SYSTÈME D'ÉVALUATION ET DE GESTION
DES RISQUES D'INONDATION EN MILIEU FLUVIAL

PROJET SEGRI

RAPPORT DE RECHERCHE 2005

Rapport de recherche No R-720-C *Janvier 2006*

Système d'**É**valuation et de **G**estion des **R**isques d'**I**nondation en milieu fluvial

Projet SEGRI

Rapports de recherche 2005

Présenté au

Fonds des Priorités Gouvernementales en Science et en Technologie – volet Environnement (FPGST-E)

13 janvier 2006

Équipe de réalisation

Institut National de la Recherche Scientifique – Eau, Terre et Environnement

Yves Secretan	Professeur, PhD
Eric Larouche	Ingénieur en informatique

Collaborateurs

Maude Giasson	Étudiante à la maîtrise
Sybil Christen	Stagiaire, informaticienne
Aurélien Mercier	Stagiaire
Thierry Malo	Stagiaire
Olivier Bédard	Stagiaire
Jean-Philippe Lemieux	Stagiaire

Pour les fins de citation : **Secretan Y., Larouche E. & coll. (2005)**.

Système d'Évaluation et de Gestion des Risques d'Inondation en milieu fluvial (SEGRI) : Rapports de recherche 2005. Québec, INRS-Eau, Terre & Environnement. Pagination multiple. (INRS-Eau, Terre & Environnement, rapport de recherche 720 c)

Pour: Fonds des Priorités Gouvernementales en Science et en Technologie – volet Environnement (FPGST-E).

ISBN: 2-89146-319-6

Introduction

Ce document vise à faire état des activités réalisées en 2005 dans le cadre du projet SEGRI. Un résumé des tâches réalisées en cours d'année est tout d'abord présenté, suivi d'un assemblage des rapports rédigés.

Les rapports rédigés sont les suivants.

- Rapport #1 : Module Simulation;

  Ce rapport présente le travail de conception et d'analyse qui fut réalisé sur le module responsable de gérer les simulations dans Modeleur2.

- Rapport #2 : Séries;

  Ce rapport présente les développements réalisés sur les séries. Il fait suite à un premier rapport publié en 2003.

- Rapport #3 : Modèles mathématiques en hydrodynamique fluviale;

  Ce rapport présente les modèles mathématiques représentant l'écoulement au sein des modèles hydrodynamiques.

- Rapport #4 : Banc de test.

  Ce rapport présente le banc de test qui fut mis en place au cours du projet. Il discute de son architecture et de son fonctionnement.

Résumé des travaux – année 2005

Janvier 2005

Arrivée d'un nouveau stagiaire au sein du groupe

Le début de l'année 2004 a salué l'arrivée de Jean-Philippe Lemieux, étudiant en informatique de l'Université de Sherbrooke. M. Lemieux allait se voir confier le mandat traitant du module Simulation.

Note : il est pertinent de souligner qu'il n'est pas facile de recruter des stagiaires à l'hiver ou à l'automne. Ceci explique pourquoi un seul stagiaire s'est joint au groupe à l'hiver 2005, ce qui a évidemment ralenti le développement.

Finalisation des livrables 2004 – version papier

Au tout début de l'année 2005, les livrables 2005 suivants furent révisés suite aux commentaires des différents intervenants:

- rapport d'étape #2;
- spécifications fonctionnelles – version finale.

Une version finale de ces documents a été livrée à la fin du mois de janvier 2005. Ces livrables ont servi pour obtenir l'approbation de financement pour la dernière tranche du projet.

Finalisation des livrables 2004 – version web

Les livrables suivants furent également développés et publiés sur le site web du projet en vue de présenter la version alpha de Modeleur2 :

- guide d'installation;
- guide de démarrage;
- projet de démonstration.

Implantation de PostgreSQL 8.0 accessible de l'extérieur

L'engin de base de données PostgreSQL 8.0 a été installé sur un serveur situé dans la DMZ (**De**-**M**ilitarized **Z**one) de l'INRS-ETE. Les bases de données sur ce serveur peuvent être accédées de l'extérieur. Il n'est donc pas nécessaire d'installer localement PostgreSQL sur la machine où tourne Modeleur2.

Février 2005

Analyse du gestionnaire de données

Au cours des deux premières années du projet, le module de gestion de données a été implanté autour d'une base de données relationnelle (BD relationnelle). Bien que le module était fonctionnel, il a été décidé de revisiter l'aspect gestion de données et de s'assurer de blinder ce module car il est la pierre d'assise du logiciel. Plusieurs aspects ont donc été introduits ou remodelés afin d'intégrer la robustesse requise.

Analyse et design du module Simulation

Le module simulation gère tout ce qui concerne l'activité de simulation. L'utilisateur doit pouvoir construire une simulation et lui ajouter des calculs contenant divers paramètres spécifiques au simulateur utilisé. Les données utilisées en entrée pour les calculs proviennent d'autres entités de Modeleur2 stockées dans la base de données. L'exécution d'un calcul prend place dans un processus externe à Modeleur2 (ex : via l'exécutable H2D2). Les résultats d'un calcul sont ensuite importés dans la base de données et liés aux données sources, desquelles ils dépendent. Cette première phase a permis d'identifier les grands acteurs et de mettre en place les scénarios d'usage via des diagrammes de séquence.

H2D2 - Parallélisation et stratégie pour le partitionnement du maillage

En vue de supporter la parallélisation, une stratégie de partitionnement du maillage avait été développée en 2004. Cette stratégie visait à assigner à chaque processus une partie du maillage et de gérer les échanges d'information nécessaires entre les processus. Les structures internes doivent travailler dans une numérotation locale mais être capable de se positionner dans le maillage global. L'objectif est de permettre à chaque processus d'effectuer un travail local, mais qu'il soit possible d'ensuite recombiner le travail fait par chacun, comme si le travail avait été fait sur un maillage global. Pour réaliser ce travail, le partitionneur ParMETIS (http://www-users.cs.umn.edu/ ~karypis/metis/parmetis/) et la librairie de calcul parallèle PETSc (http://www-unix.mcs.anl.gov/petsc/petsc-2/) furent utilisées.

Mars 2005

Implantation du module de simulation

Le diagramme des classes a été produit et une première ébauche du module de simulation a été implantée à partir de ce celui-ci.

Recrutement d'un stagiaire technicien

Xavier Lavoie, un stagiaire du Cégep Lévis-Lauzon en informatique industrielle fut recruté pour un stage de 8 semaines. Son mandat allait être de standardiser les messages d'erreur dans le logiciel et d'aider à certaines tâches de programmation.

Révision du module Validation de données

Le module validation de données a été scindé. La partie qui était liée à la topographie a été migrée dans un module distinct. Le module validation a été revu pour s'intégrer le plus fidèlement possible à l'architecture respectée par les autres modules.

H2D2 : implantation de l'élément de St-Venant

Le code implantant l'élément de St-Venant a été écrit. Il doit toutefois être testé. Une comparaison serrée entre Hydrosim et H2D2 reste donc à faire, tant au niveau des taux de convergences que des résultats.

Avril 2005

Standardisation des messages d'erreur

Tous les messages d'erreur ont été répertoriés et standardisés de façon à ce qu'il soit plus facile d'effectuer la tâche de traduction.

Rapport sur le module de simulation

Un rapport résumant les principaux sujets reliés au dossier module de simulation a été rédigé.

Révision du module Éditeur

Le module Éditeur a été révisé. Les callbacks ont été remplacés par des itérateurs. L'implantation fut techniquement plus difficile, mais la maintenance est beaucoup simplifiée. Le module éditeur utilise désormais les algorithmes de tracé VTK du module TraceVTK.

Mai 2005

Arrivée d'un stagiaire pour travailler sur un banc de test

Thierry Malo, stagiaire en génie des matériaux 2ème année issu de l'École Polytechnique de Montréal, s'est joint à l'équipe. Son mandat allait être de mettre en place un banc de test.

Arrivée d'un stagiaire pour aider à compléter des modules

Olivier Bédard, stagiaire en informatique 3ème année issus de l'Université Laval, s'est joint à l'équipe. Son mandat allait être d'aider à compléter certains module.

Arrivée d'une stagiaire pour travailler sur les séries

Sybil Christen, étudiante à l'Université de Montréal, s'est joint à l'équipe. Son mandat allait être de travailler sur les séries et développer un constructeur de série.

Extraction de la région d'un maillage

L'extraction de la région d'un maillage (i.e. la peau d'un maillage) a été implantée. Il était requis d'obtenir la région d'un maillage pour pouvoir déterminer la limite de couverture d'un champ éléments finis (pour le MNT).

Échelle de couleur

L'enregistrement en bd des échelles de couleur a été implanté. Les couches des MNT et des partitions de maillage retiennent désormais leur couleur.

Juin 2005

GDType

Les données de Modeleur2 se sont vus qualifiés à l'aide d'un GDType. Plutôt que de manipuler directement une chaîne de caractères on utilise un GDType pour interagir avec les données. Lorsqu'on demande des listes d'entités, on utilise le GDType pour qualifier la liste.

MayaVi

Afin de rendre disponible toute la puissance de VTK, l'outil MayaVi a été intégré à même Modeleur2. Il est désormais possible d'avoir accès à tous les paramètres des acteurs VTK qui sont dans une représentation graphique.

Mise en place d'un banc de test

Un banc de test a été développé. Des scénarios de test ont été développés pour couvrir la plupart des aspects du logiciel. Les résultats des tests sont publiés sur une page web dans le site interne.

Juillet 2005

Choix de connexions ODBC

Les fonctionnalités Nouveau projet et Ouvrir projet ont été modifiées pour permettre à l'usager de sélectionner la connexion ODBC sur laquelle il désire travailler.

Événements de sélection de points

Les événements de sélection sont désormais propres à chaque type de semi. La partie GUI du module topographie reçoit les événements de sélection topo et affiche les attributs des points.

Restauration de la BD de test (banc de test)

Chaque script du banc de test a été modifié pour implanter une restauration complète de la bd au tout début.

Délimitation du contour d'une couche

Les couches de topographie sont désormais limitées par la région du maillage du champ éléments finis qui leur est attribué.

Gestion des dépendances des entités

La sauvegarde d'entités dans la base de données met désormais une table dépendances des entités qui permet de savoir si une entité est essentielle à la présence d'une autre (ex : un maillage est essentiel à la vie d'un champ).

Suppression des entités

La fonctionnalité de suppression des entités à été mise en place. Chaque GDAlgoIo s'est vu implanté une méthode pour effectuer cette tâche. La présence des dépendances est également vérifiée avant de tenter une suppression.

Mise-à-jour du BDTraducteur

Le BDTraducteur qui sert à traduire les projets de Modeleur 1 a été mis à jour pour être compatible avec les changements réalisés depuis le début de l'année 2005.

Août 2005

Modification à l'importateur

L'importateur de semi de points de topographie a été modifié pour tenir compte des modifications dans les tables de la BD.

Recrutement de stagiaires pour l'automne 2005

Aurélien Mercier, étudiant 3ème année en génie informatique de l'école EPITA en France, est venu faire un stage de trois mois. M. Mercier allait se voir confier divers mandat permettant de finaliser la version de Modeleur2 à livrer fin 2005.

Outil de construction des séries : phase I

Un outil a été développé pour construire des séries 2D. L'outil permet de spécifier les coordonnées et les valeurs portées à chaque nœud de la série. Les valeurs portées peuvent être des champs, de même que des scalaires. L'outil n'est relié avec la BD.

Boîte de propriété des points

Chaque semi a sa boîte de dialogue qui présentent les attributs des points. Chaque fois que l'usager sélectionne un point, ses attributs sont affichés dans la boîte de dialogue. Il peut modifier la valeur des points et leur valeur.

Sauvegarde d'un projet

La fonctionnalité de sauvegarde d'un projet a été implantée. Elle provoque la sauvegarde de toutes les entités ouvertes par les modules.

Simplification du design BD pour les séries

Le design des séries dans la base de données a été simplifié, principalement pour qu'il soit possible d'écrire des GDAlgoIo « template ». Les séries 0D, 1D et 2D sont supportées par le nouveau modèle.

Septembre 2005

Extension du contrat de Sybil Christen

Sybil Christen, stagiaire à l'été 2005, a été engagée pour un contrat de travail allant de septembre à décembre 2005. Elle travaillait à partir de sa résidence à Montréal. Ceci a permis de valider et tester le développement à distance et des outils de connectivité.

Sauve sous projet

Une fonctionnalité permettant la sauvegarde sous de projet a été mise en place. Ceci permet de dupliquer le contenu d'un schéma bd dans un autre.

Développement à distance

Un guide fut rédigé lequel détail chacune des étapes pour être en mesure de monter l'environnement de développement sur un ordinateur à distance.

Ajout d'algorithmes BD pour gérer les séries

La façon d'emmagasiner les séries dans la base de données a été simplifiée. Des algorithmes « template » ont pu être construits pour implanter les séries 0D, 1D, 2D et les valeurs portées de type champs scalaires, champs vectoriels, champ ellipse-erreur et de type scalaire.

Zone d'étude

La fonctionnalité de la zone d'étude a été rendue fonctionnelle. Il y a désormais une zone d'étude NIL à utiliser lorsqu'on ne veut pas de zone d'étude. Il y a possibilité de construire des zones d'étude, de les modifier et de les sélectionner lors du choix d'entités géo référencées.

Utilisation de PGStream

Dans le but d'accélérer les accès à la base de données, la librairie PGStream a été mise à l'essai. Contrairement à OTL, qui fonctionne ODBC, cette librairie ne fonctionne que sur PostgreSQL.

Automatisation complète du processus de validation du logiciel

Toutes les étapes du processus du processus de validation du logiciel ont été mises en œuvre. À chaque soir, ont lieu à la chaîne, les étapes d'extraction des fichiers source de CVS, de compilation multi compilateurs, de génération de l'installateur, d'exécution de

l'installateur sur la machine de test, d'exécution des tests et de publication des résultats sur un site web interne.

Octobre 2005

Révision des classes éléments finis

En vue de supporter les séries 0D, 1D, 2D, les classes de maillage éléments finis ont été ajustées.

Outil de construction des séries : phase II

L'outil de construction de séries développé au mois d'août s'est vu intégré les algorithmes BD des séries. L'outil fut testé et amélioré pour répondre aux besoins. Il s'est vu modifié pour pouvoir construire et gérer des séries 0D et 1D.

Boîte de dialogue import/export

Une boîte de dialogue import/export a été développée. Elle permet de sélectionner l'entité à importer/exporter, les fichiers impliqués, les filtres et liens (pour les importations).

Exportation des entités

Les fonctionnalités nécessaires à l'exportation des entités de type champs éléments finis, maillage éléments finis et semi de points de topographie ont été implantées.

Choix d'un outil de CMS

Afin d'avoir une documentation facile à maintenir par une personne non spécialisée, il a été déterminé qu'un outil de gestion du contenu (CMS : Content Management Software) devait être choisi. Plusieurs critères ont mené à la décision. Plusieurs outils ont été considérés. Le choix a été PLOne car il offre beaucoup d'extension, il est gratuit et est supporté par un grande communauté de développeurs.

Utilisation d'un éditeur externe dans PLOne

PLOne offre un éditeur de texte basique pour effectuer l'édition. Toutefois, celui-ci n'est pas du niveau de Word. Il apparaissait donc un besoin clé d'être en mesure d'éditer le texte du site dans un éditeur externe. Un travail a dû être fait pour intégrer un éditeur externe.

Apprentissage de PLOne

Bien que complet et flexible, PLOne présente une courbe d'apprentissage relativement abrupte. Il a été nécessaire d'apprendre PLOne et Zope pour pouvoir utiliser efficacement l'outil et pouvoir monter le site web Modeleur2 (et donc, son manuel de l'usager).

Utilisation d'une librairie XML

Lors de la tâche d'import/export, il a été décidé que le XML allait être utilisé comme langage de balise. Plutôt que réimplanter du code qui fait le traitement XML, XMLParser, une librairie externe a été utilisée à cet effet.

Novembre 2005

Révision de la hiérarchie des champs

Le modèle de données des champs/séries combinait l'héritage multiple, l'héritage virtuel, les template et la publication en Python. Ceci a été modifié car les temps de compilation sur GCC étaient devenus trop longs pour être acceptables.

Opérations sur les champs

Les opérations addition, soustraction, multiplication par un réel, intégration, dérivation ont été introduites afin de pouvoir faire des calculs sur les champs. La publication en Python a également été faite.

Achat de licences pour les compilateurs Intel

Des licences ont été achetées pour les compilateurs Intel C++ 9.0 et Intel Visual Fortran 9.0. La librairie MKL (Math Kernel Library 7.2) a été également achetées.

Outil de sonde

Un outil de sonde a été développé pour être en mesure de suivre la position {x,y} et la valeur en chaque point d'un champ ou d'une série.

Importation des entités

Les fonctionnalités nécessaires à l'importation des entités de type champs éléments finis, maillage éléments finis et semi de points de topographie ont été implantées.

Migration des filtres import/export Hydrosim

Les filtres qui étaient responsables de l'importation des fichiers générés par Hydrosim et de l'exportation de fichiers pour Hydrosim ont été migrés de Modeleur1.

Typage des accès aux entités

Les accès aux entités sont désormais typés. Certaines entités se voient attribuer des accès import/export et c'est seulement ces entités qui peuvent être importées/exportées.

Filtrage des points

Mettre en fonction le filtrage des points à partir de ce qui avait été fait en 2004. Permettre à l'usager de choisir son filtre, et voir le résultat de l'opération de filtrage visuellement (point désactivés à l'écran).

Manuel de l'usager

La rédaction du manuel de l'usager a été complétée. Celui-ci a été rédigé à l'aide de l'outil de CMS PLOne.

Rédaction du rapport final

La rédaction du rapport final a été débutée.

Rédaction du rapport sur la problématique des inondations à Châteauguay

À l'automne de 2004, l'INRS-ETE a été sollicité pour participer à la recherche d'une solution intégrée à la problématique des inondations à Châteauguay. La méthodologie d'analyse et de réduction des risques d'inondation mise en place dans le cadre du projet SEGRI fut appliquée à la problématique des inondations à Châteauguay. Un important rapport faisant état du travail réalisé fut rédigé à la fin de l'année 2005 et finalisé au début de l'année 2006 (Leclerc et al., 2006).

Décembre 2005

Correction de bogues

En vue de livrer la seconde version de Modeleur2, une série de tests ont été effectués et différents bogues furent corrigés.

Module de validation – sauve sous

Une fonctionnalité sauve sous semi a été mise en place, et divers changements ont été apportés au module pour permettre l'édition de semis de points.

Intégration d'outils externes à Modeleur

Boa Constructor (http://boa-constructor.sourceforge.net/) a été intégré à Modeleur2 comme éditeur externe. Winpdb (http://www.digitalpeers.com/pythondebugger/) a été intégré à Modeleur2 comme débuggeur externe.

Mise en place de l'installation

Des ajustements ont été apportés pour que l'installation de Modeleur2 se fasse avec succès avec toutes les composantes qui sont requises. Des tests ont été réalisés sur une machine neutre.

Compilation automatisée sur Intel

Un travail de standardisation du code fut réalisé pour être en mesure de compiler et linker le logiciel sous le compilateur Intel (seul compilateur avec lequel nous pouvons compiler les deux logiciels, Modeleur2 et H2D2).

Corrections aux menus

Les menus ont été revus et ceux qui étaient inutilisés ou qui n'étaient pas encore implantés furent retirés.

Modifications – lecture et écriture des géométries

Étant donné le passage à la base de données PostgreSQL 8.1, il fut nécessaire de revoir les opérations de lecture/écriture des géométries dans la base de données. Ceci fut nécessaire car les géométries sont maintenant stockées de façon binaire, plutôt qu'en tant que chaîne de caractères.

Migration du site web

Le site web du groupe a été partiellement migré sur l'un des deux sites PLone (le site principal, ou le site Modeleur2). Le serveur web Apache redirige les requête vers les sites PLone. Pour le moment, les deux sites coexistent. Éventuellement, tout sera sur un des sites PLone.

Rédaction d'un rapport sur les champs/séries

Un rapport traitant des champs et des séries fut rédigé. Ce rapport se veut en quelque sorte une suite au rapport qui avait été rédigé en 2003. Il reprend là où l'implantation avait été laissée et met en place les concepts qui font que les séries sont aujourd'hui utilisables pour effectuer des calculs.

Rédaction d'un rapport sur les modèles mathématiques en hydrodynamique fluviale

Un rapport intermédiaire discutant de l'étude des modèles mathématiques en hydrodynamique fluviale fut rédigé dans le cadre du mandat d'implantation d'un modèle d'élément 2D ½.

Rédaction du rapport final

La rédaction du rapport final a été finalisée. Le document discute de la composante scientifique, de même que de la composante logicielle.

RAPPORT # 1 : Module Simulation

Rapport de développement logiciel

Module de simulation

Présenté par :

Jean-Philippe Lemieux

13 janvier 2006

Versions du document

Date	Auteur	Commentaires	Version
2005-04-14	Jean-Philippe Lemieux		V1.0

TABLE DES ILLUSTRATIONS

1. Introduction

1.1 Objectifs

Ce rapport a pour objectif d'introduire le module de simulation. Les principaux blocs formant ce module y sont expliqués, sans toutefois entrer dans les détails du processus de développement. Ce rapport a aussi comme objectif de relever les parties incomplètes et les faiblesses à améliorer pour parfaire ce module.

1.2 Contexte

Le module simulation gère tout ce qui concerne l'activité de simulation. Au cœur du module, on retrouve les calculs, qui portent l'information servant aux calculs d'une simulation. Ces calculs forment une simulation.

L'utilisateur peut construire une simulation et lui ajouter des calculs contenant divers paramètres spécifiques au simulateur utilisé. Les données utilisées en entrée pour les calculs proviennent principalement des MNT et autres entités de Modeleur2 stockées dans la base de données. Ensuite, on peut commander l'exécution d'un calcul. L'exécution prend alors place dans un processus externe à Modeleur2. Les résultats d'un calcul sont ensuite importés dans la base de données et liés aux données sources, desquelles ils dépendent.

Les résultats d'un calcul peuvent être utilisés comme point de départ d'un nouveau calcul. Dans la fenêtre graphique de la simulation, on pourra également consulter l'historique des calculs de ladite simulation, sous forme de graphique représentant le niveau de convergence obtenu à chaque calcul ou tout autre critère pouvant être obtenu du simulateur.

1.3 Structure

En premier lieu, ce rapport couvrira les choix d'implémentation qui ont été pris par l'équipe lors de l'étape d'analyse du projet et les relations entre le module de simulation et les autres modules composant Modeleur2, existants et à venir.

Ensuite, il sera question de la conception du module, en insistant sur le découpage des fonctionnalités en unités logiques. Cette section couvrira la description des classes représentant les composantes de l'activité simulation.

