ai également pu apprendre davantage du serveur d'intégration continue Jenkins. Aussi, j'ai réellement apprécié le fait d'avoir eu à utiliser du Java (pour FarmSim) et du C++ (pour la plateforme) durant le même stage.

La principale difficulté a été la prise en main de FarmSim. En effet, certains rapports des années précédentes présentaient des fonctionnalités que je n'arrivais pas à trouver dans le code source de l'application. Ce dernier n'étant pas versionné, il m'était impossible de déterminer les modifications apportées au fur et a mesure des évolutions du modèle. Le code source était néanmoins bien commenté, ce qui m'a beaucoup aidé.

Dans les développements futurs, il serait intéressant d'améliorer l'expérience utilisateur. En effet, la plupart des onglets de l'interface graphique ne gèrent pas le redimensionnement de l'écran. Les composants y sont attachés en position absolue. Aussi, aucune vérification n'est effectuée sur la validité des données entrées par l'utilisateur. A l'heure actuelle, des valeurs incorrectes entrainent des erreurs de programmation, et à moins d'être développeur, l'utilisateur n'a aucun moyen de déterminer la source de l'erreur.

Bibliographie

[Carozzi et al., 2014] Carozzi, M., Eza, U., Klump, K., Martin, R., and Massad, R. (2014). *Rotations into FarmSim*. INRA.

[Duretz, 2007] Duretz, S. (2007). *Modélisation des transferts d'azote et réalisation de bilans de gaz à effet de serre au sein des exploitations agricoles*.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Graux et al., 2013] Graux, A.-I., Lardy, R., Gaurut, M., Duclos, E., and Klump, K. (2013). *PaSim User's guide*.

Webographie

[1] Documentation swing. docs.oracle.com/javase/7/docs/api/javax/swing/JComponent.html. Consulté le 19/05/2014.

[2] Documentation sourcesup. https://services.renater.fr/sourcesup/index. Consulté le 17/07/2014.

[3] Documentation sonar. http://www.sonarqube.org/. Consulté le 17/07/2014.

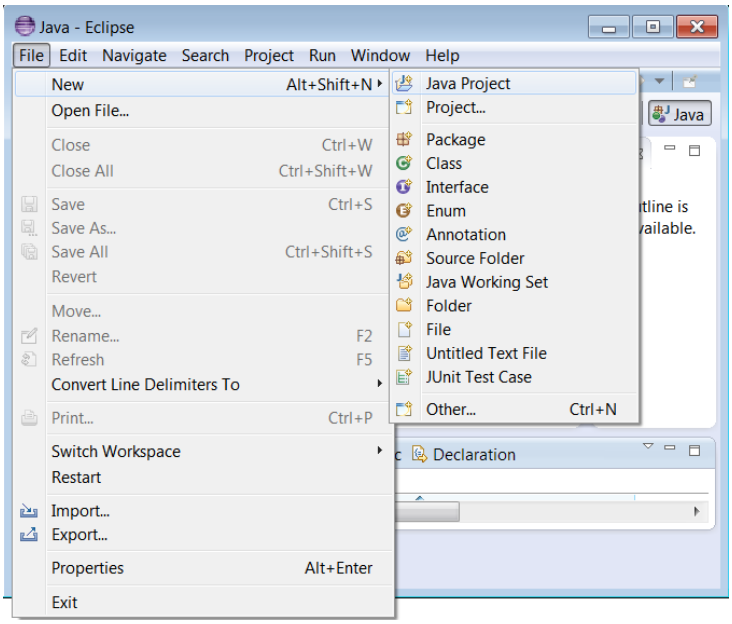
[4] Documentation qt. http://qt-project.org/. Consulté le 11/08/2014.

[5] Documentation qxt. http://libqxt.bitbucket.org/doc/tip/index.html. Consulté le 11/08/2014.