Puis, il y aura une description des principaux scénarios d'usage appuyée par des diagrammes de séquence en annexe.

Ce rapport se terminera sur une description de l'état actuel de l'implémentation du module de simulation, des fonctionnalités restant à implémenter et des développements souhaitables à venir.

1.4 Références

[1] Spécifications fonctionnelles – Modeleur2, section 4.8
\\Caxipi\green\Spécifications fonctionnelles\Spécifications - Modeleur2 - v1.69 - OFFICIELLE.doc

[2] Rapport de réunion : Simulation – Réunion 1, *2005-02-10*
\\Caxipi\green\Rapports\Simulation\Résumé de réunion 1 - Simulation - 2005-02-10 - v3.doc

[3] Rapport de réunion : Simulation – Réunion 2, 2005-02-17
\\Caxipi\green\Rapports\Simulation\Résumé de réunion 2 - Simulation - 2005-02-17 - v1.doc

[4] Rapport de réunion : Simulation – Réunion 3, 2005-03-08
\\Caxipi\green\Rapports\Simulation\Résumé de réunion 3 - Simulation - 2005-03-08 - v1.1.doc

1.5 Vocabulaire

Calcul : Unité de représentation d'une simulation. Une simulation est formée d'un ou plusieurs calculs. Abstraction utilisée pour spécifier les paramètres et opérations d'une simulation.

Ensemble simulateur : Extension comprenant le simulateur et les composantes nécessaires pour satisfaire aux exigences du plugiciel simulation. Cet ensemble doit contenir :

- les filtres (importateurs et exportateurs),
- la partie GUI du simulateur,
- la dll simulateur fournissant les GDAlgoIo spécialisés et
- les classes spécialisées notamment SNCalculXXXX.

Plugiciel simulation : Module d'extension (*plugin*) de Modeleur2 qui lui ajoute la capacité d'interfacer avec des simulateurs indépendants et de gérer des simulations complexes.

Simulateur : Application autonome qui effectue des calculs à partir d'une série de données en entrée et qui retourne une série de résultats.

Simulation : Ensemble de calculs. On utilisera plugiciel simulation pour désigner l'extension de Modeleur2 qui permet de gérer une ou plusieurs simulations.

2 Simulation et Modeleur2

2.1 Simulation

Une simulation dans le cadre de Modeleur2 est une suite d'étapes de calcul qui a pour but d'obtenir un résultat final satisfaisant, convergent. Le but recherché pour Modeleur2 est de pouvoir accommoder de multiples simulateurs à l'aide du même et unique plugiciel de simulation.

Un simulateur est une application qui s'exécute de manière indépendante dans une console externe. Il prend en entrée, une série de paramètres et produit une série de résultats en sortie, pour un calcul. La simulation étant une suite de calculs, Modeleur2 se doit de gérer les différents calculs composants une simulation. Une simulation pourra également être effectuée parallèlement sur plusieurs machines ou à distance.

2.1.1 Gestion d'une simulation

Avant de lancer une simulation, il y a des étapes à réaliser afin de bien établir les paramètres propres à la simulation. Ceci découlant du fait que Modeleur2 doit pouvoir accommoder plusieurs simulateurs différents qui peuvent exécuter une gamme de calculs distincts qui leur sont propres.

2.1.1.1 Spécification des calculs

À cette étape, les calculs sont créés et leurs paramètres déterminés selon la nature du simulateur qui sera utilisé. Les calculs peuvent être indépendants ou chaînés un à la suite de l'autre. Toutefois, l'exécution d'un calcul est présentement unitaire, c'est à dire que l'exécution de chaque calcul doit être demandée individuellement.

Les données en entrée d'un calcul proviennent des entités contenues dans la base de données et les résultats obtenus peuvent être enregistrés à leur tour dans la base de données pour assurer leur persistance. Les résultats importés sont alors liés aux données sources.

2.1.1.2 Navigation de l'historique

Un historique des calculs exécutés comportant une représentation graphique qualitative des résultats obtenus sera disponible pour assister à la gestion de la simulation. Cet historique sera évidemment dépendant du simulateur utilisé et de l'information que celui-ci peut retourner pour aider l'usager à atteindre un résultat satisfaisant. On peut par contre penser à un historique montrant le degré de convergence obtenu à la fin de chacun des calculs par exemple.

2.1.1.3 Modification des paramètres de calcul

La suite de calculs d'une simulation n'a pas à être linéaire, et grâce aux informations recueillies dans l'historique l'usager aura le loisir de modifier les paramètres des calculs à être exécutés afin de parvenir à une solution satisfaisante. Il sera aussi possible d'altérer

les résultats d'un calcul réalisé afin d'améliorer la qualité du résultat d'un calcul subséquent.

2.1.2 Exécution d'un calcul

Lors de l'exécution d'un calcul, les étapes suivantes sont effectuées :

2.1.2.1 Exportation des données

Cette étape consiste à rendre disponibles les paramètres d'entrée nécessaires aux calculs à effectuer pour qu'ils soient utilisés par le simulateur. Cette action doit également convertir les données dans un format compréhensible pour le simulateur. Ceci peut être accompli en utilisant des filtres. (Voir 3.1.4. Filtres d'importation et d'exportation)

2.1.2.2 Construction d'un fichier de commandes

À cette étape, la suite des commandes pour lancer le calcul à exécuter est déterminée et écrite dans un fichier de commandes. Pour ce faire, il y aura fusion de deux ensembles de configurations soient au niveau du calcul pour les paramètres propres aux données du calcul et au niveau du plugiciel simulation pour la configuration de l'exécution à distance ou sur un *cluster*.

2.1.2.3 Lancement des calculs

Les calculs sont lancés dans des processus indépendants pour être exécutés en arrière-plan. C'est en fait le fichier de commandes qui est exécuté. L'utilisation de processus différents permet de dissocier les deux exécutions et Modeleur2 peut ainsi être terminé alors que les calculs sont en cours.

2.1.2.4 Suivi d'avancement

En considération du temps requis par une simulation, il est souhaitable de pouvoir fournir à l'usager un moyen de suivre l'état d'avancement des calculs, puisque ceux-ci peuvent prendre des jours, voire des semaines.

2.1.2.5 Importation des résultats

Une fois les calculs terminés, il reste à importer les résultats obtenus dans la base de données de Modeleur2 pour y être utilisés par l'usager.

2.2 Les principaux acteurs et composantes du module simulation

Les acteurs impliqués dans la mise en œuvre de l'activité simulation sont :

- un ensemble simulateur externe déterminé,
- le module *Gestionnaire BD*,
- le module *Interface graphique*,
- le module *Affichage*,
- le module d'importation et d'exportation à être implémenté ultérieurement,
- et le module *Simulation* lui-même,

Les composantes d'un ensemble simulateur sont définies à la section suivante.

2.2.1 Gestionnaire BD

Le module *Gestionnaire BD* servira les accès à la base de données, soient la sauvegarde et les comptes de références ainsi que le chargement et le déchargement d'objets en mémoire via les GDAlgoIo fournis par le module de simulation et l'ensemble simulateur.

2.2.2 Affichage

Le module *Affichage* se chargera d'afficher l'historique d'une simulation. Cela est fait au moyen d'un objet graphique créé pour utiliser la structure de données de l'historique.

2.2.3 Interface graphique

Le module *Interface graphique* s'occupe quant à lui de la gestion des interactions avec l'usager via le GUI. Il est ici question des menus, de l'arborescence des simulations et de la barre d'outils. La partie GUI du module *Simulation* offrira les services de simulation à l'utilisateur en ajoutant ses menus spécialisés et la composante arborescence à l'application Modeleur2.

2.2.4 Exportation et importation

Le module d'importation et d'exportation servira à répondre aux événements d'exportation des données pour un calcul et d'importation des résultats d'un calcul, générés par le module de simulation. Les requêtes à ce module seront effectuées au moyen des objets IEExport et IEImport qui servent à spécifier les données concernées et la façon de les traiter. Des ébauches de ces classes font présentement partie du module simulation mais devraient dans le futur être intégrées au module d'importation et d'exportation.

2.2.5 Simulation

En ce qui concerne le module simulation, il sera composé d'une partie GUI qui s'attachera au GUI principal du module *Interface graphique* et d'un gestionnaire des simulations dont le rôle est de traiter les événements pertinents au module *Simulation*.

Quant à la partie de gestion des simulations, elle permettra entre autres de gérer plusieurs objets simulation chargés en mémoire. Elle pourra gérer les entrées de menu disponibles après chaque changement de sélection dans la zone de contrôle du GUI.

Le module de simulation contiendra également une interface de programmation (API) pour les calculs des ensembles simulateur sous forme de classe de calcul abstraite dont chaque calcul devra hériter.

3 Conception

3.1 Les composantes d'un ensemble simulateur

Un ensemble simulateur externe à utiliser avec Modeleur2 devra être composé des éléments suivants :

- simulateur (exécutable),
- GUI spécialisé (script Python),
- classe *GDAlgoIoCalcul* et les classes optionnelles *GDAlgoIoHistorique*, *GDAlgoIoObjGraphiqueHist* (compilées dans la dll),
- filtres d'importation et d'exportation (exécutables).

3.1.1 Exécutable du simulateur

L'exécutable du simulateur est l'application autonome appelée pour exécuter les calculs d'une simulation.

3.1.2 GUI spécialisé

La partie GUI spécialisé est un fragment de GUI pour permettre à un usager de gérer les paramètres d'un calcul qui sont dépendants du simulateur utilisé. Cette partie pourrait également générer de façon interactive les commandes qui seront exécutées pour lancer le calcul. Ceci serait exécuté lors de la saisie des paramètres afin de présenter à l'usager les commandes générées par ses choix de paramètres pour ce calcul.

3.1.3 Classes *GDAlgoIoCalcul*, *GDAlgoIoHistorique*, *GDAlgoIoObjetGraphiqueHist*

La classe *GDAlgoIoCalcul* servira quant à elle à spécifier comment un objet calcul spécifique à ce simulateur sera créé, chargé/déchargé de la mémoire, et enregistré dans la base de données. Optionnellement, les *GDAlgoIoHistorique* et *GDAlgoIoObjGraphiqueHist* peuvent être fournis pour spécifier un format d'historique autre que celui par défaut.

3.1.4 Filtres d'importation et d'exportation

Les filtres d'importation et d'exportation sont des exécutables servant à traduire les données au format intermédiaire de Modeleur2 pour être utilisées par le simulateur et retraduire les résultats du calcul au format intermédiaire pour qu'ils puissent être importés dans la base de données. Ces filtres seront ajoutés au module d'importation et d'exportation de manière à les rendre accessibles de façon indépendante du simulateur. Ceci permettra de réutiliser des filtres avec des simulateurs différents sans avoir à maintenir de duplicité. Il sera également possible d'importer des résultats effectués avec un autre simulateur sans la nécessité d'avoir accès au simulateur.

Pour les tâches d'importation et d'exportation, on utilisera des objets IEImport et IEExport grâce auxquels il sera possible de spécifier la donnée à importer ou exporter, le filtre à utiliser pour ce type de donnée, les paramètres d'exécution du filtre le cas échéant et le fichier contenant les résultats à importer ou bien le fichier dans lequel on souhaite exporter les données.

3.2 Ajout d'un ensemble simulateur

Lors de l'ajout d'un ensemble simulateur, un objet représentant le simulateur sera créé et enregistré dans la base de données. Le simulateur sera désormais disponible à être sélectionné lors de la création d'une nouvelle simulation. Les filtres d'importation et d'exportation seront quant à eux ajoutés au module d'importation/exportation et ils y seront accessibles indépendamment du simulateur utilisé.

Lors de la création ou l'ouverture d'une simulation, le nom du fichier du fragment de GUI spécialisé sera envoyé par le bus de transmission au plugiciel Interface graphique où il sera intégré au point d'ancrage prévu. De la même manière, les instances *GDAlgoIoCalcul* seront communiquées au Gestionnaire de BD qui les utilisera pour charger et décharger les objets calcul spécifiques au simulateur.

3.3 Découpage des composantes

Premier aspect d'intérêt pour le découpage des composantes, l'emplacement de l'interface de programmation doit procurer la meilleure souplesse d'utilisation possible. Parmi les paramètres de souplesse d'utilisation recherchés, on note l'accommodation de multiples simulateurs de manière générique ainsi qu'un fort découplage entre les composantes.

3.3.1.1 Modèles d'ancrage

Le modèle d'ancrage utilisé en est un par héritage. Dans le modèle par héritage le simulateur fournit une classe calcul pour construire ses spécifications de calculs. Cette classe hérite de l'interface abstraite calcul du plugiciel simulation qui déclare les méthodes devant être implémentées par l'ensemble simulateur. Ceci a pour effet de créer un calcul spécialisé dès sa construction car les appels se font via l'interface calcul. Ce modèle est représenté à la figure 2.

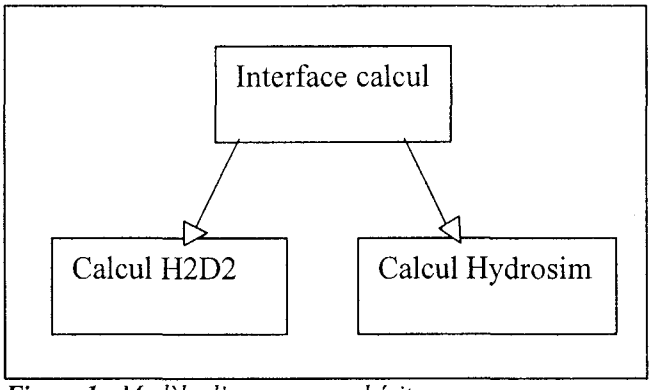


***Figure 1** : Modèle d'ancrage par héritage*

3.4 Revue du diagramme de classes

Le diagramme de classe peut être trouvé en annexe. Ce diagramme représente les différents intervenants de l'activité simulation et leurs relations.

SNGestionnaire : La partie du plugiciel simulation qui est attachée au bus et qui assure toutes les interactions du module simulation qui doivent voyager par le bus. Ceci comprend la réception et l'envoi des événements sur le bus de transmission et le processus d'initialisation du module lors du démarrage de Modeleur2.

De plus, le SNGestionnaire est responsable de gérer les activations et désactivations des entrées de menu du GUI, conjointement avec la partie GUI du Module.

SNSimulateur : Classe représentant un simulateur. Une instance de SNSimulateur servira de *proxy* vers la dll simulateur. Elle recevra et réalisera toutes les interactions avec la dll du simulateur.

SNSimulation : Classe du module simulation pour représenter une simulation. Conteneur de calculs, chaque simulation est associée à un seul simulateur. À chaque simulation, est également associé un objet graphique pour l'affichage de l'historique.

SNCalcul : Classe abstraite représentant un calcul et spécifiant l'interface que doit implémenter la classe calcul d'un simulateur. Cette classe implémente les fonctionnalités pour exécuter et arrêter le calcul. Un objet calcul devra également maintenir une représentation de ses paramètres pour pouvoir fournir les listes d'objets à exporter et importer, le contenu du fichier de commandes utilisé pour l'exécution et pouvoir vérifier la validité de ses paramètres.

SNHistorique : Classe pour représenter l'historique d'une simulation. Il y a un historique associé à chaque calcul et qui sert de structure de données pour toutes les données qu'on souhaite utiliser pour tracer l'historique.

SNObjetGraphiqueHist : Classe héritant de AFObjetGraphique et qui servira afin de recueillir les données contenues dans les historiques des calculs et de les présenter sous forme de graphique dans la fenêtre de la simulation.

SNSelectionSimple : Classe qui représente une sélection dans l'arborescence de la zone de contrôle du GUI. Cette classe est utilisée par les événements pour pouvoir communiquer la sélection actuelle dans l'arborescence et à quel calcul ou simulation se rapporte l'événement.

IEExport et IEImport : Classes servant à spécifier au module Import/export les données à exporter ou importer de même que les filtres à utiliser, le fichier de sortie ou d'entrée et les options du filtre, le cas échéant. Pour permettre les exécutions de calculs à distance, des adresses URL (Uniform Resource Locator) seront utilisées pour identifier les noms de fichiers. La structure exacte de ces objets reste encore à déterminer.

3.5 Interactions entre les objets SNSimulation, SNCalcul et les plugiciels de Modeleur2 attachés au bus.

Il a été choisi d'utiliser le SNGestionnaire comme point d'entrée pour toutes les interactions par événement avec les objets SNSimulation et SNCalcul. Le SNGestionnaire gère alors tous les envois et réceptions de messages via le bus.

De plus, en forçant les interactions entre le GUI et le SNGestionnaire par le bus, on s'assure une indépendance entre ces deux composantes et un moyen pratique d'émuler les interactions générées par le GUI. Finalement, cette orientation ressemble beaucoup a ce qui a déjà été implémenté dans d'autres modules, ce qui donne donc une meilleure homogénéité sur l'ensemble du code de Modeleur2.

Pour que le SNGestionnaire puisse gérer efficacement les interactions du GUI vers les instances représentant une simulation, les références de l'objet SNCalcul auquel l'interaction est destinée et SNSimulation de laquelle le calcul fait partie seront encapsulées dans un objet SNSelectionSimple qui sera transmis au SNGestionnaire lors de la transmission de l'événement. Dans le cas général, ce sera un objet construit par l'explorateur d'arborescence de la partie GUI simulation.

4 Scénarios d'usage

4.1 Revue des diagrammes de séquence

Cette section passe en revue les principaux diagrammes de séquence. Ces diagrammes peuvent être trouvés en annexes. Ces diagrammes de séquence ne représentent pas tous les scénarios d'utilisation possibles avec le module de simulation mais bien les scénarios les plus intéressants dans le cadre de la conception.

4.1.1 Initialisation du plugiciel simulation

L'initialisation du plugiciel simulation suit la procédure standard mise de l'avant dans Modeleur2 pour l'initialisation d'un plugiciel.

4.1.2 Ajout d'un ensemble simulateur

L'ajout d'un ensemble simulateur ne se fait pas en cours d'utilisation de Modeleur2. Un objet SNSimulateur représentant le simulateur dans le système Modeleur2, est créé vide et ensuite on lui assigne le nom de la dll de l'ensemble simulateur, puis les paramètres propres au simulateur sont chargés via la fonction `chargeParamsSimulateur` qui doit être exportée par la dll du simulateur. On sauve ensuite dans la base de données seulement les données nécessaires (le nom de la dll) pour le recharger dynamiquement à partir de la dll.

Lors de l'ajout, on doit également communiquer les filtres au module d'importation et d'exportation afin qu'il puisse les utiliser ultérieurement.

4.1.3 Créer une simulation

Lors de la création d'une simulation, le SNGestionnaire charge le SNSimulateur correspondant au simulateur qui sera utilisé. Le SNSimulateur pour un simulateur donné ne sera en réalité chargé que lorsqu'il n'est pas préalablement chargé, sinon le SNGestionnaire recevra une référence au SNSimulateur déjà chargé. Après le chargement de l'objet simulateur, le SNGestionnaire retrouve les GDAlgoIo fournis par la dll et les envoie par événement au gestionnaire BD afin de permet la gestion mémoire de ces objets.

À la création de la donnée SNSimulation proprement dite, le SNSimulateur est assigné à la simulation de même que son objet graphique dont la construction est commandée par le GDAlgoIoSimulation.

4.1.4 Ouvrir une simulation

Très similaire à la création d'une simulation, son ouverture charge par contre toutes ses composantes à partir de GDAlgoIoSimulation. Pour que le GDAlgoIo nécessaire au chargement des calculs soit connu du Gestionnaire BD au moment du chargement, le SNGestionnaire exécute une requête SQL pour retrouver l'identification du simulateur correspondant à la simulation à ouvrir. Le simulateur est chargé et ses GDAlgoIo sont communiqués au Gestionnaire BD avant l'envoi de l'événement d ouverture de la simulation.

4.1.5 Enregistrer une simulation

Opération inverse de l'ouverture d'une simulation. Le GUI maintient un booléen qui est vrai si la simulation a été modifiée depuis la dernière sauvegarde.

4.1.6 Créer un calcul H2D2

Lors de la création d'un calcul, on retrouve les moyens de construire le calcul et son historique. L'historique est assigné au calcul lors de sa création par le GDAlgoIoCalculXXXX.

4.1.7 Ouvrir un calcul (ex. H2D2)

Aucun diagramme n'est associé avec cette opération. Cette opération est en fait la copie d'un calcul d'une simulation à une autre, en autant que les deux simulations utilisent le même simulateur.

L'usager pourra commander cette fonctionnalité graphiquement en effectuant un glisser-déposer. À ce moment, le calcul sera préalablement ouvert donc chargé et il n'y aura qu'à en faire un clone à l'aide du Gestionnaire BD et à en faire l'ajout à la simulation le recevant.

4.1.8 Gérer les paramètres d'un calcul

Le GUI simulation retrouve le nom du simulateur associé à la simulation dont on désire gérer les paramètres afin de retrouver le fragment de GUI à importer et à construire pour la gestion des paramètres. Le fragment de GUI est placé dans un dictionnaire du GUI simulation lorsque l'ensemble simulateur est initialisé.

La gestion des paramètres est laissée à la discrétion de la personne qui implémente l'ensemble simulateur puisque cette partie demande l'interaction du fragment de GUI et de l'instance SNCalculXXXX qui sont tous deux fournis par l'ensemble simulateur.

4.1.9 Exécuter un calcul

Avant de démarrer l'exécution d'un calcul la méthode `estValide` sera appelée sur l'objet SNCalcul afin de s'assurer que les paramètres sélectionnés pour le calcul sont valides. De plus, seul un calcul ayant l'état NON_EXECUTE peut être exécuté.

La méthode `executeCalcul` sera implémentée par la classe SNCalcul parent fournie par le plugiciel simulation. Les différences inhérentes aux systèmes d'exploitation et leur système de fichiers seront encapsulées dans des classes spécialisant la classe SNCalcul de base pour un système d'exploitation spécifique.. L'écriture des fichiers de commandes requiert donc une méthode qui devra être implémentée par la classe SNCalculXXXX afin de retourner une liste des fichiers de commandes à créer et leur contenu respectif.

L'assignation de l'état du calcul sera faite avant de démarrer le calcul afin d'éviter les cas où le calcul occuperait la totalité des ressources de la machine. Si le simulateur le permet, un suivi d'avancement périodique sera effectué et affiché dans le GUI. Une manière simple d'implémenter cela est de lire dans un fichier les données sur l'avancement du calcul qui y seront inscrites par le simulateur. Le pourcentage du calcul exécuté ainsi que le niveau de convergence sont des données intéressantes à recueillir. Puisque Modeleur2 peut être terminé durant l'exécution d'un calcul, il faut pouvoir reprendre le suivi d'avancement sur les calculs qui étaient en cours d'exécution à la fermeture de Modeleur2.

4.1.10 Importer les résultats

À l'importation des résultats, l'usager aura la chance de choisir le préfixe à attribuer aux résultats qu'il importe Par défaut, ce préfixe sera formé à partir du nom du calcul qui a fourni ces résultats. Pour ce qui est de l'importation proprement dite, le SNGestionnaire récupère la liste d'objets IEImport de l'instance SNCalcul et la transmet au module Import/Export qui se charge d'enregistrer et de lier les nouveaux résultats dans la BD.

4.2 Revue des exigences du module simulation envers un ensemble simulateur

La liste des exigences pour un ensemble simulateur a été dressée. Celle-ci décrit les composantes qui doivent être fournies avec un simulateur pour pouvoir l'intégrer à Modeleur2.

Parmi les fonctions exportées par la dll, celles dont le nom débute par `cree` retournent une instance de la classe spécifiée qui a été construite par la dll. Quant à la fonction chargeParamsSimulateur, elle prend en entrée une instance SNSimulateur vide et y charge tous les paramètres propres au simulateur. Une instance SNSimulateur est la représentation du simulateur au niveau du module de simulation dans Modeleur2.

Voici la présente liste de ces exigences :

Exigences pour un ensemble simulateur

- Filtres d'importation et d'exportation
- Fragment de GUI (DlgGereParametres)
- dll contenant :
    - GDAlgoloCalculXXXX
    - GDAlgoloHistorique (facultatif)
    - GDAlgoloObjetGraphiqueHist (facultatif)
    - Classes :
        - SNCalculXXXX héritant de SNCalcul,
        - SNHistoriqueXXXX,
        - SNObjetGraphiqueHistXXXX

  et exportant les fonctions suivantes :
    - creeGDAlgoloCalcul()
    - creeGDAlgoloHistorique()
    - creeGDAlgoloObjetGraphiqueHist()
    - chargeParamsSimulateur(SNSimulateurP)

La classe SNCalculXXXX devra :

- Pouvoir retourner les listes <IEExport> et <IEImport>. Ces objets devront être créés par le fragment de GUI (*Python*) ou l'instance SNCalculXXXX (*C++*).
- Pouvoir retourner une liste des fichiers de commandes à créer et leur contenu.
- Maintenir l'état du calcul et autres données utiles à l'historique ainsi que les méthodes pour les récupérer.

5 État actuel

Dans l'état actuel du module simulation, les principales interactions ont été implémentées, soient l'intégration du module sur le bus de transmission, la réception et l'envoi des événements et leur traitement, ainsi que les interactions entre les parties C++ et Python et le partage des objets entre les deux.

Il est possible de créer des simulations, de leur ajouter des calculs et de commander leur exécution sur la machine locale.

Les parties n'ayant pas encore été implémentées :

Les interactions entre le module de simulation et le Gestionnaire BD n'ont pas encore été implémentées et il n'est donc pas possible présentement d'enregistrer ou de charger des composantes de simulation. Cela implique l'implémentation des classes GDAlgoIo pour interagir avec la base de données ainsi que la création des tables dans le schéma de la base de données pour accueillir les données reliées aux simulations. Pour l'instant, ces interactions se font au moyen de l'instruction `new` encapsulée dans les GDAlgoIo.

L'objet graphique de l'historique n'a pas été implémenté, il faudra étudier de quelle manière il est le plus souhaitable de présenter les informations contenues dans les historiques des calculs. Il serait également intéressant de pouvoir interagir avec les éléments de l'historique au moyen de la souris en pouvant sélectionner des calculs directement dans l'historique et en ayant accès à un menu contextuel.

Parmi les éléments du GUI, notons que la barre d'outils et les menus contextuels ne sont pas fonctionnels.

La façon dont les mécanismes permettant l'exécution à distance des calculs seront implémentés n'a pas encore été déterminée. Il s'agit principalement de permettre la configuration des paramètres d'exécution à distance au niveau du module simulation et d'implémenter un mécanisme permettant d'harmoniser cette configuration avec celle du calcul pour produire les fichiers de commandes pour l'exécution du calcul. Ceci s'applique également pour l'exécution de calculs sur un cluster.

Pour l'implémentation de cette fonctionnalité, il serait souhaitable d'avoir une classe de base spécifiant l'interface et des classes les implémentant selon les particularités des différents systèmes d'exploitation et systèmes de fichiers supportés par Modeleur2. En passant par l'interface commune, les interactions avec différents systèmes d'exploitation seront transparentes pour l'utilisateur.

Le chargement de la dll d'un simulateur au moyen de l'objet SNSimulateur n'est pas encore en fonction et il n'y a pas eu de prototype d'ensemble simulateur préparé pour pouvoir démontrer la boucle complète des étapes d'une simulation.

6 Conclusion

6.1 Résumé

Dans ce document, on a présenté l'activité simulation et on a résumé le travail d'analyse qui lui a été consacré. De plus, il a été question du design et de l'implémentation effectuée jusqu'à maintenant en ce qui concerne le module simulation.

6.2 Perspectives futures

Au moment de la rédaction de ce document, le prototype en est à sa toute première ébauche.

Pour le module simulation, les entêtes et les interfaces des classes principales ont été mises en place. De plus, la mise en place de la mécanique d'envoi/réception d'événements a été effectuée. Les objets partagés en Python ont été publiés et leurs interactions fonctionnent bien.

La suite logique des événements est de compléter le prototype du module en suivant les conclusions du travail d'analyse. Cela demande entre autres de:

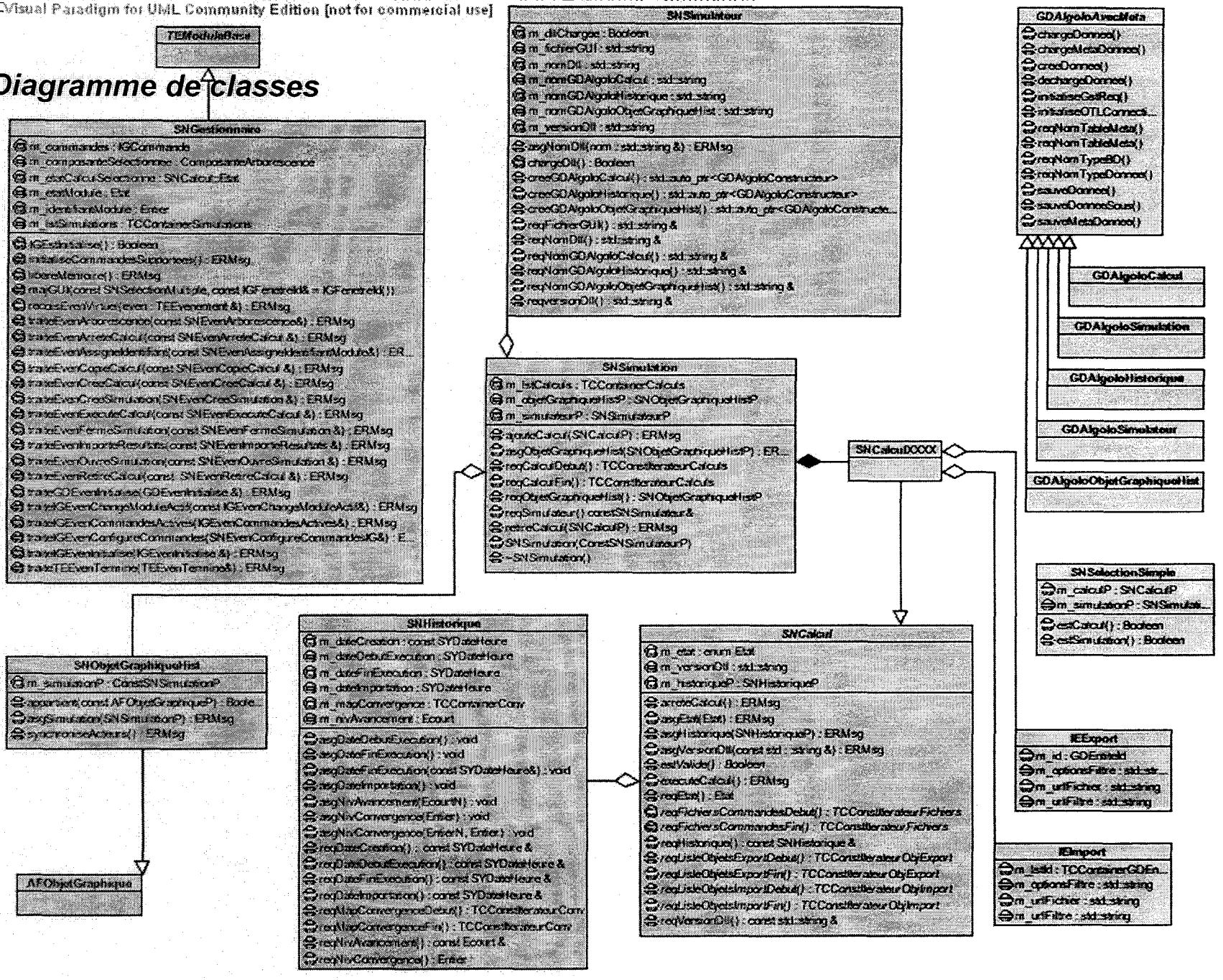
- Modéliser les tables reliées à la simulation dans la BD;
- Revisiter et implanter la création/chargement/déchargement/sauvegarde des structures de simulation (GDAlgoIoSimulation, GDAlgoIoSimulateur) dans le gestionnaire de BD
- Implémenter l'objet graphique de l'historique pour présenter visuellement les données de l'historique.
- Terminer la mise en œuvre du GUI Python (Barre d'outils, menu contextuel);
- Déterminer la méthode à utiliser pour l'exécution de calculs à distance et la fusion de la configuration à distance avec celle du calcul pour produire les fichiers de commandes.

7 Annexe

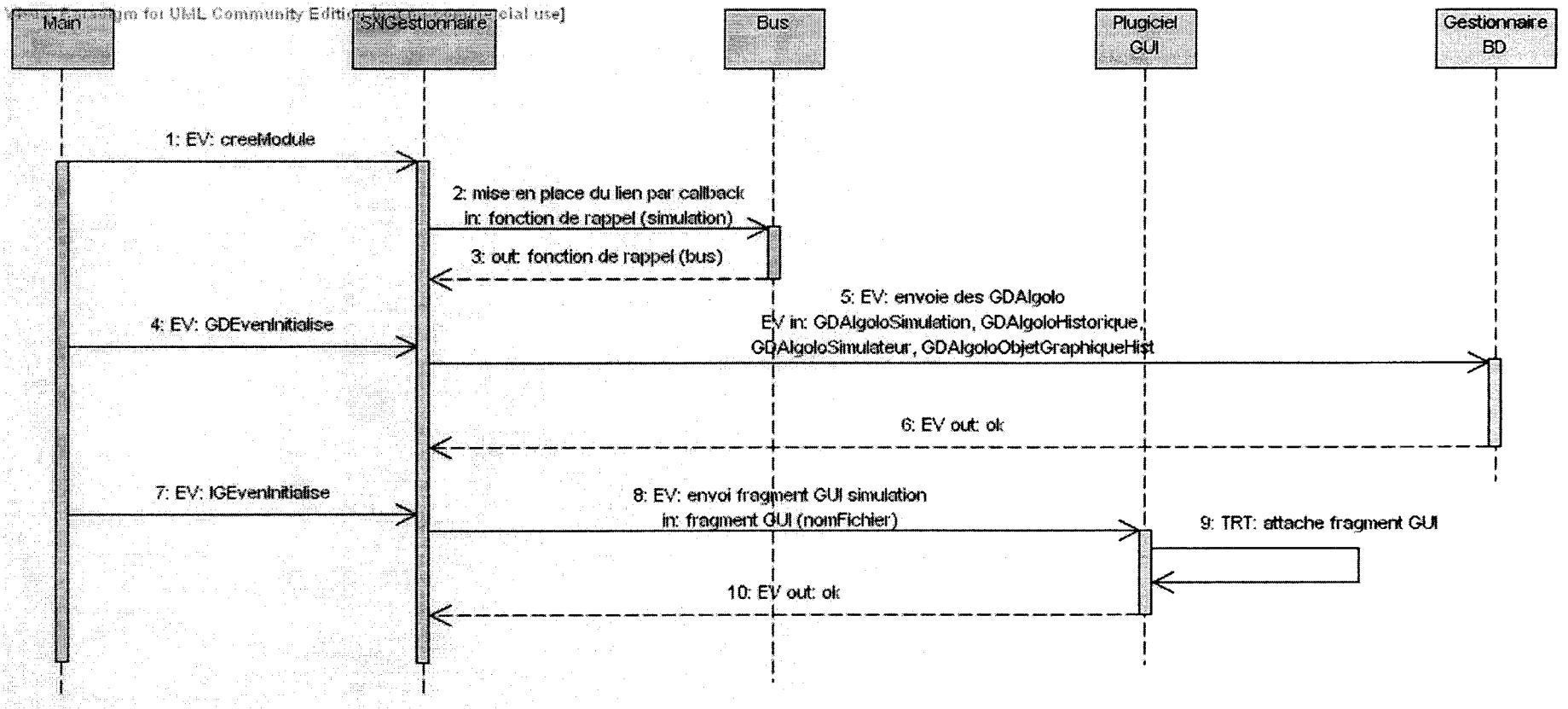
7.1 Terminologie et acronymes utilisés dans les diagrammes de séquence

- asg : Assignation;
- BD : Base de données;
- DLG : Boîte de dialogue;
- EV : Événement envoyé sur le BUS;
- EV in : Paramètre d'entrée envoyé avec l'événement;
- EV out: Paramètre de retour de l'événement;
- GUI : Interface graphique utilisateur;
- MAJ : Mise à jour;
- MNU : Item de menu;
- req : Requête;
- TRT : Traitement interne;

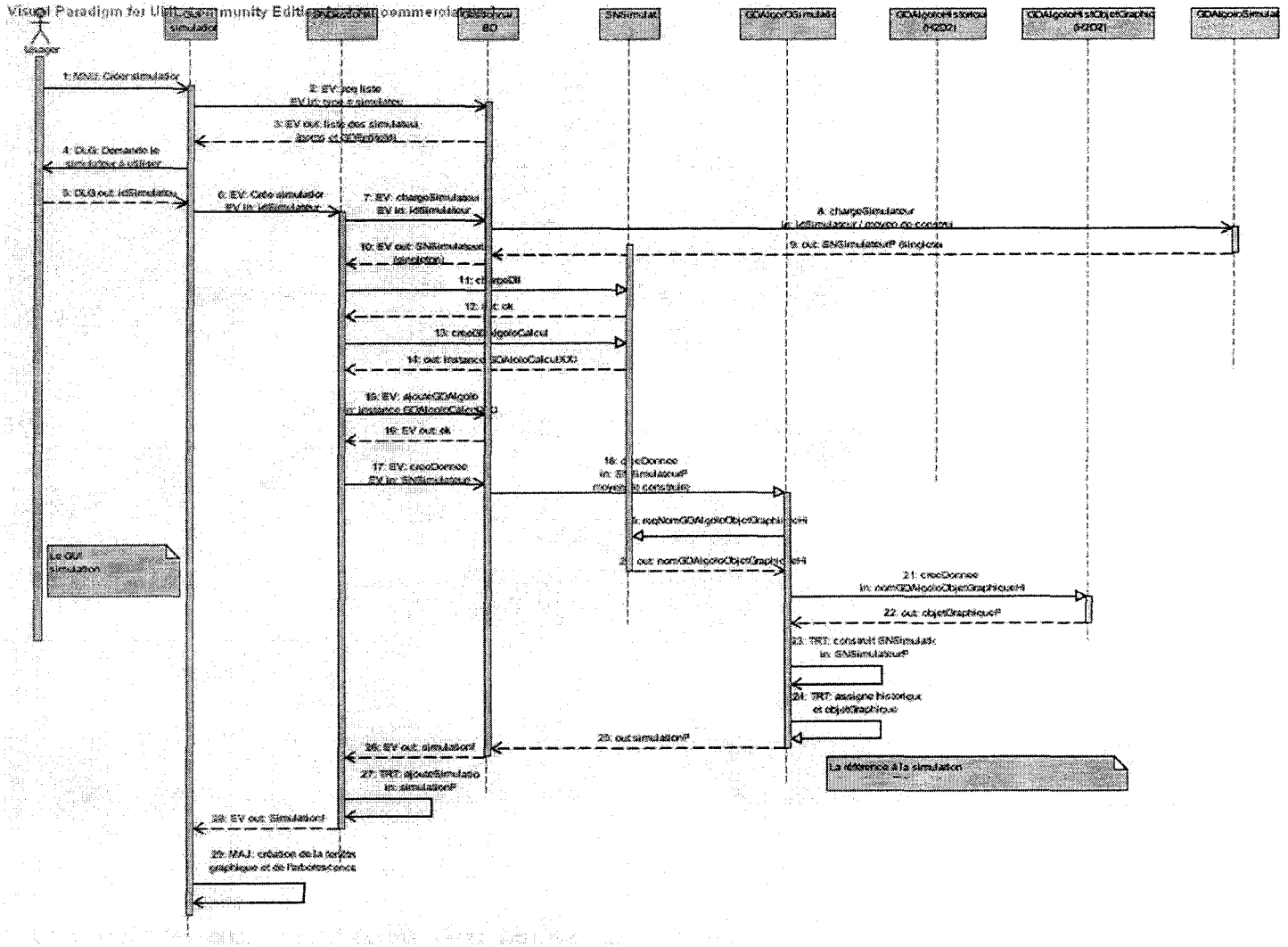
7.2 Diagramme de classes



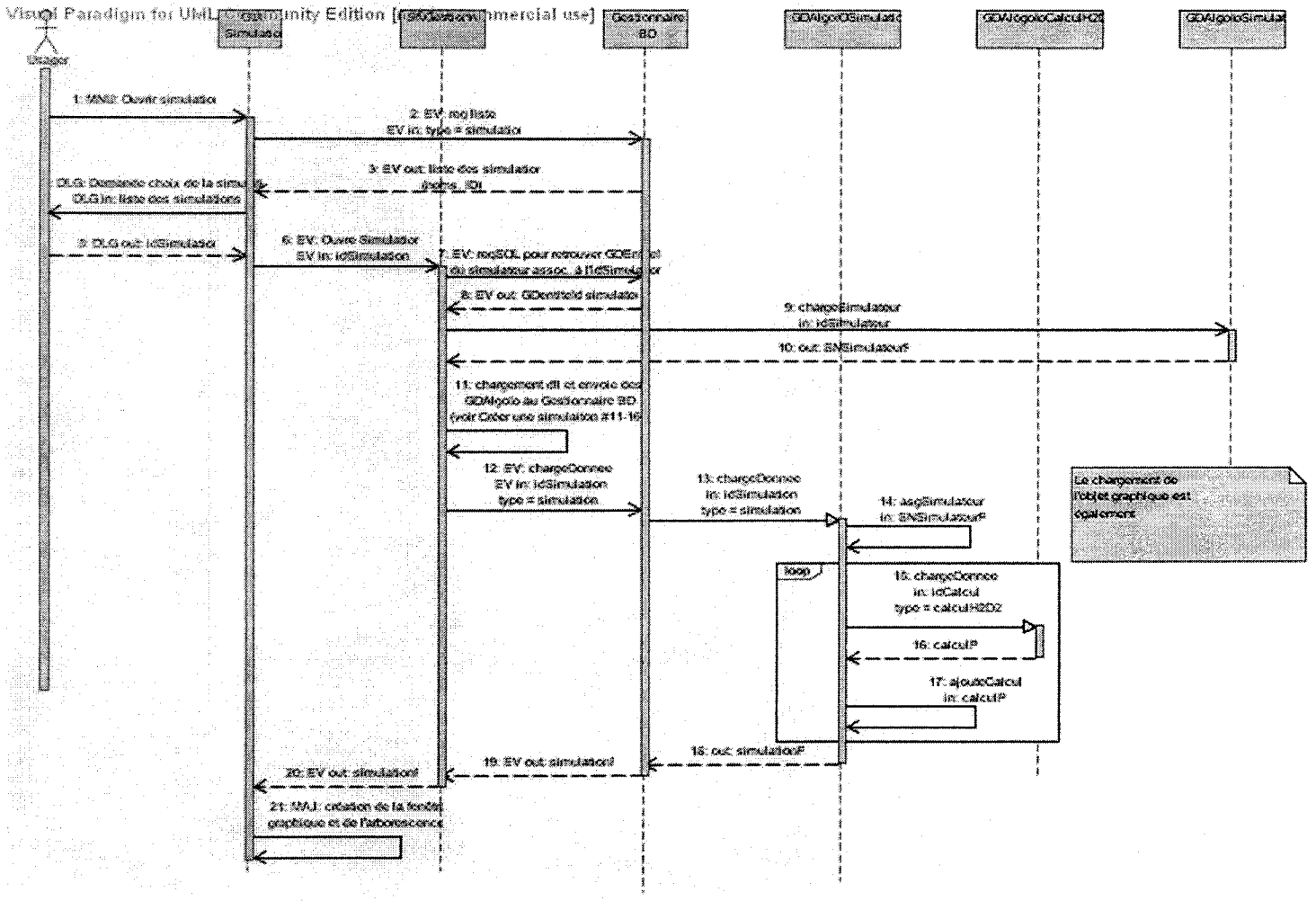
7.3 Initialisation du plugiciel simulation