Annexes

A
Import du code source de FarmSim dans Eclipse

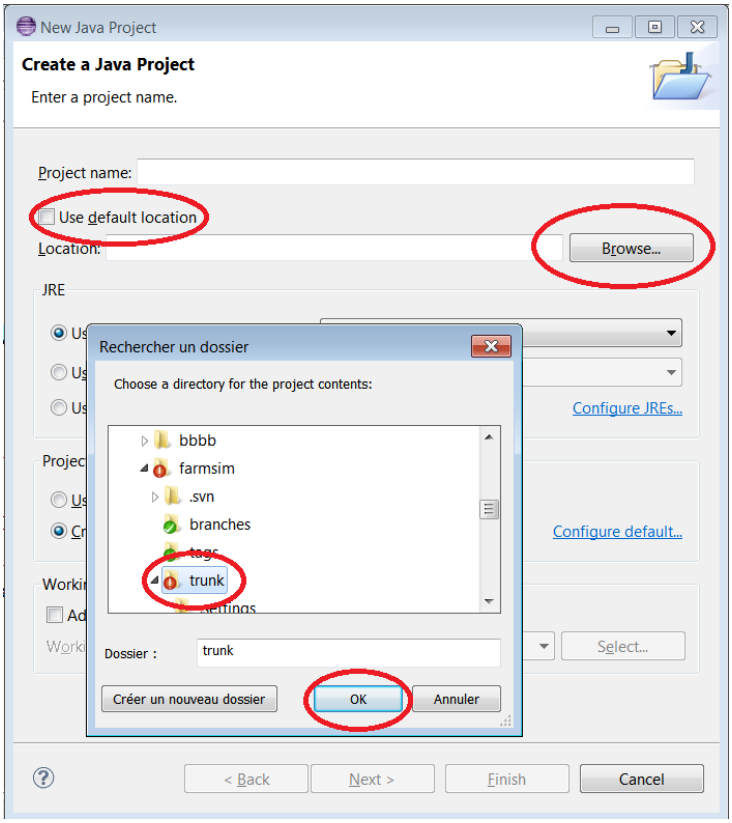
1. Lancer Eclipse
2. Créer un nouveau projet Java

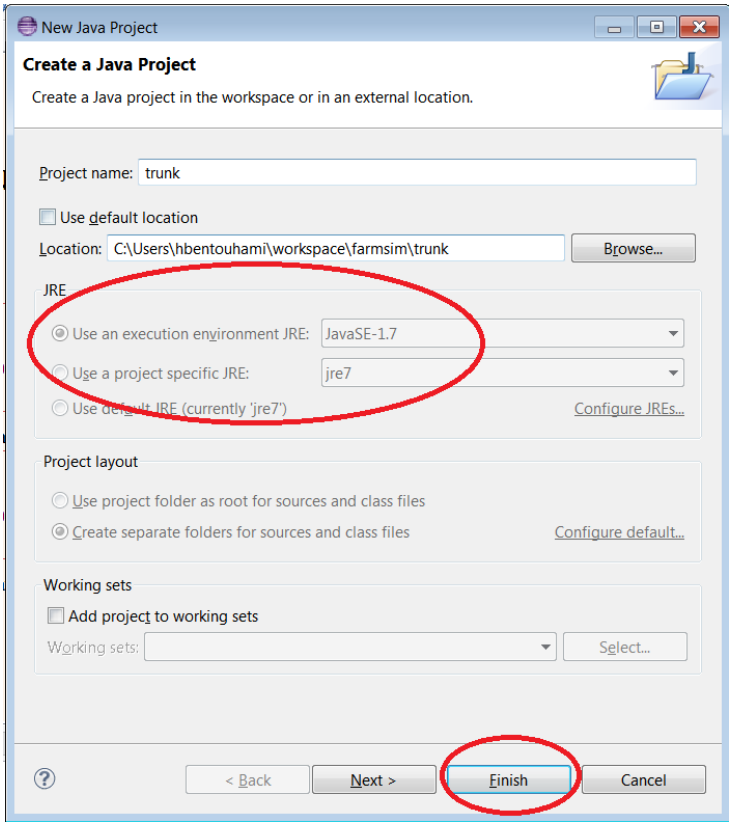


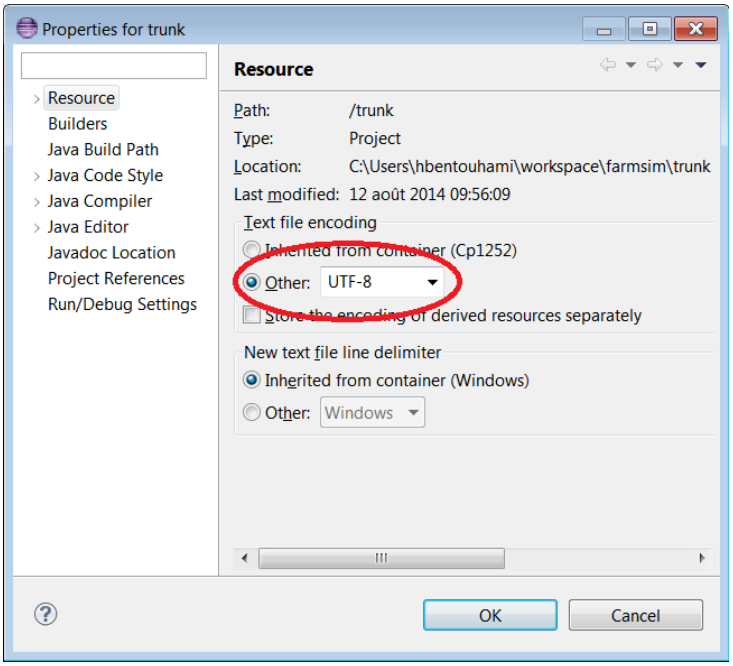
3. Dans la fenêtre suivante, décochez la case **Use default location** et cliquez sur le bouton **Browse** pour aller chercher le répertoire **trunk** de FarmSim puis cliquez sur **OK**.