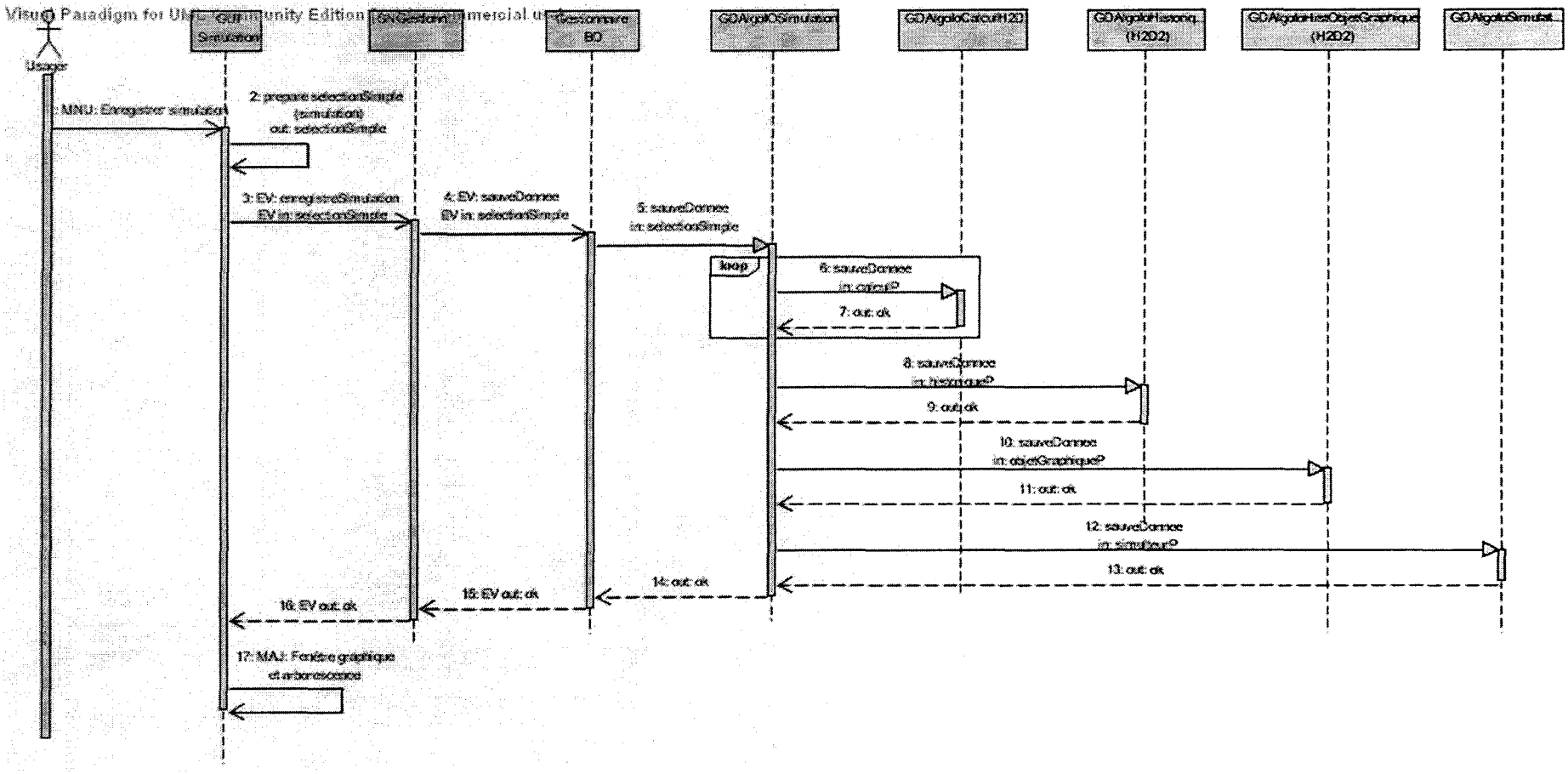
7.4 Créer une simulation



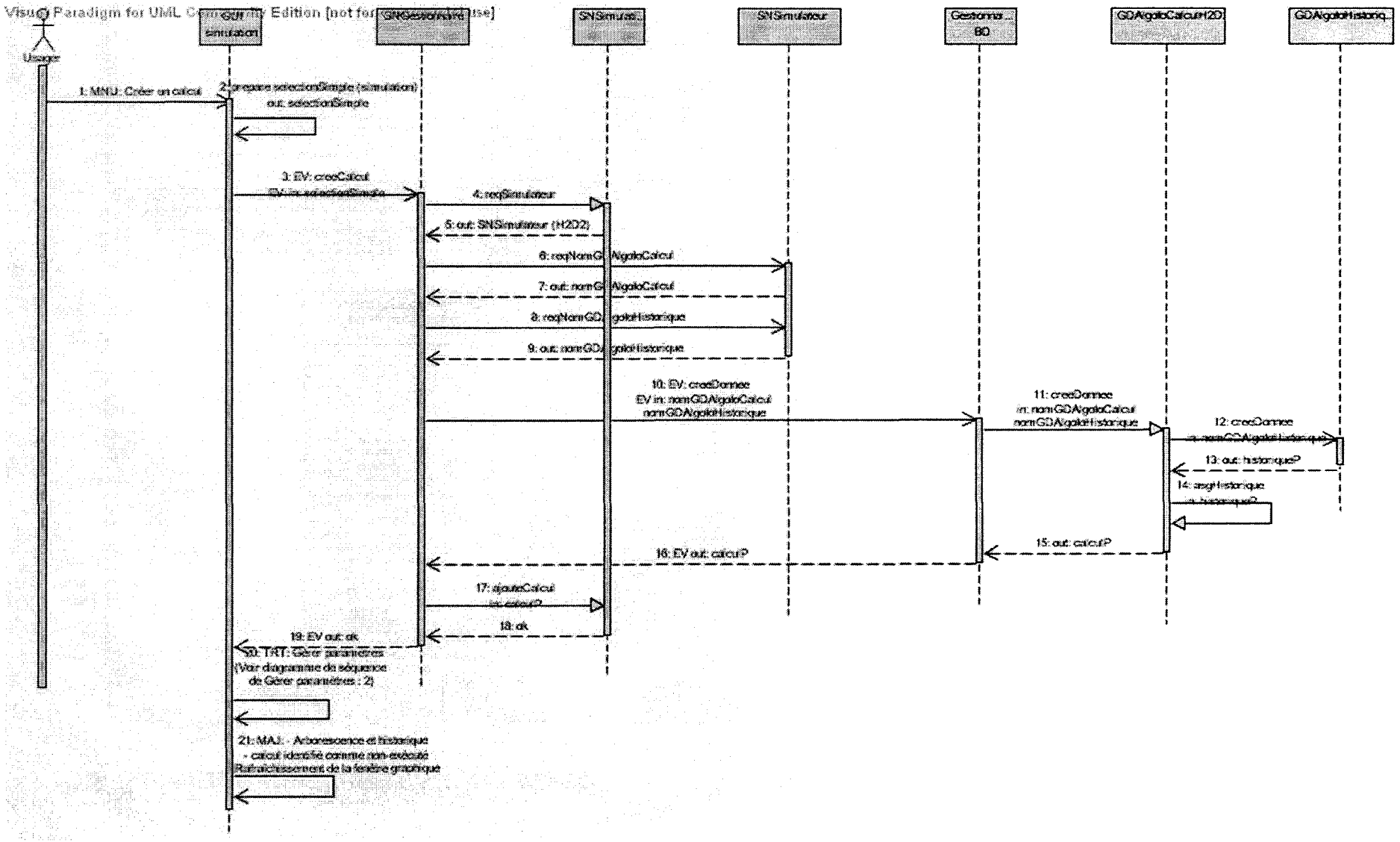
7.5 Ouvrir une simulation



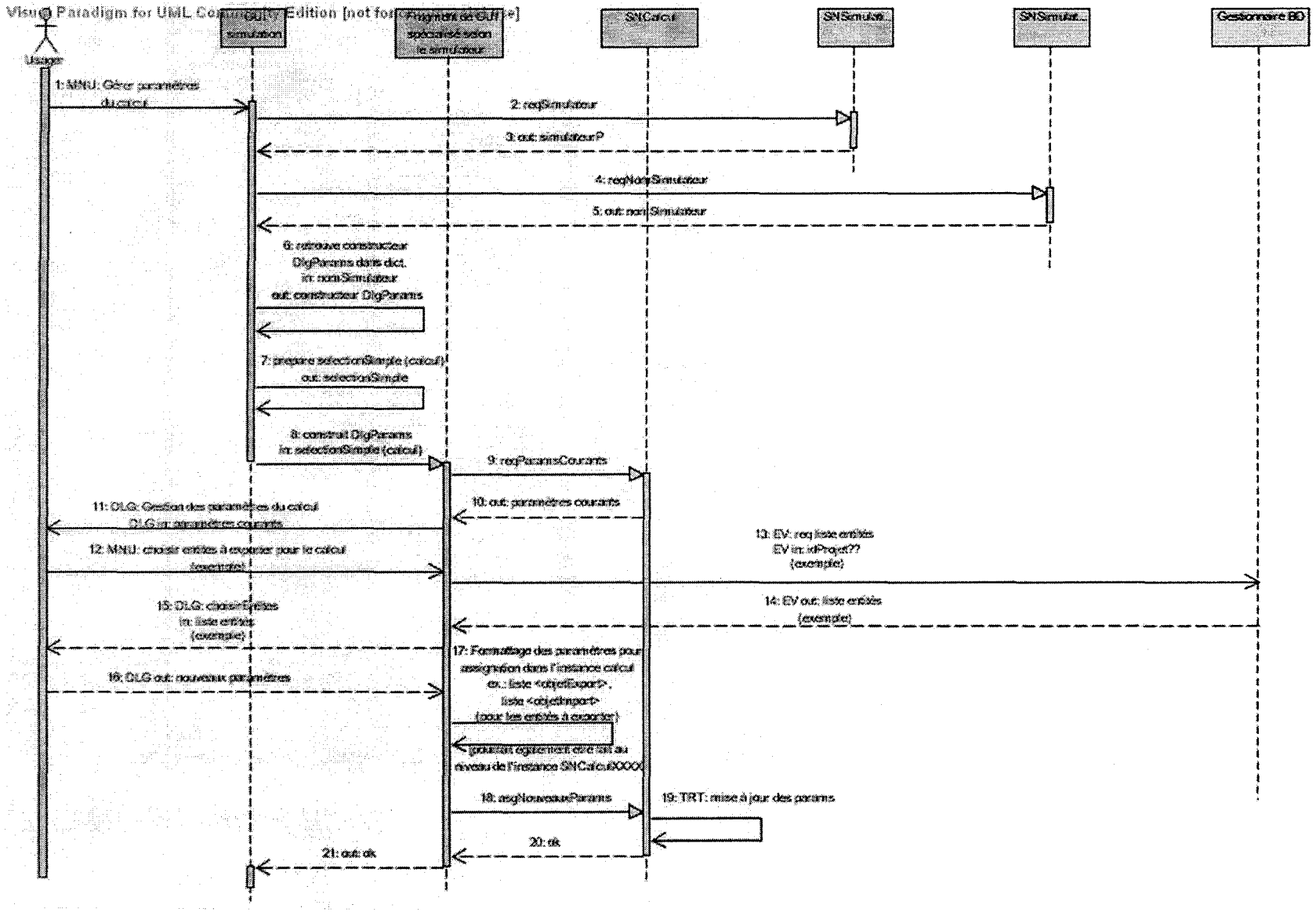
7.6 Enregistrer une simulation



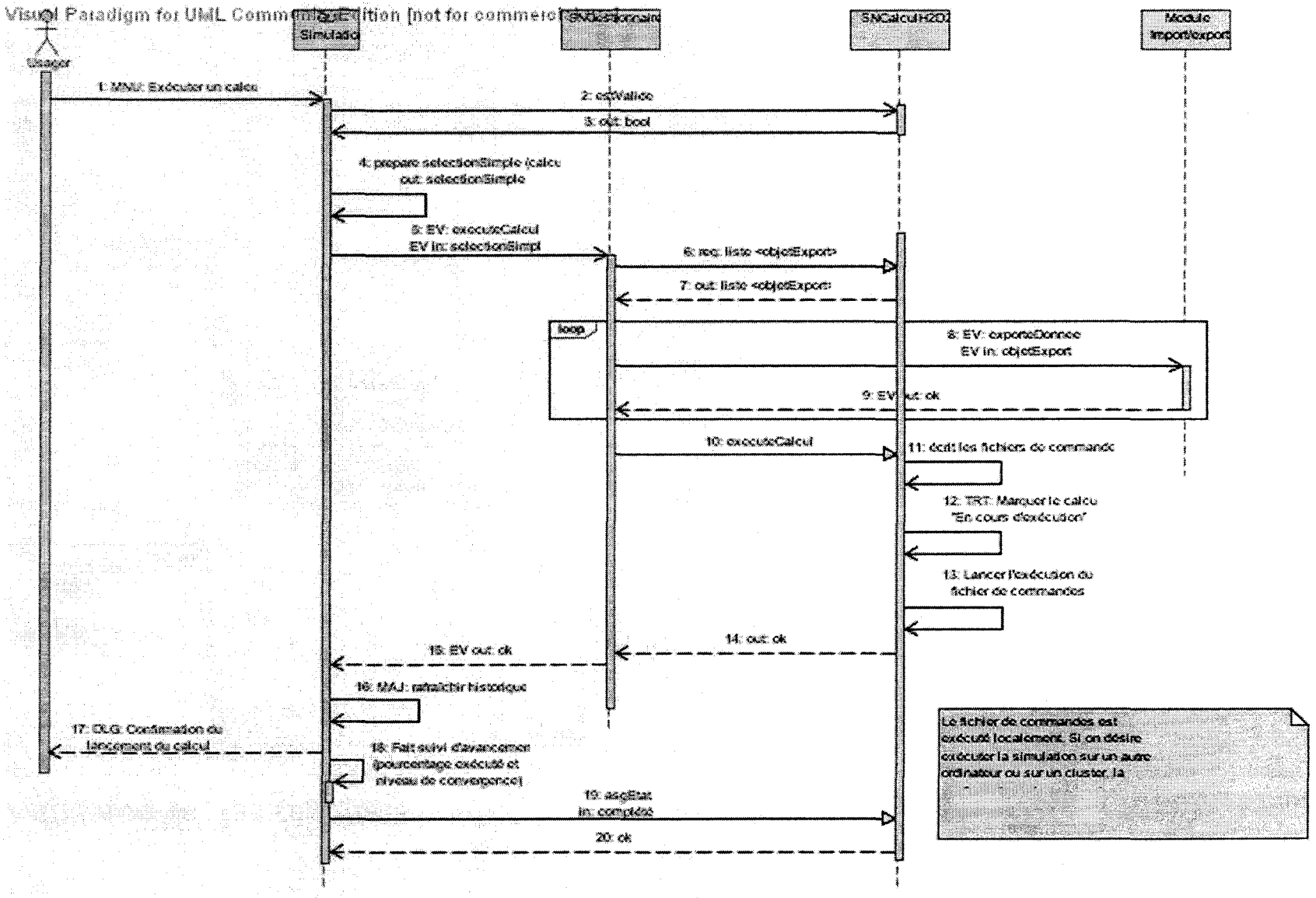
7.7 Créer un calcul H2D2



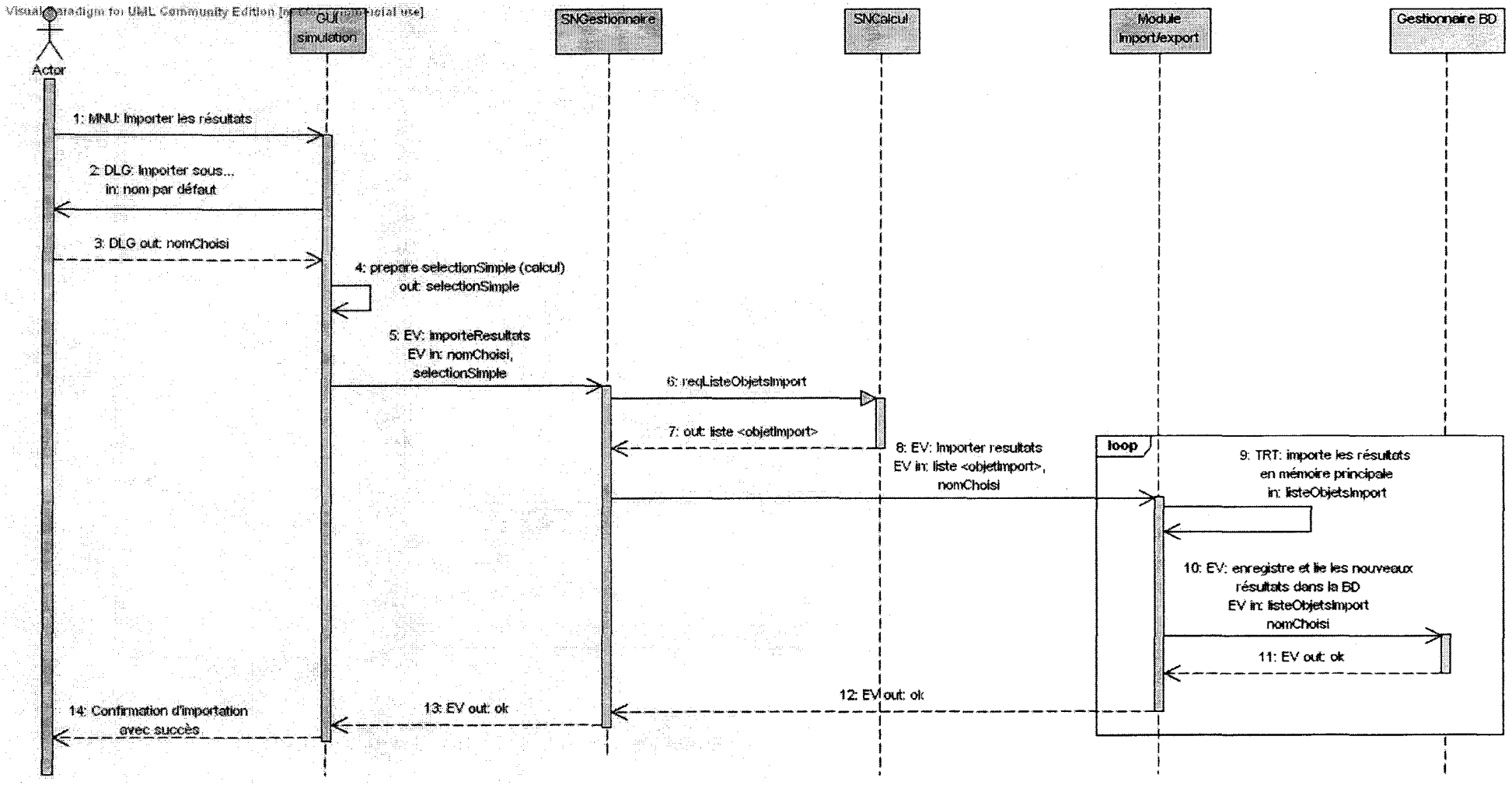
7.8 Gérer les paramètres d'un calcul



7.9 Exécuter un calcul



7.10 Importer les résultats



RAPPORT # 2 : Séries

Rapport de développement logiciel

Séries

Présenté par :

Sybil Christen

13 janvier 2006

Versions du document

Date	Auteur	Commentaires	Version
2005-08-26	Sybil Christen	Version intiale	0.1

1. Introduction

1.1 Objectifs

L'objectif de ce rapport est de résumer les modifications effectuées dans Modeleur au cours de l'été 2005 en lien avec les séries et leur interface graphique. On y présente les notions mises en oeuvre ainsi que quelques notions utiles au développeur désirant effectuer des modifications à cet aspect du logiciel.

1.2 Référence

[1] Spécifications de Modeleur II, INRS,
[2] Rapport de développement logiciel : Champs-Séries, par Olivier Kaczor
[3] Rapport de développement logiciel : Régions, par Sybil Christen
[4] Rapport d'analyse : GDTypes, par Sybil Christen
[5] Compte rendu de développement : Séries, par Sybil Christen

1.3 Contexte

Les objectifs poursuivis lors de cet étape du développement étaient :

- De compléter les classes relatives aux séries ;
- De tester les fonctionnalités d'interpolation, d'itération, etc. pour les différentes sortes de série possibles ;
- De publier les classes en Python pour chacune des sortes de séries possibles ;
- De préparer des GDAlgoIo pour chaque sorte de série possible en employant la structure de base de données déjà développée;
- D'implanter une interface GUI permettant de créer et de sauvegarder des séries ;
- D'implanter une interface GUI permettant de manipuler et d'interroger des séries.

Ces objectifs ne sont pas complets au moment d'écrire ce rapport. L'état du développement est donné dans le rapport d'étape [5].

Le présent document décrit l'analyse et les choix effectués au cours de cette étape de développement.

2 Séries

2.1 Définitions

2.1.1 Coordonnée

Au sens courant, une **coordonnée** indique une position sur un axe (abscisse ou ordonnée, etc). Dans Modeleur, un objet Coordonnées (typiquement la classe GOCoordonneesXYZ) regroupe *les coordonnées sur chacune des dimensions* du plan de travail. Ainsi, dans les documents de Modeleur, l'emploi du terme « coordonnée » au singulier peut référer à un objet Coordonnées, possédant plusieurs coordonnées. Exemple : « l'itérateur retourne une paire coordonnée-valeur ».

2.1.2 Région et itérateur

Les **régions** sont employées par les champs (et par extension les couches) ainsi que par les séries. Pour la majorité des champs, il s'agit d'une région spatiale dont le rôle est de répondre à des questions de nature géométrique telles que : *Le point (x,y) est-il situé dans la région ? La couche Y et Z se superposent-elles ?*

Dans le cas particulier du champ élément finis d'une série, il s'agit d'une région ayant des coordonnées non spatiales et décrivant un **espace de solution** (voir définition suivante).

Grâce à la région, on peut parcourir un champ ou une série à intervalles réguliers et extraire les valeurs correspondant aux coordonnées générées. Cela est fait grâce aux **itérateurs de région**.

Une région n'est pas nécessairement continue. Elle peut être constituée de plusieurs polygones, ou contenir des trous.

2.1.3 Espace de solution

Un **espace de solution** correspond à un ensemble de coordonnées (réelles, complexes ou autres) pour lesquelles on possède des données. Par exemple, on pourrait avoir des valeurs de niveaux d'eau en fonction du débit d'une rivière et en fonction du niveau d'eau du fleuve dans laquelle cette rivière se déverse. Les coordonnées pour lesquelles on a des données correspondent à l'espace de solution. Tout comme une région de l'espace physique, on peut interroger l'espace de solution pour savoir si une coordonnée précise y est comprise (pour savoir, par exemple, si l'on possède des données pour un débit X et un niveau d'eau Y). On peut également itérer sur l'espace de solution.

2.2 Concepts relatifs aux séries

2.2.1 Séries simple et composée

Une **série** est une suite de données reliées selon une logique, par exemple des champs de niveaux d'eau en fonction du temps. Elle permet d'interpoler des valeurs aux points situés entre les données.

Il existe deux sortes de série : les **séries simples**, constituées à partir de données de simulation ou de terrain, et les **séries composées** sont dérivées de deux séries que l'on combine.

Une définition plus complète des séries ainsi qu'un rappel des concepts des éléments finis sont données en [1] et [2].

2.2.2 Donnée portée de la série

On appelle donnée portée la donnée examinée par la série. Dans l'exemple d'une série de champs de niveaux d'eau en fonction du temps, la donnée portée est un champ de niveaux d'eau. La série devra pouvoir porter tout type de donnée : des réels, des champs ou tout autre objet qui offre les méthodes nécessaires à l'interpolation (implémentant l'interface d'une structureAlgébrique).

Il sera possible de se déplacer sur la série et de consulter les données portées à des pas de coordonnées réguliers. L'itérateur de région pourra être employé à cet effet.

2.2.3 Dimensions de la série

La série peut avoir plus d'une dimension ou axe : par exemple, une série de niveaux d'eau en fonction du débit d'un cours d'eau et du niveau du fleuve dans lequel il se déverse possède deux axes ou dimensions. En théorie, les séries pourraient avoir autant de dimensions que désiré, mais pour l'instant, elle seront limitées à deux car les éléments tridimensionnels et leur algorithmes correspondants n'ont pas encore été définis dans Modeleur.

Afin de permettre un traitement uniforme de différents types de données au sein de Modeleur2, les séries à zéro dimensions, ne contenant qu'une seule valeur, seront également implantées.

2.2.4 Coordonnées de la série

Les séries doivent permettre l'utilisation de types de coordonnées différentes sur leurs axes, par ex. des réels, des entiers ou des complexes. Le type de coordonnée peut également différer parmi les axes d'une série[1].

2.3 Analyse

2.3.1 Généralités

Dans l'abstrait, toute série possède :

- Un ensemble de coordonnées (de l'espace de solution) pour lesquels on possède des données;
- Un ensemble de valeurs correspondant à ces coordonnées;
- Une région espace de solution au sein de laquelle on peut interpoler des valeurs;
- Un ou plusieurs axes définissant l'espace de solution. Le nombre d'axes détermine les dimensions de la série.

2.3.2 Requis et contraintes

Les besoins de typage des séries est discuté en détail à la section 2.3.5. Les définitions suivantes sont utiles à la compréhension de la présente section :

Type mathématique : type employé pour représenter une donnée, par ex. un réel.

Type physique : type décrivant le contenu d'une donnée, par ex. topographie, température. Il s'agit d'une description générale, le détail (date, lieu, température de quoi) étant plutôt conservée dans les métadonnées.

Unité : format de mesure d'une donnée, par ex. degrés celsius.

0D, 1D, 2D : zéro, une, deux dimensions.

2.3.2.1 Séries simples

Les données portées par une série doivent être compatibles. Elles doivent donc être de même type (mathématique et physique) et d'unités compatibles[2]

.

[1] L'utilité de ce dernier critère est à revoir, car sans lui, le design des classes pourrait être grandement simplifié, notamment en permettant à celles-ci d'accepter un container d'axes. On aurait alors une seule sorte de série simple plutôt qu'une classe distincte pour chaque nombre de dimensions.

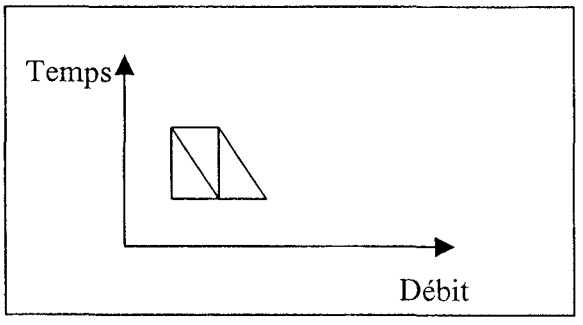
[2] Construire la série avec des unités compatibles ou identiques ? Il sera plus facile d'interpoler avec des unités identiques; on pourrait convertir les unités au besoin lors de l'affectation des champs à la série.

Les données portées par une série sont des classes qui implantent l'interface SAAlgèbre (afin de permettre l'interpolation).

Lorsque la donnée portée d'une série est un champ éléments finis, les champs employés doivent partager le même maillage.

Pour une série 2D, on doit permettre la création de deux axes de même type. En effet, puisque le type physique est une description générale, on pourrait se retrouver avec une série qui modélise une donnée en fonction de la température (de l'air) sur X et de la température (de l'eau) sur Y.

Pour les séries 2D, seules les séries régulières seront implantées dans un premier temps, c'est à dire des séries dont les valeurs sont placées sur une grille d'éléments rectangulaires (Q4) (voir illustration dans les spécifications). Toutefois, les classes doivent être conçues de manière à permettre la création de séries dont les valeurs sont placées irrégulièrement sur le plan formé par les axes (illustré ci-dessous).



2.3.2.2 Séries composées

Une série composée s'utilise de la même manière qu'une série simple. Par contre, elle est construite différemment, en combinant deux séries.

Il doit exister un axe compatible parmi les axes de chacune de ces deux séries (même type de donnée physique, logique, et même unité ou unités convertibles). Sur l'axe compatible, il doit exister une plage de valeurs communes (la série combinée ne portera que les valeurs de la plage commune).

On peut combiner une série 1D de champs avec une série 1D de scalaires. Mais on ne peut pas combiner deux séries 1D de champs, puisqu'on ne peut pas modéliser des champs sur un axe.

2.3.3 Sortes de séries

Il existe de nombreuses combinaisons de série. En effet, la combinaison de série varie selon :

- La sorte de série (simple ou composée) ;

- Le nombre de dimensions (0,1,2 ...) ;

- La donnée portée. Les données dont l'usage est immédiatement prévu sont : des réels, des champs élément finis (scalaires, vectoriels ou ellipse erreur) et des champs analytiques (scalaires, vectoriels ou ellipse erreur) ;

- Le type des coordonnées : réel, complexe. Il est à noter que le type de coordonnée peut différer d'un axe à l'autre. (aucune classe Coordonnées ne modélise ceci présentement dans Modeleur2) ;

- Pour une série simple, le type de nœud employé au sein de son maillage. Pour l'instant, seul le MGNoeud<GOCoordXYZ> est employé, mais cela pourrait éventuellement changer.

Le nombre de séries existantes pourrait très rapidement se multiplier. Si l'on prévoit huit sortes de données portées (6 champs, réels, entiers), un maximum de trois dimensions, et une seule sorte de coordonnée (3D réelles), on aurait 32 sortes de séries simples (sans compter les séries composées). En ajoutant deux types de coordonnées (en postulant que tous les axes possèdent toujours le même type de coordonnée, ce qui ne sera pas nécéssairement le cas), on obtient 104 sortes de séries simples!

La plupart du temps, les séries seront employées pour porter des données de type Champ. Il existe présentement 6 sortes de champs dérivant toutes de la classe parente SRChamp. Dans le présent cas, il aurait été pratique de définir une classe de série générique pour les champs dont la donnée portée serait un SRChamp (dont on ne précise pas la sorte spécifique). Ce genre de programmation générique est impossible dans le cadre de Modeleur2, puisque les classes SRChamp sont des classes templates qui doivent être explicitement instanciées à la compilation. De plus, de manière générale, c'est la programmation par template plutôt que la programmation générique qui domine dans Modeleur2. Les séries doivent donc également être implantées en tant que classes template.

Chacune des éventuelles (104 ou plus ou moins) classes de séries devra être instanciée à partir des templates afin que l'on puisse les créer et les enregistrer dans la base de données. Cela exigera la déclaration d'un GDType pour chacune, tel que décrit à la section suivante.

2.3.4 GDType des séries

Les variantes de type de données portées et de type de coordonnées des séries impliquent l'utilisation de tables (relations) différentes pour enregistrer les séries dans la base de données. Afin d'associer chaque sorte de série à une classe de la famille GDAlgoIo (responsable de l'insérer dans la base de données et de l'instancier en mémoire) un typage des sortes de séries est requis. Dans Modeleur2, chaque entité possède un GDType qui joue ce rôle.

Un GDType décrivant une série aurait au minimum les composantes suivantes :

- Sorte : Série + Simple | Composée;
- Dimensions : 0D | 1D | 2D | etc. ;
- Donnée portée : ChampEFScalaireNiveau_eau ;
- Type des coordonnées des axes : GOCoordonneeXYZ.

Les composantes exactes sont à déterminer. Le rapport d'analyse sur les GDType [4] décrit une proposition nomenclature qui pourrait être utilisée dans le cadre des séries. On emploie ici le nom de la classe de Coordonnee pour la décrire, mais il faudrait plutôt employer un GDType plus formel.

Il est à noter que le GDType décrit ci-dessus ne permettrait pas de filtrer les entités en fonction du type de donnée physique des axes, puisque celui-ci n'est pas décrit dans le GDType. Pour le permettre, il faudrait détailler le type correspondant à la coordonnée.

2.3.5 Besoins au niveau du typage

Afin de valider les combinaisons des séries (séries composées), celles-ci doivent être typées. La présente section décrit les besoins des séries au niveau du typage.

Trois composantes de type sont nécessaires :

- L'unité: degrés Celsius, centaine de m^3/h;
- Le type « mathématique » : Réel, ChampScalaire;
- Le type « physique », ou ce qui est représenté : topographie, température. Le type physique ne décrit qu'une généralité et les détails seront stockés dans les métadonnées.

Ces types seront prédéfinis et non au choix de l'utilisateur.

La solution choisie devra permettre de :

- Filtrer les types qui peuvent être représenté sur un axe (ex. scalaireTemperature : oui; champEFScalaireTemperature : non).
  Nécessaire pour : présenter à l'utilisateur les choix de type représentables sur l'axe lors de la définition des axes d'une série.
- Filtrer les types qui offrent les opérations algébriques requises pour interpoler (=interface SAAlgebre?).
  Nécessaire pour : présenter à l'utilisateur les choix de type portés par une série. Alternativement, ceci pourrait être *hardcodé*.
- Extraire le type de donnée physique ou mathématique d'un GDType.
  Nécessaire pour: l'annotation des axes et/ou l'insertion de métadonnées.

- D'effectuer des conversions entre unités, que ce soit entre mm et pouces; ou entre mm et centaines de mm.
  Nécessaire pour: combiner deux séries dont les unités sont convertibles.

La classe GDType permet déjà de représenter les types mathématiques et physiques combinés : elle le fait par exemple pour les entités couches (couche Topographique). Cette classe est présentement utilisée pour trouver les GDAlgoIo appropriés pour créer un objet, ainsi que pour obtenir une liste des objets existants.

Présentement, le GDType de nombreuses entités ne comprend pas le type physique (par ex. les entités champ), celui-ci n'étant pas nécessaire pour déterminer le GDAlgoIo à utiliser. Ce type physique devra leur être ajouté, ce qui ne devrait pas avoir d'impact sur le fonctionnement des GDAlgoIo actuels. La classe GDType devra aussi être élargie afin de permettre la représentation des séries et leurs multiples composantes (données portées, coordonnées, dimensions). Une analyse plus approfondie est requise et débutée en [4].

Des améliorations sont également requises au niveau de l'obtention de listes d'objets filtrés. Il est présentement possible d'obtenir tous les objets en mémoire en fonction de leur GDType ou d'une composante du GDType (par exemple tous les objets dont le type contient *scalaire*), mais il n'est pas possible de filtrer en fonction du rôle joué par la composante (par ex. trouver tous les types dont le type mathématique est *scalaire*, trouver si un type donné est de type physique *température*).

Il faudra rendre disponible en Python une méthode permettant d'obtenir une liste des GDTypes disponibles.

Les unités seront conservées à part. Les classes qui les modélisent n'existent pas encore, mais une analyse a été entamée. La solution de représentation proposée par Barton, Nackman (Scientific and engineering C++) a été retenue :

- On représente les unités fondamentales par un ensemble de classes: masse, longueur, temps, intensité
- On représente les unités dérivées tel que vitesse (longueur/temps), accélération (longueur/temps^2) dans des classes dérivées des unités fondamentales;
- On représente les unités spécifiques dans des classes dérivées. Acceleration m_par_s; Longueur mètre; Longueur pieds, etc.

Cette méthode permet de capter des erreurs de conversion de types à la compilation.

Les classes d'unités fondamentales fournissent des opérateurs pour l'addition d'unités similaires et la multiplication par une quantité sans unités. Les opérateurs d'addition et de multiplication par des unités spécifiques ne sont définies qu'entre unités convertibles.

2.4 Conception

2.4.1 Classes série simple

La classe SRSerieSimple possède les attributs suivants :

- Un maillage dont les nœuds représentent les coordonnées de l'espace de solution pour lequel on possède des valeurs[3];
- Un champ élément finis, lié au maillage de la série, (dit champ *de série*) portant les données de la série. (si les données portées sont des champs, nous avons un champ portant des champs).
- Une région espace de solution, celle qui correspond au maillage de la série, donc celle que l'on retrouve dans le champ.
- Un attribut axe pour chaque axe.

La série simple est implantée à l'aide d'une classe template dont les paramètres template sont[4] :

- TTDonnee: type de donnée portée
- TTMaillage: type de maillage. Le maillage définit le type de coordonnées, le type de nœud et le type d'éléments de la série.

L'attribut maillage de la série simple est un MGMaillage dont le type de Coordonnées peut varier. Le MGMaillage possède une liste de MGNœuds, classe également template sur les coordonnées. Le maillage d'une série est de type BASE. Une série à zéro dimension possède un maillage d'un seul élément (de type point) et d'un seul nœud[5].

L'attribut champ de la série simple est un SRChampElementFinis[6], template sur le type de donnée portée et le type de maillage (donc le type de coordonnées).

Les points suivants différencient un champ régulier d'un champ *de série*:

- Il n'y a qu'un seul champ de série par maillage de série.

---

[3] On aurait pu éviter de définir un attribut maillage et plutôt employer directement celui du champ. La définition d'un attribut maillage distinct était prévue afin de permettre le réemploi du GDAlgoIoChamp pour la sauvegarder du champ de série (ce GDAlgo ne sauvegarde pas le maillage, dont le champ n'est pas propriétaire), mais vu l'implantation d'une structure de base de données complètement distincte, cette logique ne tient plus.

[4] Les paramètres template d'un Série abstraite diffèrent : elle exige seulement un type de donnée et un type de coordonnées en paramètre. On pourrait voir une Série abstraite comme étant un ensemble de Coordonnées, auquel cas l'implémentation courante est adéquate, ou comme un ensemble de Noeuds et d'Éléments, auquel cas elle devrait avoir un paramètre template Maillage tout comme la Série simple. Ce sera A REVOIR lors de l'analyse et la mise en oeuvre des séries composées : on pourra alors mieux définir la classe abstraite Série.

[5] A REVOIR : il serait peut-être plus efficace de ne conserver que la donnée, sans maillage ?

[6] Ici, élément finis indique que ce champ porte les valeurs d'un maillage (d'un MGMaillage et pas nécessairement EFMaillage). Pour un maillage, le terme EF indique autre chose, à savoir des éléments spatiaux avec coordonnées 3D réels. Donc, on emploie ici un champ *element finis* lié à un maillage *non EF*.

- Le maillage de série représente des coordonnées d'un espace de solution : des coordonnées non spatiales (pas nécessairement des réels, non convertibles vers GEOS) sur des axes de temps ou de débit par exemple.
- La région du champ de série n'est pas spatiale, mais représente un espace de solution. Elle est donc représentée par une SRRegion et non une GORegion comme pour la majorité des champs. [3].

2.4.2 Classe axe

La classe SRAxe modélise les axes, qui sont des attributs d'une série simple.

Ses attributs de la classe SRAxe sont le GDType de l'axe (type mathématique = obligatoirement scalaire, physique = variable) et les indices minimaux et maximaux de la série sur cet axe. Cette dernière information est utile pour la création de séries composées.

Puisque le type de coordonnée employée peut varier selon les séries, la classe SRAxe est template de la sorte de coordonnée[7]. Le paramètre template TTType modélise le type mathématique seulement et non le type physique.

La liste des pas pour chaque axe n'est pas un attribut de la classe, parce qu'elle est utile et n'a de sens seulement lors de la construction d'une série grille régulière. On modélise des classes de série et d'axe le plus générique possible.

2.4.3 Classes série simple 0D, 1D, 2D

Il existe une classe représentant les séries simples à zéro dimensions, une classe pour les séries simples à une dimension, une classe pour les séries simples à deux dimensions. Elles dérivent chacune de la classe série simple. Elles possèdent respectivement zéro, un et deux attributs axe.

On ne peut pas faire usage d'une seule classe série simple, indépendant du nombre de dimensions, possédant un attribut container d'axes, dans la mesure où classe Axe est template du type de coordonnée et que ce type peut varier entre axes[8]. Le type de donnée du container d'axe ne peut donc être déterminé à la compilation.

Il existe deux autres raisons de modéliser des classes distinctes selon les dimensions :
- La signature des méthodes d'itération (reqIterateurChampDebut) diffère en fonction du nombre de dimensions ;
- On pourrait vouloir optimiser la série 0D et ne pas y conserver d'attribut maillage ou champ (on conserver alors simplement l'unique valeur portée, tout en offrant la

[7] Note : Il s'agit *d'une* coordonnée. On nomme ce paramètre template TTType plutôt que TTCoord afin de ne pas le confondre avec le paramètre template Coordonnees, utilisé dans plusieurs classes, lequel modélise un *ensemble* de coordonnées.

[8] Possiblement à revoir. Si ce critère n'est aps essentiel, il simplifie considérablement la mise en place des classes, Voir note précédente à ce sujet.

même interface que pour les autres séries). Cela sera d'autant plus utile si on emploie fréquemment des séries 0D. Pour cette même raison, on ne révèle pas l'existence du maillage dans l'interface des séries (on n'y offre pas de méthode reqMaillage, on y reflète plutôt les méthodes du maillage qui sont utiles, comme les itérateurs de nœuds).

2.4.4 Construction de la série simple

La construction de la série est faite en dehors de la classe SRSérie, à l'aide d'une classe algorithme. Cette implantation permet d'ajouter des algorithmes pour créer des séries/maillages différents sans avoir à modifier les classes de série.

Le premier algorithme de création de séries à être implanté, « CreateurSerie », initialise des séries 1D et 2D *régulières* en fonction de pas sur chacun des axes. L'algorithme s'occupe de générer un maillage, une région correspondante et des axes appropriés. L'algorithme fournit également un méthode pour créer une série 0D à partir d'une seule valeur. Puisqu'il n'y a pas d'axes ni de pas dans une série 0D, et puisque l'utilisateur ne construira pas explicitement de telles séries [2], un algorithme simplifié a été conçu qui non seulement génère la série, mais de plus lui assigne sa valeur

Les classes série possèdent deux états : vide (objet instancié avec le constructeur par défaut) et initialisée (attributs initialisés). L'algorithme de CreateurSerie s'occupe de l'initialisation. La méthode privée estInitialisée indique l'état de l'objet. Elle est employée lors de la vérification de l'invariant.

3 Interface GUI de création de séries

3.1 Interface utilisateur prévue

La section qui suit décrit l'interface utilisateur prévue. Elle est plus détaillée que celle des spécifications [2] et en diffère dans la mesure où elle ne fait pas usage d'indices mais affiche à l'utilisateur les pas de ses axes directement à l'écran. Cette manière de procéder semble plus prometteuse parce que plus intuitive pour l'utilisateur et aussi plus simple à mettre en oeuvre, mais elle demeure à valider à l'essai.

3.1.1.1.1 Création série simple 0D (sans GUI)

La série à 0-dimensions sera créé automatiquement au besoin à partir d'une valeur spécifiée par l'utilisateur. Ceci sera transparent pour l'utilisateur, afin de lui faciliter la vie. L'interface décrite ci-dessous ne s'applique donc qu'aux séries à une dimension et plus.

3.1.1.1.2 Création série simple 1D et 2D (avec GUI)

Les champs doivent avoir été créés dans Modeleur2 ou chargés dans la base de données au préalable.

L'utilisateur spécifie la donnée portée : son type mathématique (réel), son type physique (température ou niveau d'eau) et son unité (m^3/h). Ces choix sont effectués parmi une liste prédéfinie. L'utilisateur peut spécifier plus de détails sur la donnée portée (ex : température moyenne estivale); cette information sera conservée dans les métadonnées de la série.

L'utilisateur spécifie la dimension de la série à créer (le nombre d'axes). Pour chaque axe, il définit le type mathématique (seuls les réels sont permis actuellement) et physique (température) [ces types sont combinés dans des GDTypes prédéfinis] ainsi que son unité[9].

L'utilisateur saisit ensuite la liste des coordonnées (pas) sur l'axe. Ceux-ci sont soit spécifiées individuellement, soit générés à partir d'une valeur initiale, d'une intervalle et d'un nombre de répétitions.

L'utilisateur indique que la saisie des axes est terminée et qu'il désire saisir des valeurs [bouton Éditer valeurs]. Il peut alors saisir (dans le cas d'une donnée portée réelle) ou sélectionner (dans le cas d'une donnée portée champ) une valeur pour chaque nœud. Pour cela, un tableau des nœuds est présenté. Il liste les valeurs des coordonnées des nœuds[10].

Axe 1 (débit m/s)	**Axe 2 (température °C)**	**Valeur donnée**
0.1	10	Champ niveaux d'eau 2003-01-12 900-600
0.1	20	Champ2
0.1	50	Champ3
0.5	10	Champ4
0.5	20	Champ5
0.5	50	Champ6
2	10	...
2	20	
2	50	

[9] Ici, plutôt que de spécifier séparément les types de données à mettre sur les axes et dans la donnée portée, on pourrait faire choisir parmi une liste prédéfinie de tous les types de séries disponibles. Cette solution est moins conviviale d'autant plus qu'il pourrait rapidement y avoir de nombreux choix. Pour lancer l'événement de création de la série appropriée, on composera un GDType en y insérant les composantes correspondant aux choix de l'utilisateur.

[10] Et non des indices, internes au logiciel, tel que décrit dans les spécifications – comportement à confirmer sur utilisation.

Si l'utilisateur crée une série portant des champs élément finis, il devra commencer par sélectionner le maillage auquel les champs correspondent. Une liste de maillages avec leurs champs correspondant lui sera présentée. Le choix du maillage terminé, seuls des champs correspondant à ce maillage pourront être choisis.

La série peut être créée (en mémoire) ou sauvegardée uniquement lorsque toutes ses valeurs sont initialisées et valides.

3.2 Création de séries : Guide de l'utilisateur

Lancer Modeleur2. Ouvrir un projet.

Dans le menu Outils, choisir Éditeur de séries.

Choisir le nombre de dimensions de la série à créer (l'interface s'ajustera en conséquence ; actuellement, la dimension est fixée à 2).

Choisir la donnée portée et ses unités (actuellement fixé).

Saisir les pas sur les axes. Pour saisir des pas manuellement : saisir le nombre de pas désirés dans la case Pas, puis ENTRÉE au clavier ou appuyer sur le bouton Générer.

Se placer dans la grille de saisie des pas, saisir un chiffre au clavier (texte interdit) puis appuyer sur TAB (et non ENTREE[11]), ou encore cliquer avec la souris sur la case suivante.

Les chiffres doivent être en format anglophone (0.0 et non 0,0)[12]. On peut saisir des pas en double ou dans le désordre, ce qui sera corrigé à la saisie des pas et à la création de la série (mais la correction ne sera pas visible dans l'écran de saisie des pas).

Pour générer des pas réguliers, saisir le nombre de pas désirés, le début et l'intervalle, puis appuyer sur le bouton Générer. On peut ensuite éditer les pas générés à la main, en ajouter ou supprimer (lorsqu'on en supprime, ce sont toujours les pas de la fin qui partent). Pour supprimer des pas, diminuer le chiffre de la case Pas, puis appuyer sur le bouton Générer.