4. Configurez ensuite la JRE associée au projet. Pour que FarmSim fonctionne il faut au minimum la version **1.7** de Java. Une fois cela fait, cliquez sur **Finish**.

5. Enfin, cliquez dans le menu **Project**, cliquez sur Properties. Une fenêtre s'affiche dans laquelle vous pouvez spécifier l'encodage des sources (qui devrait normalement être à UTF-8).


New Java Project
Create a Java Project
Enter a project name.
Project name:
Use default location
Location:
Browse...
JRE
Configure JREs...
Configure default...
Select...
Rechercher un dossier
Choose a directory for the project contents:
bbbb
farmsim
.svn
branches
tags
trunk
Dossier :
trunk
Créer un nouveau dossier
OK
Annuler
< Back
Next >
Finish
Cancel


New Java Project
Create a Java Project
Create a Java project in the workspace or in an external location.
Project name: trunk
Use default location
Location: C:\Users\hbentouhami\workspace\farmsim\trunk
Browse...
JRE
Use an execution environment JRE: JavaSE-1.7
Use a project specific JRE: jre7
Configure JREs...
Project layout
Use project folder as root for sources and class files
Create separate folders for sources and class files
Configure default...
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel


Properties for trunk
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Run/Debug Settings
Resource
Path: /trunk
Type: Project
Location: C:\Users\hbentouhami\workspace\farmsim\trunk
Last modified: 12 août 2014 09:56:09
Text file encoding
Inherited from container (Cp1252)
Other: UTF-8
Store the encoding of derived resources separately
New text file line delimiter
Inherited from container (Windows)
Other: Windows
OK
Cancel

B
Exemple d'un fichier de configuration de CERES avec rotations

```
Rapeseed test site, Grignon, 2009
   226   1998         (Start of simulation: Day and Year)
 48.50      1         (Latitude and outputs frequency in days)
 60.00  10.10  12.60  (Runoff Coef, T avg. yearly, Thermic Amplitude)
Residues management (kg/ha, used if Switch5 not -1.0)
2340.00  25.00 045.6 0001.00 020.00        (Straws Prof C/N Roots C/N)
Switches  0.00   1.10   1.00   1.10   0.00 (AutoIrr InSoil IniN PreFlow Residues)
Initial Conditions
 Thick  Moist   NH4    NO3     ( cm   vol/vol  mg N/kg soil )
 15.00  0.260   0.50  04.50
 15.00  0.260   1.40  04.50
 15.00  0.249   1.10  00.70
 15.00  0.249   1.10  00.70
 15.00  0.297   0.90  01.20
 15.00  0.297   0.90  01.20
 15.00  0.338   0.30  02.90
 15.00  0.338   0.30  02.90
0000
Rapeseed with agronomically-sensible N dose
RAPESEED                 Crop Name
ROCHE                 Cultivar
Sowing     237 Density 010.0 Depth   4.00
0000                    Irrigations (Day Amount /mm)
000                     Fertilisations N (Day Amount /kg N.ha-1 Depth /cm Type)
063     100.0  5.      6
074     060.0  5.      6
000
368 1 1                  Residue management (tillage day / tillage type / straw removal or incorporation)
00
NEXT
Bare soil period after harvest of rapeseed
FALLOW                       Crop Name
Capitole                     Cultivar
Sowing     001 Density 000.1 Depth   4.00
0000                    Irrigations (Day Amount /mm)
000                     Fertilisations N (Day Amount /kg N.ha-1 Depth /cm Type)
000
368 1 1                  Residue management (tillage day / tillage type / straw removal or incorporation)
00
ROTA
```