Il n'y a pas moyen de sortir de la zone d'édition des pas avec le clavier[13]. Il est malheureusement plus efficace de travailler avec la souris.

[11] C'est inconsistant. wxPython n'offre pas énormément d'événements pour mieux gérer ceci. A REVOIR.

[12] L'éditeur de Float fourni par wx n'offre pas d'options ;a ce niveau. Je n'ai pas essayé de changer mes paramètres d'utilisateur au niveau de l'OS.

[13] Return sur édition du dernier pas fonctionne mais alors la modification n'est pas prise en compte.

Une fois la saisie des axes terminés, spécifier les valeurs portées en appuyant sur « Éditer données portées ». La fenêtre « Éditer données portées » s'ouvre et présente une grille avec une ligne pour chaque coordonnée de la série. Cliquer sur la grille dans la colonne Donnée (ou utiliser TAB ou les flèches au clavier) pour faire apparaître le dialogue de choix de champs. Choisir un champ puis OK. Lorsque la saisie des champs est terminée, appuyer sur OK, ou encore sur Annuler pour ignorer tous les changements apportées depuis l'ouverture de la fenêtre.

De retour dans la fenêtre Editeur de séries, appuyer sur le bouton Créer. Une fois la série créée, l'interface se met en mode de modification de série, et les bouton Sauver et Fermer sont activés.

Appuyer sur le bouton « Interpoler » pour ouvrir la fenêtre « manipulation de série » pour interpoler ou itérer dans la série. (EN DEVELOPPEMENT. Ne restera sans doute pas sur ce dialogue).

4 Tests

Des tests unitaires en C++ pour la GORegion, GOEnveloppe, GOIterateurRegion, GOMultiPolygone ainsi que pour l'algorithme de filtre maillage et peau sont disponibles dans le module Test_Region[14] (tests à jour en date d'août 2005).

Aucune classe de coordonnée autre que les GOCoordonnéesXYZ n'est présentement utilisée dans Modeleur2. Pour instancier et tester une version générique (n'utilisant pas la spécialisation GO) d'une SRRegion ou SREnveloppe, on peut employer une coordonnée GOCoordXYZ<Entier> (les GOCoordonnéesXYZ étant définies comme GOCoordXYZ<Dreel>).

L'algorithme de filtre peau a été testé avec un EFMaillage simple fermé (éléments T6L) et un maillage simple troué (éléments Q4). Il n'a pas été testé avec des maillages d'éléments mixtes, lesquels n'ont jamais été employés par Modeleur2 à ce jour.

5 Conclusion

5.1 Résumé

Ce rapport résume les principaux concepts et décisions en lien avec le développement des séries effectué au cours de l'été 2005. Les classe modélisant les séries simple de zéro à deux dimensions ont été mises en oeuvre et publiées en Python. Des scripts et une interface de création de Séries à deux dimensions portant des données champ élément

[14] Qui, en fin de compte, a servi à tester tout ce que j'ai développé et non seulement les régions.

finis sont fonctionnels. Plusieurs éléments restent à compléter, dont la sauvegarde des séries dans le base de donnée et l'adaptation du GUI aux différents types de série.

Certaines questions doivent être discutées plus en détail et implantées avant que le développement des séries puisse être terminé, notamment la mise en oeuvre d'un GDType amélioré ainsi que des régions gérant plusieurs dimensions (discuté en [3]).

RAPPORT # 3 : Modèles mathématiques en hydrodynamique fluviale

Étude des modèles mathématiques en hydrodynamique fluviale

Rapport intermédiaire Janvier 2006

Rapport intermédiaire

Étude des modèles mathématiques en hydrodynamique fluviale

Présenté par
Maude Giasson

12 janvier 2006

Résumé

Diverses simplifications des équations de Navier-Stokes peuvent avoir lieu dans le but d'obtenir un système à résoudre numériquement. Parmi ces simplifications, notons la moyenne de Reynolds, permettant d'éliminer la nature instantanée du système, l'hypothèse hydrostatique, supposant que la pression varie linéairement avec la profondeur, et l'intégration sur la verticale ou l'horizontale, permettant de considérer le problème de manière bidimensionnelle ou unidimensionnelle.

Le présent travail se concentre sur les modèles mathématiques représentant l'écoulement au sein des modèles hydrodynamiques. Dans un premier temps, les systèmes d'équations utilisés par différents auteurs seront étudiés et comparés et ce, de manière à faire ressortir les différentes hypothèses posées lors des simplifications. Par la suite, une dérivation complète, menant des équations de Navier-Stokes à un système de St-Venant $2D$, sera reprise en détail ce qui permettra de mettre en lumière les différentes hypothèses posées lors d'une telle dérivation ainsi que des équivalences et différences avec d'autres dérivations.

Table des matières

Table des figures

Notation

α_{ij}	Terme de dispersion
c_w	Coefficient de traînée du vent
$\overline{d}$	Diamètre moyen des grains
g	Gravité
h	Niveau d'eau
h'	Cote de fond
H	Profondeur
η	Coefficent de Manning (associé à la rugosité)
η_m	Coefficent des macrophytes
η_g	Coefficent de frottement de la glace
η_{tot}	Coefficient de frottement global
i, j	Représentent les sens, en notation d'Einstein
l_m	Longueur de mélange
ν_T	Viscosité turbulente
p	Pression
p_a	Pression atmosphérique
r_{ij}	Contraintes de Reynolds
R_{ij}	Contraintes de Reynolds moyennées sur la colonne d'eau
ρ	Densité de l'eau
ρ_a	Densité de l'air
T_{ij}	Contraintes visqueuses moyennées sur la colonne d'eau
T	Petit intervalle de temps
τ_{ij}	Contraintes visqueuses
τ_{iz}^s, τ_{iz}^f	Contraintes de surface et de fond
$\tau_{iz}^{s;w}$, $\tau_{iz}^{s;g}$	Contraintes de surface liées au vent et à la glace
$\tau_{iz}^{f;r}$, $\tau_{iz}^{f;m}$	Contraintes au fond liées à la rugosité et aux macrophytes
u, v, w	Composantes de la vitesse
W	Partie moyennée sur la colonne d'eau des composantes de la vitesse
$\mathbf{w}$	Vecteur de vitesse du vent

w_i	Composantes de la vitesse du vent
x, y, z	Composantes d'un plan cartésien

De manière générale, les valeurs en majuscules sont des moyennes sur la verticale des valeurs en minuscules. La hauteur d'eau h et la profondeur H font exception à cette règle.

Chapitre 1

Introduction

La connaissance des qualités hydrodynamiques d'un cours d'eau s'avère essentielle pour mener à bien des problèmes d'ingénierie, comme, par exemple, ceux concernant la gestion des ressources hydroélectriques, la construction d'infrastructure et le transport maritime. Mais les utilisations des informations fournies par les modèles hydrodynamiques ne s'arrêtent pas à l'ingénierie. En particulier, la profondeur et la température peuvent être déterminantes en ce qui concerne l'habitat pour diverses espèces. De la même manière, en aménagement du territoire, la connaissance hydrodynamique d'un cours d'eau, couplée à des données statistiques, permet de juger des risques d'inondation. Les résultats obtenus à partir d'un modèle hydrodynamique servent d'entrées importantes pour différents modèles, tant en sédimentologie et en hydrogéologie qu'en écologie et en aménagement. Il s'agit d'un domaine actuel où les nouvelles avancées théoriques passent rapidement vers des équipes pratiques.

Vu de l'extérieur, les différents modèles hydrodynamiques se ressemblent. Il s'agit d'applications permettant d'obtenir le débit et le niveau d'eau sur le domaine considéré et ce, moyennant des connaissances sur le terrain, telles la topographie, la rugosité du fond et des conditions aux frontières. La manière d'obtenir les résultats varie toutefois d'un modèle à l'autre selon les simplifications et les méthodes de discrétisation choisies. Dans ce travail, nous nous intéressons aux modèles mathématiques, donc aux simplifications du problème plutôt qu'à sa résolution sous forme discrète.

Dans un contexte où les ressources informatiques sont finies, les simplifications sont appliquées afin de résoudre le problème considéré dans un temps raisonnable. Ces simplifications permettent d'accélérer la résolution en diminuant le nombre

de variables et d'équations à considérer. À titre d'exemple, dans le cadre des problématiques d'inondations, dont le niveau d'eau est la variable d'intérêt, une moyenne sur la verticale des variables peu avoir lieu sans perte significative de qualité dans l'information. De telles simplifications ont plus d'impact lorsqu'on s'intéresse non pas à des problèmes d'inondations, mais plutôt à des problèmes d'habitat de poisson, où les vitesses au fond peuvent alors devenir déterminantes. Les hypothèses posées lors du développement d'un modèle ont donc un impact majeur sur les applications qui pourront en suivre, c'est pourquoi il est intéressant de comparer les différents modèles au niveau des hypothèses posées lors du développement mathématique.

1.1 Situer le modèle mathématique

En hydrodynamique, on s'intéresse aux lois régissant le mouvement des liquides ainsi qu'aux résistances qui s'opposent à ce mouvement. Les équations de Navier Stokes, décrivant ces phénomènes, sont bien connues, mais leur résolution analytique reste limitée à des écoulements extrêmement simples, d'où l'intérêt d'apporter des solutions numériques. Cet intérêt est d'autant plus grand que, dans les dernières décennies, les capacités informatiques, tant en ce qui concerne la mémoire que la rapidité d'exécution, ont augmentés rapidement et continuent encore d'augmenter, rendant possible des résolutions de plus en plus fines.

Les équations de Navier Stokes décrivent l'écoulement d'un fluide compressible évoluant dans un domaine $3D$. Selon le problème étudié, le système d'équation sera modifié avant de passer à la résolution numérique. Le fluide (eau dans notre cas) sera considéré newtonien et incompressible. L'écoulement étant irrégulier dans le temps, une échelle de base sera définie et l'influence des énergies turbulentes de plus petites échelle sera paramétrée. Dans certains, une intégration ou moyenne horizontale et/ou verticale aura lieu, menant ainsi à un modèle bidimensionnel ou unidimensionnel selon le cas. L'hypothèse de pression hydrostatique peut aussi être posée, de manière à diminuer le nombre d'inconnue, en reliant la pression en un point à la hauteur d'eau au dessus de ce point. Toutes ces simplifications sont choisies en fonction du problème considéré et permettent de construire un nouveau système d'équation formant le modèle mathématique à partir duquel une solution numérique sera apportée.

Une fois le modèle mathématique établit, le domaine doit être discrétisé. Cela peut se faire, entre autres, par différences finies, par volumes finis ou par éléments finis. Quelle que soit la méthode choisie, elle permet de transformer le modèle

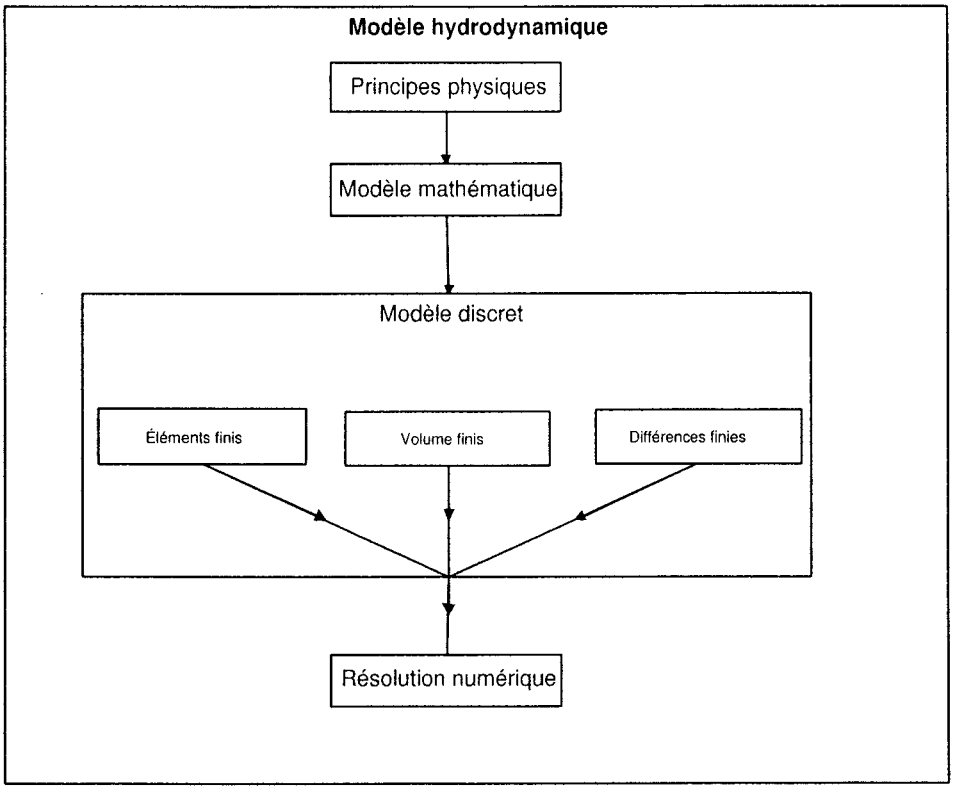


FIG. 1.1 – Conception d'un modèle

mathématique en un système algébrique que nous appelons ici le modèle discret.

Le modèle discret doit ensuite être résolu. Il s'agit en fait de résoudre un système algébrique et cela implique le recourt à des méthodes numériques.

Ce travail s'intéresse aux différents modèles mathématiques pouvant être utilisés au sein d'un modèle hydrodynamique. La 1.1 décrit les principales étapes de la conception d'un modèle hydrodynamique et, plus particulièrement, la position du modèle mathématique, principal sujet de ce travail, dans cette structure.

1.2 Buts

Ce travail étudie les modèles mathématiques utilisés dans l'élaboration d'un modèle capable de représenter l'hydrodynamique fluviale. Les deux principaux volets de ce travail sont l'étude de modèles existant et le développement détaillé d'un modèle.

L'étude porte d'abord sur les modèles développés par une sélection d'auteurs. Le travail se concentre non seulement les modèles mathématiques utilisés par ces derniers, mais particulièrement surla séquences d'hypothèses posées pour parvenir aux modèles en question. L'objectif principal de cette section est d'établir la portée de chaque hypothèse et les équivalences entre certaines séquences d'hypothèses et, par le fait même, entre certains modèles.

Le second aspect de ce travail concerne le développement détaillé d'un modèle mathématique. Avec comme seul point de départ les principes de conservation de base, une série d'hypothèses seront posées et détaillés de manière à obtenir les équations de Navier Stokes, puis, suite à une intégration verticale, des équations bidimensionnelles. On étudira les hypothèses devant être posées pour obtenir un système de Saint-Venant standard.

Les buts se résument ainsi :

1. Étudier une sélection de modèles mathématiques, comparer les séquences d'hypothèses posées
2. Détailler le développement d'un modèle de Saint-Venant

Chapitre 2

Étude bibliographique

Le sujet à l'étude est un écoulement tridimensionnel, dépendant du temps, où l'une des frontières du domaine est délimitée par une surface libre. Les équations de Navier Stokes décrivent ces écoulements. Dans la majorité des modèles hydrodynamiques, (p. ex. CASULLI et STELLING (1998), FERRARI et SALERI (2004), HENICHE ET AL. (2002), STELLING et BUSNELLI (2001) et TOME et MCKEE (1994)) le fluide, de l'eau en occurrence, est considéré incompressible, ce qui se laisse représenter par les équations de Navier Stokes pour les fluides incompressibles. Devant l'impossibilité de résoudre analytiquement ces équations pour des problèmes non triviaux, il faut chercher des solutions numériques. Or, la plupart des écoulements d'intérêt étant turbulents, une version moyennée sur la turbulence est utilisée, ce qui mène aux équations dites *RANS* pour *Reynolds-Averaged Navier-Stokes equation*, où les variables sont décomposées en une somme de deux termes, soit une valeur moyenne sur une échelle de temps macroscopique et la variation par rapport à cette moyenne. L'influence de l'agitation turbulente sur l'écoulement moyen est représentée à l'aide d'un terme turbulent qui, associé à de la viscosité, doit être paramétré. Il existe différents modèles de turbulence pour des problèmes tridimensionnels, dont ceux présentés dans RODI (1984). Le présent travail, ne s'attarde pas aux modèles de turbulence. Toutefois, après avoir dérivé un modèle $2D$ on appliquera un modèle de turbulence similaire à ceux existant en $3D$.

Selon ALCRUDO (2002), les solutions complètes tridimensionnelles sont limitées à des écoulements lents ou permanents (p. ex. SINHA ET AL. (1998), CASULLI et STELLING (1998) et YE et MCCORQUODALE (1998)). Plus récemment, d'autres algorithmes pour résoudre les équations de Navier Stokes $3D$ non hydrostatique avec surface libre ont été présentés (p. ex. YUAN et WU (2004), KOÇYIGIT ET AL.

(2002)), mais dans ces modèles la formulation mathématique a peu d'intérêt, l'attention se portant plutôt sur la formulation du modèle discret et sa résolution numérique. Notons toutefois que KOÇYIGIT ET AL. (2002) séparent la pression en une composante hydrostatique et une non hydrostatique et solutionnent le problème dans un système en coordonnées sigma, c'est à dire un système transformant la coordonnée verticale en σ variant sur $[-1, 0]$ entre le fond et la surface libre.

Lorsque les échelles horizontales du domaine sont d'un ordre supérieur aux échelles verticales, le problème peut être simplifié en posant l'hypothèse hydrostatique. LELERC (1985) et MIGLIO ET AL. (1999) présentent des modèles tridimensionnels, utilisant plusieurs couches sur la verticale où seule l'équation de conservation de la masse est modifiée et ce, par une intégration sur la verticale. Aucune équation supplémentaire n'est nécessaire car la vitesse verticale, considérée constante au départ, est ensuite récupérée en post-traitement. HENICHE (1995) propose une approche similaire mais ajoute un terme représentant la pression due aux sédiments dans la colonne d'eau. Les modèles tridimensionnels demeurent coûteux (en temps informatique et en espace mémoire) en particulier parce qu'il est nécessaire d'adapter la discrétisation du domaine au mouvement de la surface libre (AUDUSSE (2005)). De tels modèles s'avèrent toutefois nécessaires pour résoudre certains problèmes, par exemple, pour des cas de dispersion de contaminant (e.g. BUIL (1999)) ou d'autre avec des recirculations importantes (p. ex. LELERC (1985)).

Les modèles bidimensionnels horizontaux, plus particulièrement ceux posant l'hypothèse de pression hydrostatique, sont largement utilisés. Ils ont l'avantage important d'alléger la discrétisation du domaine de calcul, ce dernier ne se déplaçant plus avec la surface libre. Ils conviennent bien aux problèmes où la variable d'intérêt est la hauteur d'eau, qui est elle-même une variable bidimensionnelle. La formulation la mieux connue en $2D$ provient de SAINT-VENANT (1871), où elle directement dérivée des équations de conservation de la masse et du mouvement dans le plan (ALCRUDO (2002). Les équations de St-Venant posent l'hypothèse hydrostatique et les vitesses sont remplacées par leur valeur moyenne sur la verticale. Des systèmes similaires sont obtenus à partir des équations Navier Stoke en supposant que les vitesses horizontales ne varient pas sur la verticale (e.g. HenicheAl2002) ou par une intégration des équations sur la verticale (e.g. HEROUIN (1991), PAQUIER (1995), SECRETAN et DUBOS (2005), ZHANG (1992)) . Lors de l'intégration verticale, un terme représentant la différence entre la vitesse moyenne et la vitesse réelle apparaît. Il s'agit du terme de dispersion. Différentes solutions pour paramétrer ce terme sont étudiées (p.ex. SECRETAN et DUBOS (2005)), mais pour obtenir un système équivalent au système de St-Venant, il

faut supposer que l'intégration sur la verticale de cette différence est nulle et faire certaines hypothèses concernant les contraintes en surface et au fond. Nous ferons cette démonstration dans le présent document. Les applications des modèles bidimensionnels de type St-Venant sont nombreuses. À titre d'exemple, un ambitieux projet de modélisation du St-Laurent utilisant HYDROSIM est actuellement en cours.

Dans les modèles bidimensionnels obtenus par une intégration sur la verticale, le terme représentant la turbulence (ou la viscosité) n'est pas intégrée mais sa valeur moyenne est paramétrée. FERRARI et SALERI (2004) ont développé à nouveau des équations de type Saint-Venant à partir de Navier Stokes, en conservant les termes de viscosités provenant du système tridimensionnel. Cela permet en particulier de comparer le modèle $2D$ obtenu à un modèle $3D$, la fermeture turbulente étant elle aussi intégrée sur la verticale. Pour analyser le terme de dispersion ainsi que la pression moyenne sur la verticale, FERRARI et SALERI (2004) se basent sur des équations aux dimensions. Ce travail est une extension de celui de GERBEAU et PERTHAME (2001). Ces derniers ne considèrent pas la variation de topographie (ce qui mène à une situation où l'hypothèse hydrostatique est applicable) et travaillent principalement en $1D$. FERRARI et SALERI (2004) introduisent pour leur part de faibles pentes et conservent des termes représentant la pression. Les dimensions verticales sont supposées inférieures aux dimensions longitudinales - ce qui n'est pas sans rappeler l'hypothèse hydrostatique - et c'est par rapport à ce ratio qu'ils obtiennent un système bidimensionnel approximant au second ordre le problème tridimensionnel.

STEFFLER et GHAMRY (2002) introduisent aussi un système de type Saint-Venant. Or, ils ne supposent pas les vitesses uniformes sur la verticale, ni la pression hydrostatique. À partir d'un système de Saint-Venant standard, ils remplacent les vitesses horizontales par une équation linéaire alors que la pression et la vitesse verticale sont remplacées par des équations quadratiques. L'ajout de variable supplémentaire au problème est compensé par des équations de moments elles aussi jointes au modèle mathématique, ce qui porte le nombre d'équations à 10 tout comme le nombre de variables sur lesquels s'effectuent le calcul.

Il existe aussi des modèles bidimensionnels verticaux et des modèles unidimentionnels. Ces derniers sont très fréqents et trouvent de nombreuses applications, mais ce sujet n'a pas été étudié.

La figure 2 résume principales hypothèses et traitements (rectangles) menant à l'obtentions de différents modèles (losanges) étudiés dans ce chapitre.

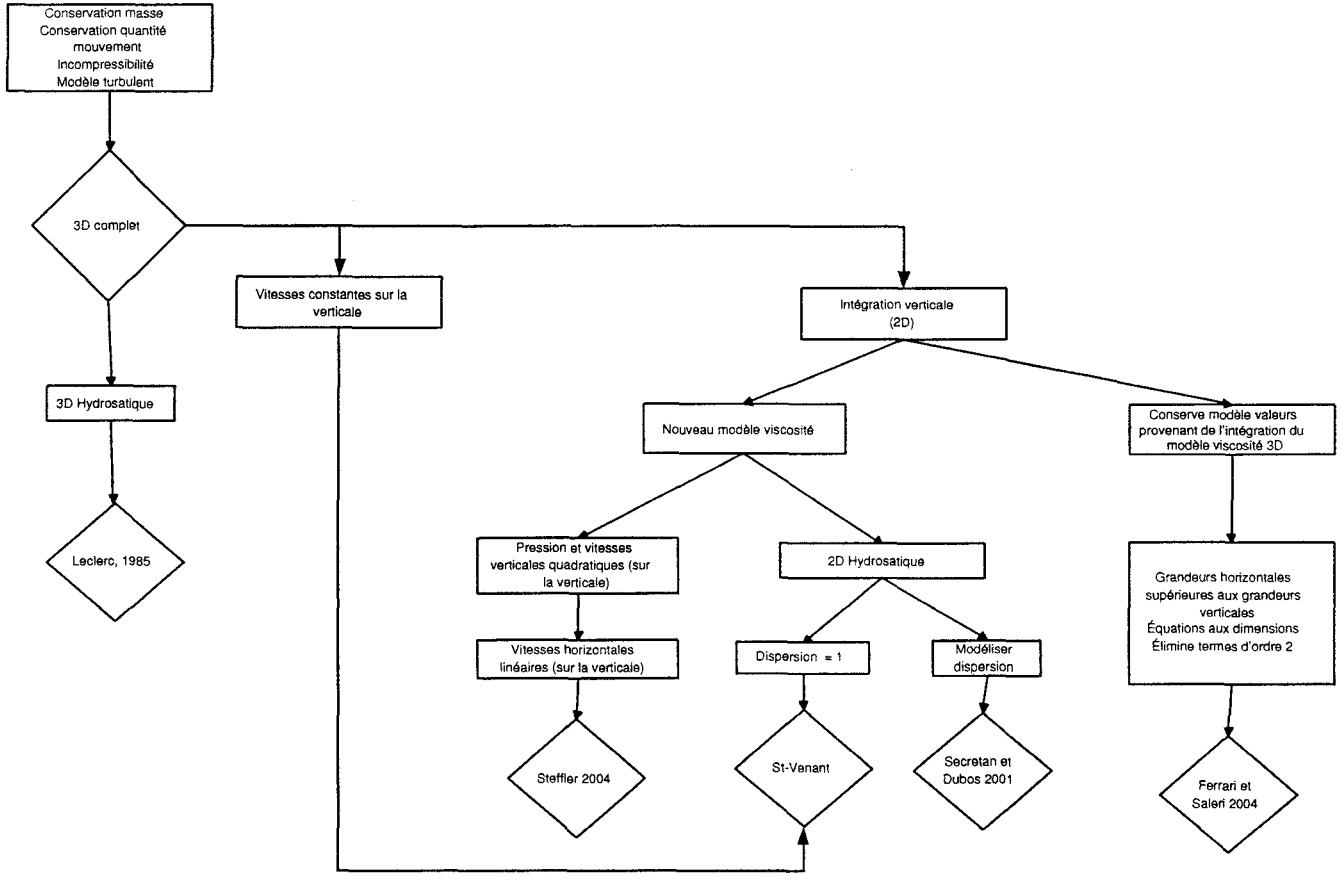


FIG. 2.1 – Principales hypothèses et modèles associés

Chapitre 3

Développement d'un modèle mathématique

Ce chapitre reprend en détail le développement d'un modèle mathématique de Saint-Venant en effectuant une intégration verticale des équations de Navier Stokes. Lors de l'intégration, on verra en particulier apparaître le terme de dispersion, pour ensuite le poser constant et obtenir le système souhaité. Des fermetures seront aussi présentées. La figure 3 présente l'ordre dans lequel les hypothèses sont posées dans ce chapitre et les systèmes d'équations obtenus avant d'en arriver au modèle mathématique de Saint-Venant.

Le point de départ sera le système 3.1 représentant les équations de Navier Stokes pour un fluide incompressible. On a aussi effectué une moyenne de Reynolds (turbulente) sur ces équations.

$$\begin{aligned} \frac{\partial u_j}{\partial x_j} &= 0 \\ \frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} &= \frac{1}{\rho}\left(f_i - \frac{\partial p}{\partial x_i} + \frac{\partial\left(\tau_{ij} + r_{ij}\right)}{\partial x_i}\right) \end{aligned} \tag{3.1}$$

où u_i est la variable de vitesse, ρ est la densité (constante), f_i la somme des forces extérieures (comme la force de Coriolis ou le vent par exemple), p est la variable de pression et finalement τ_{ij} et r_{ij} forment le tenseur des contraintes, le premier pour la viscosité, le second pour les contraintes turbulentes.

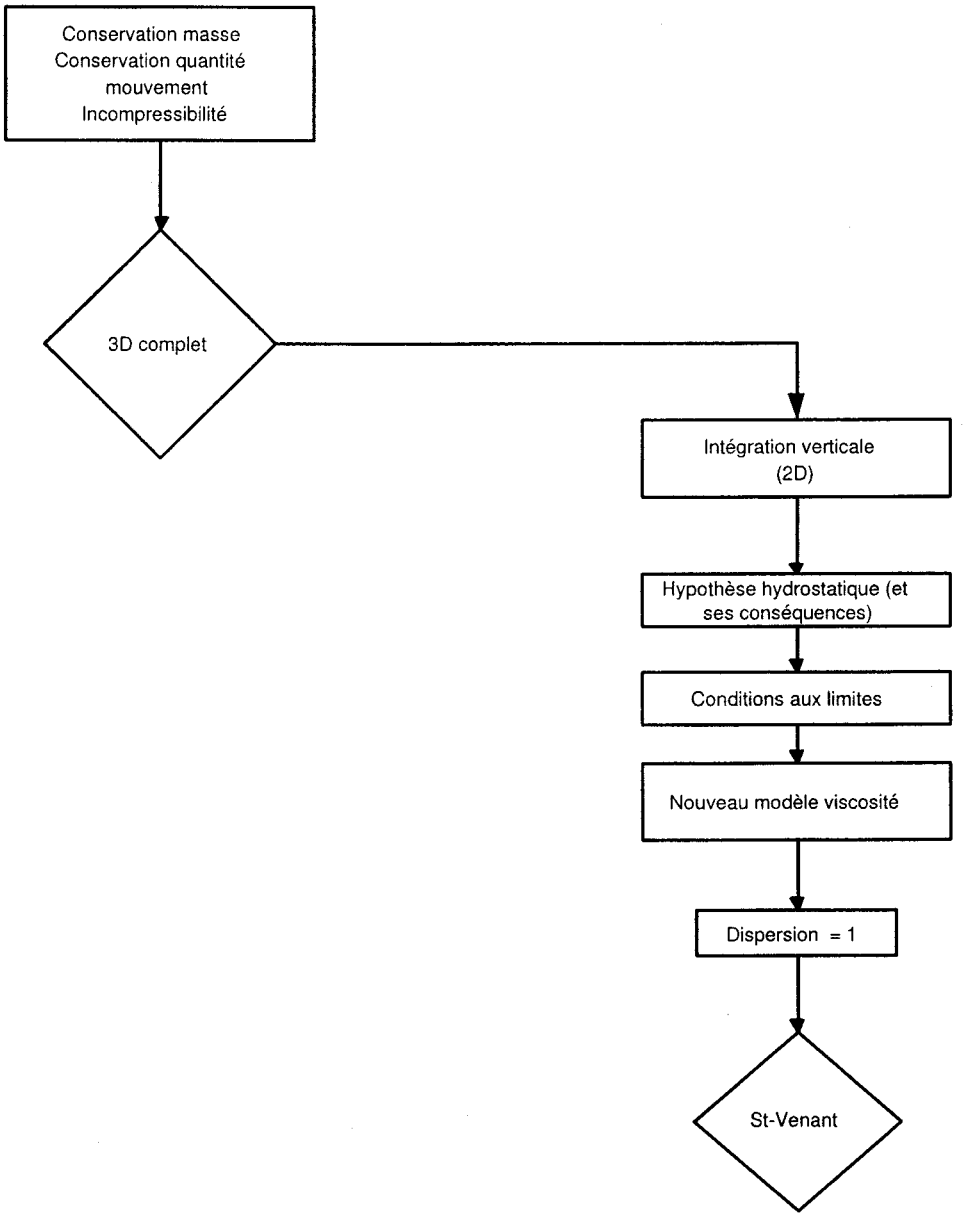


FIG. 3.1 – Séquence d'hypothèses posées dans ce chapitre

3.1 Hypothèse hydrostatique

Nous poserons l'hypothèse hydrostatique, laquelle se formule ainsi :

> *En tout point du domaine, la pression ne dépend que de la hauteur de la colonne de fluide située au dessus de ce point*

ou, de manière mathématique :

$$p = \rho g(h - z) + p_a \tag{3.2}$$

où p_a est la pression atmosphérique. Il est à noter que l'hypothèse hydrostatique implique une pression atmosphérique constante sur le domaine.

Pour que l'hypothèse hydrostatique soit applicable, il faut en particulier que l'accélération verticale soit nulle ou négligeable. Ainsi, les domaines ayant de fortes pentes seront difficilement modélisés. En dérivant 3.2 on obtient une nouvelle formulation pour l'équation de la conservation de la quantité de mouvement en z, où l'on note, en comparant à 3.1 que les accélérations verticales sont bien négligeables sous l'hypothèse hydrostatique :

$$\frac{\partial p}{\rho \partial z} = -g \tag{3.3}$$

L'expression 3.2 peut aussi être dérivée selon x et selon y de manière à obtenir :

$$\frac{1}{\rho}\frac{\partial p}{\partial x_i} = g\frac{\partial h}{\partial x_i} \tag{3.4}$$

i ne représentant dorénavant que deux dimensions. En appliquant ce résultat à (3.1) et en laissant tomber l'équation en z (implicitement incluse dans le système) on obtient :

$$\begin{aligned} \frac{\partial u_j}{\partial x_j} &= 0 \\ \frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} &= -g\frac{\partial h}{\partial x_i} + \frac{1}{\rho}\left(f_i + \frac{\partial(\tau + r)}{\partial x_j}\right) \end{aligned} \tag{3.5}$$

Le système ainsi obtenu représente un système $3D$ hydrostatique.

3.2 Integration verticale

L'intégration verticale a pour objectif de passer d'une représentation $3D$ à une représentation $2D$ en éliminant la dépendance à z. Il est aussi possible de

supposer l'uniformité des vitesses horizontales par rapport à la verticale, ce qui mène directement à un système de Saint-Venant. Ici l'intégration aura lieu sans supposer l'uniformité de la vitesse sur la verticale, ce qui, dans un premier temps, ne conduira pas tout à fait au même système. On retrouvera un système de Saint-Venant, mais seulement après avoir ajouté certaines conditions. L'intérêt d'un tel détour, est de séparer l'hypothèse d'uniformité des vitesses horizontales en un ensemble d'hypothèses pouvant être étudiées individuellement.

Commençons tout d'abord par introduire les conditions cinématiques de fond et de surface. Ce sont des conditions limites. Au fond, on considère qu'il n'y a pas d'écoulement normal au lit. En surface, on impose la condition naturelle que l'eau suit la surface libre. Ces conditions permettent d'obtenir des équations définissant w en surface et seront utiles pour simplifier les équations obtenues lors de l'intégration verticale. Par définition, on a que $w = \frac{dz}{dt}$. Cette définition est utilisée pour obtenir, $w|_h = \frac{dh}{dt}$ et $w|_{h'} = \frac{dh'}{dt}$. En calculant les dérivées totales dh et dh', on obtient :

$$w(h) = \frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y} \tag{3.6}$$

et

$$w(h') = \frac{\partial h'}{\partial t} + u\frac{\partial h'}{\partial x} + v\frac{\partial h'}{\partial y} \tag{3.7}$$

L'égalité en (3.6) est connue sous le nom de condition cinématique de surface alors que celle en (3.7) se nomme la condition cinématique de fond. Dans notre modèle, on suppose un fond fixe, c'est à dire que $\frac{\partial h'}{\partial t} = 0$. Ceci simplifie la condition cinématique de fond.

Procédons maintenant à l'intégration verticale. Il s'agit ici d'intégrer (3.5) entre le fond (h') et la surface (h). À noter que h' et h sont des fonctions dépendant de la position (x,y) et du temps (t). De manière générale, les termes contenant des dérivées partielles par rapport à x et y seront intégrés à l'aide de la formule de Leibnitz (A.1) alors que ceux contenant des dérivées par rapport à z seront intégrés directement. Les termes seront simplifiés à l'aide des conditions cinématiques (3.7) et (3.6). La notion de moyenne intervient aussi dans la résolution. Par exemple, U représente la vitesse moyenne sur la verticale :

$$\frac{1}{H}\int_{h'}^{h} U dz = U \tag{3.8}$$

Soit $\varepsilon_i(z)$ tel que $U_i\varepsilon_i(z)$ est la différence entre la variable u_i au point z et sa moyenne sur la colonne, on peut écrire :

$$u_i(z) = U_i(1 + \varepsilon_i(z)) \tag{3.9}$$

On aura alors :

$$\begin{aligned}\frac{\partial}{\partial x_j}\left[\int_{h'}^{h} u_i\, u_j dz\right] &= \frac{\partial}{\partial x_j}\left[U_i\, U_j \int_{h'}^{h}(1+\varepsilon_i+\varepsilon_j+\varepsilon_i\varepsilon_j)dz\right] \\ &= \frac{\partial}{\partial x_j}\left[U_i\, U_j H(1+\frac{1}{H}\int_{h'}^{h}(\varepsilon_i\varepsilon_j)dz\right] \qquad (3.10)\end{aligned}$$

Le terme $\frac{1}{H}\int_{h'}^{h}(\varepsilon_i\varepsilon_j)dz$ est souvent supposé nul, ce qui n'est pas nécessairement le cas. On peut facilement imaginer que les vitesses en x et en y sont toutes deux inférieure à leur moyenne pour des petits z et supérieures à leur moyenne pour des grands z. Ainsi, l'intégrale du produit donne un terme positif, parfois considéré (résultat empirique) comme valant $0,1$. En réalité, il s'agit bel et bien d'une variable selon x, y et t. En posant $\alpha_{ij} = (1+\frac{1}{H}\int_{h'}^{h}(\varepsilon_i\varepsilon_j)dz)$ on obtient le **terme de dispersion**, que nous conservons pour l'instant, quoi qu'il sera plus loin supposé constant ($=1$).

En utilisant le fait que le fond n'est pas mobile dans le temps, la définition de la moyenne verticale et du terme de dispersion et les conditions aux limites présentées plus haut, l'intégration donne le résultat suivant, dans lequel i et j valent 1 et 2 :

$$\begin{aligned}\frac{\partial h}{\partial t}+\frac{\partial HU_j}{\partial x_j} &= 0 \\ \rho\left(\frac{\partial HU_i}{\partial t}+\frac{\partial}{\partial x_j}\left(U_i\, U_j H\alpha_{ij}\right)\right) &= -\rho g H\frac{\partial h}{\partial x_i}+F_i H+\frac{\partial\left(T_{ij}+R_{ij}\right)}{\partial x_j} \\ &\quad -\left(\tau_{ij}|_h+r_{ij}|_h\right)\frac{\partial h}{\partial x_j} \\ &\quad +\left(\tau_{ij}|_{h'}+r_{ij}|_{h'}\right)\frac{\partial h'}{\partial x_j} \\ &\quad +\tau^s_{iz}-\tau^f_{iz} \qquad (3.11)\end{aligned}$$

Ici, les contraintes verticales en surface sont représentées par $\tau^s_{iz} = \tau_{iz}|_h + r_{iz}|_h$ alors que celles au fond sont $\tau^f_{iz} = \tau_{iz}|_{h'} + r_{iz}|_{h'}$. Finalement les lettres majuscules T et R représentent respectivement les moyennes verticales de τ et r.

On a déjà mentionné que sous l'hypothèse hydrostatique, les pentes au fond doivent être faibles (étant donné que les accélérations verticales sont négligeables). Ainsi, on suppose les contraintes horizontales négligeables afin de simplifier 3.21 (DUBOS (2001)).

Pour la suite, on regroupe la somme $(T_{ij}+R_{ij})$ sous un seul terme, noté R_{ij}, qui sera fixé à l'aide des fermetures (voir section 3.3). Ainsi, même si l'on présente ici une moyenne pour le tenseur des contrainte visqueuses et turbulentes à

partir de leur valeur dans le modèle $3D$, il est à noter qu'ils seront paramétrés directement à partir du modèle $2D$. Ainsi, selon la fermeture choisie, on risque de perdre l'équivalence avec le modèle $3D$. FERRARI et SALERI (2004) présentent une déduction différente d'un modèle de type Saint-Venant à partir des équations de Navier Stokes où les termes de viscosité sont conservés. Dans ce cas, la comparaison du modèle développé avec un modèle $3D$ est plus simple.

En posant le terme de dispersion constant (valant 1), et en appliquant les simplifications mentionnées dans les paragraphes précédents au modèle 3.21, on obtient :

$$\begin{aligned} \frac{\partial h}{\partial t} + \frac{\partial H U_j}{\partial x_j} &= 0 \\ \frac{\partial H U_i}{\partial t} + \frac{\partial}{\partial x_j}(U_i\, U_j H) &= -gH\frac{\partial h}{\partial x_i} + \frac{F_i H}{\rho} + \frac{1}{\rho}\frac{\partial R_{ij}}{\partial x_j} \\ &\quad + \tau^s_{iz} - \tau^f_{iz} \end{aligned} \tag{3.12}$$

3.3 Fermetures

Le système (3.12) comporte de nombreuses variables. À terme, seules les vitesses horizontales et la hauteur d'eau seront conservées. Il faut donc mettre en relation les termes turbulents, les lois de comportement à la surface et au fond et les forces extérieures avec les 3 variables de base du modèle. Les fermetures s'effectuent donc sur R_{ij}, τ^s_{iz}, τ^f_{iz} et F_i.

Évaluer les termes turbulents ($2D$)

Nous avons déjà vu que les contraintes de Reynolds (r_{ij}), expriment la contribution des turbulences aux valeurs moyennes. Ici, on cherchera à paramétrer les contraintes de Reynolds moyennées sur la colonne d'eau R_{ij}. Pour ce faire, on se base sur ce qui est fait en $3D$. Toutefois, il aurait été préférable de poser d'abord le modèle turbulent (en $3D$), lequel aurait pu être lui aussi intégré sur la verticale. C'est un peu ce qu'ont fait FERRARI et SALERI (2004) et nous ne reprendrons pas leur travail ici. Notons toutefois que le système qu'ils obtiennent est légèrement différent d'un système de Saint-Venant $2D$ standard.

À l'image de l'hypothèse de Boussinesq, valide pour les modèles $3D$, on relie les tensions de Reynolds et le tenseur des contraintes à la viscosité turbulente ν_T

de la manière suivante :

$$R_{ij} = \nu_T \left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i} \right) \tag{3.13}$$

À priori, ν_T varie dans l'espace et dans le temps. Il faut maintenant relier cette variable aux autres déjà présentes dans le système. Dans les modèles $3D$, il est possible d'introduire la loi de Prantdl. La formulation suivante peut être choisie :

$$\nu_T = l_m^2 \sqrt{2\left(\frac{\partial U}{\partial x}\right)^2 + 2\left(\frac{\partial V}{\partial y}\right)^2 + \left(\frac{\partial U}{\partial y} + \frac{\partial V}{\partial x}\right)^2} \tag{3.14}$$

Lois de comportement à la surface et au fond

Des lois de comportement doivent être établies pour représenter les contraintes à la surface et au fond. Au fond, on retrouve la résistance des plantes aquatiques ($\tau_{xz}^{f;m}$ et $\tau_{yz}^{f;m}$), et celle due à la rugosité ($\tau_{xz}^{f;r}$ et $\tau_{yz}^{f;r}$). En surface se trouvent les contraintes du vent ($\tau_{xz}^{s;w}$ et $\tau_{yz}^{s;w}$) et la résistance de la glace ($\tau_{xz}^{s;g}$ et $\tau_{yz}^{s;g}$).

Contraintes du vent ($\tau_{iz}^{s;w}$) On utilise une loi de traînée telle la suivante :

$$\tau_{iz}^{s;w} = \rho_a c_w |\mathbf{w}| w_i \tag{3.15}$$

où

- ρ_a est la densité de l'air
- c_w le coefficient de traînée
- w_i la composante en (i) du vent
- $|\mathbf{w}|$ le module de la vitesse du vent.

Notons que c_w est fonction de la rugosité de la surface (les vagues). Il est donc lui-même fonction du vent. Pour sa part, le vent $(\mathbf{w})$ est mesuré à une hauteur de représentation, soit environ à 10 mètres.

Contraintes du lit ($\tau_{iz}^{f;r}$) On exprime cette contrainte à l'aide d'une loi de Manning, aussi nommée loi de Chézy, ou Strickler de la manière suivante :

$$\tau_{iz}^{f;r} = \frac{1}{H^{1/3}} \rho g \eta^2 |\mathbf{u}| u_i \tag{3.16}$$

Dans cette équation, le terme η représente le coefficient de Manning, établit en fonction du diamètre des grains. C'est en fait un paramètre relié à la rugosité. Le $H^{1/3}$ a été obtenu de manière empirique.

Établir le η de Manning Classiquement, ce coefficient est utilisé dans les modèles unidimensionnels pour représenter à la fois le frottement global de la section et les accélérations convectives. En 2D, son rôle est réduit. Il s'agit d'une propriété locale ($f(x,y)$) strictement réservée au frottement et dépendant de la taille du substrat. Dans le modèle développé, la formule suivante est utilisée :

$$\frac{1}{\eta(x,y)} = 19.8 \log\left[\frac{9.15}{\overline{d}(x,y)}\right] \tag{3.17}$$

où $\overline{d}(x,y)$ est le diamètre moyen des grains.

Contraintes des macrophytes ($\tau_{iz}^{f;m}$) On emploie également une loi de Manning pour décrire cette contrainte :

$$\tau_{iz}^{f;m} = \frac{1}{H^{1/3}} \rho g \eta_m^2 |\mathbf{u}| u_i \tag{3.18}$$

où η_m est relié aux espèces présentes, leur densité spatiale et leur stade de croissance. C'est donc une variable dans l'espace et dans le temps : $\eta_m(x,y,t)$.

Contraintes de la glace ($\tau_{iz}^{s;g}$) On utilise une loi de Manning avec η_g.

Somme des contraintes Étant donné que trois des contraintes répondent à la même loi, on peut poser $\eta_{tot} = \sqrt{\eta_g^2 + \eta_m^2 + \eta^2}$ et obtenir une loi globale représentant les contraintes au fond et à la surface :

$$\frac{1}{\rho}\left(\tau_{iz}^{s} - \tau_{iz}^{f}\right) = \frac{\rho_a}{\rho} c_w |\mathbf{w}| w_i + \frac{1}{H^{1/3}} g(\eta_{tot}^2) |\mathbf{u}| u_i \tag{3.19}$$

Forces extérieures Comme force extérieure, on ne retrouve que la force de Coriolis. On peut poser $\mathbf{F} = \left\{ \begin{array}{c} 2\omega sin(\phi) v \\ -2\omega sin(\phi) u \end{array} \right\}$, où ω représente la vitesse angulaire de rotation de la terre et ϕ la latitude. Toutefois, dans plusieurs problèmes, dont ceux d'inondation (ALCRUDO (2002)), la force de Coriolis a un effet minimal et peut être supposée négligeable.

3.4 Modèle de Saint-Venant

En appliquant la fermeture turbulente 3.3 au modèle bidimensionnel 3.12, on obtient un système de Saint-Venant $2D$ (où les forces de Coriolis sont supposées nulles) :

$$\begin{aligned} \frac{\partial h}{\partial t} + \frac{\partial HU_j}{\partial x_j} &= 0 \\ \frac{\partial HU_i}{\partial t} + \frac{\partial}{\partial x_j}\left(U_i\, U_j H\right) &= -gH\frac{\partial h}{\partial x_i} + \frac{1}{\rho}\frac{\partial\left(\nu_T\left(\frac{\partial U_i}{\partial x_j} + \frac{\partial U_j}{\partial x_i}\right)\right)}{\partial x_j} \\ &\quad +\tau_{iz}^s - \tau_{iz}^f \end{aligned} \tag{3.20}$$

On a donc obtenu un système de Saint-Venant par une intégration sur la verticale des équations de Navier Stokes.

La forme non-conservative est obtenue en utilisant l'équation de conservation de la masse dans les équations de continuité :

$$\begin{aligned} \frac{\partial h}{\partial t} + \frac{\partial HU_j}{\partial x_j} &= 0 \\ H\left(\frac{\partial U_i}{\partial t} + U_j\frac{\partial U_i}{\partial x_j}\right) &= -gH\frac{\partial h}{\partial x_i} + \frac{1}{\rho}F_i H + \frac{1}{\rho}\frac{\partial\left(R_{ij}\right)}{\partial x_j} \\ &\quad +\tau_{iz}^s - \tau_{iz}^f \end{aligned} \tag{3.21}$$

Notons que cette forme ne peut être obtenue que sous l'hypothèse $\alpha_{ij} = 1$. Remplacer α, même par une constante, rend la simplification par l'équation de continuité impossible.

Chapitre 4

Conclusion

Un modèle mathématique de Saint-Venant (3.20) a été obtenu à l'aide d'une intégration verticale des équations de Navier Stokes. Les hypothèses suivantes ont dues être posées :

1. Le fluide est incompressible
2. La pression est hydrostatique
3. Conditions cinématiques aux frontières
4. Fond immobile dans le temps
5. Pente au fond faible
6. Contraintes horizontales négligeables au fond et en surface
7. Terme de dispersion constant et vaut 1
8. Force de Coriolis négligeable
9. Modèle turbulent $3D$ s'applique bien à un système $2D$

La première hypothèse est souvent posée, même dans les systèmes tridimensionnels complets. L'hypothèse 8 n'est pas très importante, on peut la poser ou non et on obtient des systèmes similaires. Si l'on intégrait un système tridimensionnel complet, sans faire les autres hypothèses, on devrait alors obtenir un système $2D$ approximant les vitesses moyennes aussi bien qu'un système $3D$, à condition, bien entendu, qu'il soit possible de le résoudre, ce qui serait surprenant. Bien que généralement acceptable, la condition 4 risque de causer problème dans les cas où la sédimentation et le transport sont importants.

Les hypothèses les plus importantes sont sans aucun doute 2, 7 et 9.

En ce qui concerne l'hypothèse hydrostatique, rappelons qu'elle implique une pression atmosphérique constante sur le domaine, ce qui n'est pas toujours le cas,

en particulier pour les grands domaines. Dans ces cas, le modèle peut certainement être sectionné en tronçons. Une autre implication de l'hypothèse hydrostatique est que les accélérations verticales sont négligeables. Ainsi, 5 est une conséquence de 2. On utilise ensuite 5 pour déduire que les contraintes horizontales sont négligeables au fond et en surface (6), ce qui resterait à démontrer.

Poser $\alpha_{ij} = 1$ est plus faible que de supposer les vitesses constantes sur la verticale. En effet, on laisse au produit des vitesses la liberté de varier, mais ces variations doivent se faire de manière symétrique autour de la moyenne. Une autre constante peut être assignée à α ce qui n'amène qu'un coefficient différent à un des termes du modèle conservatif. Un sujet d'étude intéressant consiste à modéliser α_{ij} à partir des autres variables du système, ainsi, on ne se restreint plus à une constante. Dans sa thèse DUBOS (2001) s'est concentrée sur ce sujet. Quoi que certaines améliorations des résultats furent notée, on peut dire, sous toutes réserves, qu'aucun modèle pour α n'est, reconnu de manière générale.

Finalement, on a considéré le modèle de viscosité turbulente. En fait, lorsqu'il est question de faire des parallèles entre le $2D$ et le $3D$, il est intéressant d'avoir un modèle turbulent équivalent, ce qui n'est pas le cas. Ce le serait si le modèle turbulent tridimensionnel avait été intégré sur la verticale. FERRARI et SALERI (2004) ont fait ce travail et obtiennent un modèle bidimensionnel légèrement différent.

Annexe A

Règle de Leibnitz

La règle de Leibnitz s'exprime de la manière suivante :

$$\begin{aligned}\int_{a(x,y,t)}^{b(x,y,t)} \frac{\partial}{\partial x} f(x,y,z)dz &= \frac{\partial}{\partial x}\left(\int_{a(x,y,t)}^{b(x,y,t)} f(x,z)\partial z\right) \\ &\quad -f(x,b(x,y,t))\frac{\partial b(x,y,t)}{\partial x} \\ &\quad +f(x,a(x,y,t))\frac{\partial a(x,y,t)}{\partial x} \qquad \text{(A.1)}\end{aligned}$$

Annexe B

Moyenne de Reynolds

La plupart des variables physiques peuvent être exprimées comme étant la somme d'une variable moyenne sur l'intervalle et d'une variable turbulente. Ainsi, une variable quelconque α variant dans le temps est représentée ainsi :

$$\alpha(t) = \overline{\alpha}(t) + \alpha'(t)$$

où α' varie mais vaut, en moyenne 0 ; $\overline{\alpha}$ est une moyenne mobile. Il s'agit en fait de la moyenne de α sur l'intervalle $[t, t+T]$. Cet intervalle doit être relativement petit. En particulier, lorsqu'on arrivera au numérique, il faudra s'assurer que l'intervalle T est nettement inférieur au pas numérique choisi. La moyenne mobile $\overline{\alpha}$ se définit ainsi :

$$\overline{\alpha}(t) = \frac{1}{T}\int_t^{t+T} \alpha(t)dt$$

Sur chacun des intervalle, on considère la fonction $\overline{\alpha}(t)$ constante. Ainsi on a :

$$\begin{aligned}\overline{\alpha}(t) &= \frac{1}{T}\int_t^{t+T} \alpha(t)dt \\ &= \frac{1}{T}\int_t^{t+T} \overline{\alpha}(t)dt + \frac{1}{T}\int_t^{t+T} \alpha'(t)dt \\ &= \frac{1}{T}\int_t^{t+T} \overline{\alpha}(t)dt \end{aligned} \tag{B.1}$$

C'est à dire que la valeur de $\overline{\alpha}$ au point t est égal à la moyenne que la fonction prendra sur l'intervalle $[t, t+T]$. Nous utiliserons les trois propriétés suivantes :

$$\frac{1}{T}\int_t^{t+T} a\alpha(t)dt = a\overline{\alpha}(t)$$

$$\frac{1}{T}\int_t^{t+T} \frac{\partial}{\partial x_j}\alpha(x_j,t)dt = \frac{\partial}{\partial x_j}\overline{\alpha}(x_j,t) \tag{B.2}$$

$$\frac{1}{T}\int_t^{t+T} \alpha(t)\beta(t)dt = \overline{\alpha}(t)\overline{\beta}(t) + \frac{1}{T}\int_t^{t+T} \alpha'(t)\beta'(t)dt$$

En statistiques, le terme $\frac{1}{T}\int_t^{t+T} \alpha'(t)\beta'(t)dt$ est nommé covariance. Il est alors noté $\overline{\alpha'\beta'}(t)$. C'est cette seconde notation qui sera utilisée pour la suite.

Bibliographie

Alcrudo, F., 2002. *A state of the art review on mathematical modelling of flood propagation.* Firts IMPACT project Workshop, Wallingford.

Audusse, E., 2005. A multilayer Saint-Venant model : Derivation and numerical validation. *Discrete and Continuous Dynamical Systems-Series B*, 5(2) :189–214.

Buil, N., 1999. *Modélisation tridimensionnelle du transport de polluants dans les écoulements à surface libre.* Thèse de doctorat, Université Claude Bernard - Lyon 1.

Casulli, V. et G. S. Stelling, 1998. Numerical simulation of 3D quasi-hydrostatic, free-surface flows. *Journal of Hydraulic Engineering*, 124(7) :678–686.

Cioffi, F., F. Gallerano, et E. Napoli, 2005. Three-dimensional numerical simulation of wind-driven flows in closed channels and basins. *Journal of Hydraulic Research*, 43(3) :290–301.

Dubos, V., 2001. *Validation des vitesses d'un modèles hydrodynamique bidimensionnel ; Prise en compte de la variabilité des profils verticaux des vitesses par un terme de dispersion.* Thèse de maîtrise, Université du Québec. Mémoire.

Faure, J. B., N. Buil, et B. Gay, 2004. 3-D modeling of unsteady free-surface flow in open channel. *Journal of Hydraulic Research*, 42(3) :263–272.

Ferrari, S. et F. Saleri, 2004. A new two-dimensional shallow water model including pressure effects and slow varying bottom topography. *Esaim-Mathematical Modelling and Numerical Analysis-Modelisation Mathematique Et Analyse Numerique*, 38(2) :211–234.

Gerbeau, J.-F. et B. Perthame, 2001. Derivation of viscous Saint-Venant system for laminar shallow water ; numerical validation. *Discrete and Continuous Dynamical Systems-Series B*, 1(1) :89–102.

Heniche, M., 1995. *Modélisation tridimensionnelle par éléments finis d'écoulements à surface libre.* Thèse de doctorat, Université de technologie de Compiègne.

Heniche, M., Y. Secretan, P. Boudreau, et M. Leclerc, 2000. A two-dimensional finite element drying-wetting shallow water model for rivers and estuaries. *Advances in Water Ressources*, 23 :359–372.

Heniche, M., Y. Secretan, P. Boudreau, et M. Leclerc, 2002. Dynamic tracking of flow boundaries in rivers with respect to discharge. *Journal of Hydraulic Research*, 40(5) :589–597.

Herouin, E., 1991. Modélisation des écoulements complexes à surface libre en milieu naturel (étude bibliographique). Rapport de dea de mécanique des fluides, Université Claude Bernard (Lyon1).

Koçyigit, M. B., R. A. Falconer, et B. Lin, 2002. Three-dimensional numerical modelling of free surface flows with non-hydrostatic pressure. *International Journal for Numerical Methods in Fluids*, 40 :1145–1162.

Lelerc, M., 1985. *Modélisation tridimentionnelle des écoulements à surface libre par éléments finis : application au lac St-Jean (Québec).* Thèse de doctorat, Université de technologie de compiègne.

Li, B. et C. A. Fleming, 2001. Three-dimensional model of navier-stokes equations for water waves. *Journal of Waterway Port Coastal and Ocean Engineering-Asce*, 127(1) :16–25.

Lin, P. Z. et C. W. Li, 2002. A sigma-coordinate three-dimensional numerical model for surface wave propagation. *International Journal for Numerical Methods in Fluids*, 38(11) :1045–1068.

Miglio, E., A. Quarteroni, et F. Saleri, 1999. Finite element approximation of quasi-3D shallow water equations. *Computer methods ins applied mechanics and engineering*, 174 :355–369.

Paquier, A., 1995. *Modélisation et simulation de la propagation de l'onde de rupture de barrage.* Thèse de doctorat, Université Jean Monnet.

Rodi, W., 1984. *Turbulence models and their application in hydraulics. A state of the art review.* Institute F.F.H.U.O Karlsruhe, Karlsruhe.

Saint-Venant, B. d., 1871. Théorie du mouvement non-permanent des eaux, avec application aux crues des rivières et à l'introduction des marées dans leur lit. *Compte rendu des scéances de l'Académie des Sciences*, 73 :147–157.

Secretan, Y. et V. Dubos, 2005. Modélisation de la variabilité verticale des vitesses dans un modèle hydrodynamique 2D horizontal. *Revue Européenne des Éléments Finis*, 14(1) :115–138.

Sinha, S. K., P. Sotiropoulos, et A. J. Odgaard, 1998. Three-Dimensional Numerical Model For Flow Through Natural Rivers. *Journal of Hydraulic Engineering*, 124(1) :13–24.

Steffler, P. et H. Ghamry, 2002. Two dimensional vertically averaged and moment equations for rapidly varied flows. *Journal of Hydraulic Research*, 40(5) :579–587.

Steffler, P. et H. Ghamry, 2005. Two-dimensional depth-averaged modeling of flow in curved open chanel. *Journal of Hydraulic Research*, 43(1) :44–55.

Steffler, P. M. et Y. C. Jin, 1993. Depth averaged and Moment Equations for Moderately Shallow Free Surface Flow. *Journal of Hydraulic Research, IAHR*, 31(1) :5–17.

Stelling, G. S. et M. M. Busnelli, 2001. Numerical simulation of the vertical structure of discontinuous flows. *International Journal for Numerical Methods in Fluids*, 37(1) :23–43.

Tome, M. F. et S. McKee, 1994. GENSMAC : A computational marker and cell Method for free surface flows in general domains. *Journal of Computational Physics*, 110(1) :171–186.

Ye, J. et J. A. McCorquodale, 1998. Simulation of curved open channel flows by 3D hydrodynamic model. *Journal of Hydraulic Engineering*, 124(7) :687–698.

Yuan, H. L. et H. W. Chin, 2004. A two-dimensional vertical non-hydrostatic sigma model with an implicit method for free-surface flows. *International Journal for Numerical Methods in Fluids*, 44 :811–835.

Yuan, H. L. et C. H. Wu, 2004. An implicit three-dimensional fully non-hydrostatic model for free-surface flows. *International Journal for Numerical Methods in Fluids*, 46 :709–733.

Zhang, B., 1992. *Modélisation d'écoulements à surface libre avec fronts mobiles par éléments finis*. Thèse de doctorat, Université de Compiègne.

RAPPORT # 4 : Banc de test

Rapport de développement logiciel

Banc de tests élaboré pour Modeleur 2

Présenté par:

Thierry Malo

13 janvier 2006

Versions du document

Date	Auteur	Commentaires	Version
7 octobre 2005	Sébastien Quessy	Ajout de précisions sur l'utilisation du banc de test.	2.0

Table des matières:

Introduction

Le projet de conception de Modeleur 2, logiciel de simulation et de modélisation en hydrodynamique, est en cours depuis maintenant trois ans et la fonctionnalité du logiciel est sans cesse croissante. Une grande variété d'événements sont maintenant disponibles mais il n'est malheureusement pas garanti que ceux-ci soient toujours opérationnels. Il va de soit que des erreurs de programmation sont inévitables lors de la conception d'un logiciel et il n'est évidemment pas toujours simple de les identifier.

C'est pourquoi il a été pensé utile par Yves Secrétan et Éric Larouche de concevoir un banc de tests qui aurait pour but de reproduire le plus de séries d'événements possibles. S'il est nécessaire de spécifier des séries d'événements, c'est qu'il faut généralement exécuter plusieurs opérations avant de pouvoir procéder à l'évaluation d'une fonctionnalité précise. Ainsi, grâce à des combinaisons d'événements programmées en Python, il est possible de contrôler la fonctionnalité de la majorité des opérations disponibles dans Modeleur 2, et ce à l'intérieur de divers contextes différents les uns des autres.

Le présent rapport aura donc pour objectif de présenter le banc de tests conçu par le stagiaire Thierry Malo au cours de l'été 2005. Il y sera élaboré le contenu et la structure du banc de tests ainsi son fonctionnement en tenant compte des différentes façons de l'utiliser. Il y sera aussi décrit la structure des tests et la méthode employée pour écrire les scripts *script_test.py*. Le rapport se terminera par la description de l'entretien requis par le banc de tests afin de le maintenir à jour par rapport aux différents changements qui surviennent régulièrement.

1 - En quoi consiste le banc de tests

Principalement, le banc de tests est un regroupement de plus de 150 tests ayant chacun pour objectif d'évaluer le bon fonctionnement d'un événement dans des circonstances bien précises. Le logiciel Modeleur 2 est constitué de nombreux modules afin de simplifier la présentation et l'utilisation de celui-ci. Pour cette raison, le banc de tests suit une division tout à fait similaire avec des noms de répertoires rappelant ceux employés dans la structure du code de Modeleur 2.

Modeleur 2 permet de manipuler toutes sortes d'entités pour lesquelles les événements de base ne varient pas nécessairement. Pour chaque entité sont écrits des tests sur ces événements de base. Voilà pourquoi plusieurs tests semblent se répéter et être impertinents mais, d'une certaine façon, ils ont tous leur propre utilité. Principalement, il faut comprendre que la plupart des événements spécifiques à certaines entités impliquent l'utilisation de nombreux événements de base. Puisque chaque test doit concentrer son intérêt sur un événement en particulier, ou même parfois sur un attribut précis d'un événement, il faut alors s'assurer à l'aide de tests plus simples que ces événements de base fonctionnent adéquatement. De cette façon, on est certain qu'ils ne sont pas la cause de l'échec des tests plus élaborés dont ils ne sont pas le sujet. L'autre aspect qui explique la présence de tous les tests de base pour chacune des entités est qu'il est intéressant, et parfois même nécessaire, de manipuler certaines entités de l'extérieur même de l'interface graphique de Modeleur. C'est actuellement le cas des champs éléments finis que l'on ne peut créer ni manipuler à partir de l'interface graphique.

Bien sûr, le roulement des scripts s'avère fort avantageux pour les programmeurs puisqu'il leur fournit un moyen de valider les modifications qu'ils apportent au code ainsi que le nouveau code qu'ils ajoutent. Le banc de tests a aussi l'avantage, par exemple, d'éviter de travailler avec les modules d'interface graphique et de visualisation, ce qui est enviable lorsque ceux-ci sont non-opérationnels (en cours de changement). Enfin, il serait intéressant d'ajouter que, grâce au contrôle que l'on peut exercer sur les modules que l'on choisit d'importer pour chacun des tests, il est possible de vérifier quelles sont les dépendances des événements envers les différents modules du logiciel. En effet, un test renvoie parfois un message d'erreur informant la nécessité d'importer une librairie particulière. Si ce n'est d'un oubli d'importation de la part de l'auteur du script, ce genre de message indique alors qu'une dépendance imprévue et indésirable existe dans le code, ce qui permet aux programmeurs de revoir leur code de manière à éliminer ces fausses dépendances.

2 - Aspect technique et structure d'un test

Techniquement parlant, les tests sont programmés en Python. Afin de simplifier la tâche du programmeur en lui évitant de reproduire le même code continuellement, quelques fichiers nommés *Module_Test_...* sont des regroupements de fonctions. Une fois définies dans ces fichiers, il s'agit alors de spécifier les valeurs qu'on désire assigner à chacun de leurs attributs pour ainsi les utiliser adéquatement selon le contexte choisi pour chaque test.

En terme de contenu, tous les *script_test.py* suivent un certain modèle qui se résume selon la liste de sections suivante :

i) Commentaires
ii) Importations
iii) Chargement de la bd
iv) Ouverture d'un projet
v) Corps d'un test : Série d'événements
vi) Localisation d'un test

Ainsi, pour construire un quelconque test en conformité avec la structure actuellement mise en place, il faut produire chacune de ces sections en respectant l'ordre présenté.

i) Commentaires

Tout les tests débutent par la présentation de leur objectif qui est souvent suivie d'une série de commentaires. Ceux-ci permettent de comprendre toute la logique derrière la façon dont le script est écrit. Il arrive parfois que l'ordre des événements, ou bien seulement la présence de l'un d'entre eux, ne semblent pas tout à fait logique. De plus, il arrive que des attributs soient inadéquatement assignés et ce de manière intentionnelle. Voilà donc pourquoi les commentaires sont essentiels à l'écriture de ces scripts.

ii) Importations

Il y a ensuite les importations. Elles sont essentielles à l'exécution de chaque test. Il est important de souligner qu'elles doivent inclure tous les modules contenant des fonctions requises à l'exécution de ces scripts ainsi que les modules auxquels on fait appel en utilisant les événements de Modeleur 2.

iii) Chargement de la bd

Les scripts en soi commencent toujours par le chargement de la bd. En effet, le *main* débute toujours par l'appel de la fonction *restore* qui fait elle même appel au fichier *restore_bd.bat*. Tel que l'indique son nom, elle permet de charger la bd requise pour chacun des tests. Actuellement, tous les tests utilisent la même bd. Ceci signifie que pour exécuter tout le banc de tests, il faut tout d'abord construire une banque de données contenant une cinquantaine d'entités ayant des noms bien précis. Annexé à ce rapport se trouve un tableau des entités constituant la bd recommandée afin d'exécuter le banc de

tests tel que programmé actuellement. Lorsque la construction de cette bd est complète, il s'agit de la sauvegarder à l'aide du fichier *sauve_bd.bat*. Pour le moment, la bd porte le nom de *bd_test* et elle est sauvegardée sous le même nom, soit *bd_test.txt*. Les noms d'entités peuvent évidemment être changés mais ceci implique que tous les scripts qui font référence à ces entités soient adaptés en conséquence. Voilà pourquoi il est recommandé d'utiliser une bd des plus semblables à celle offerte.

L'utilisation de ce principe de chargement de bd au début de chaque test permet de s'assurer que cette bd demeure intacte tout au long de l'exécution du banc de tests. On évite ainsi d'ajouter constamment de nouvelles entités dans la bd et, surtout, de devoir ensuite les supprimer une à une.

iv) Ouverture d'un projet

L'ouverture d'un projet constitue pour la majorité des tests la première étape suivant le chargement de la bd. Elle est importante puisque tous les événements à propos des entités vérifient qu'un projet soit bel et bien en mode actif.

v) Corps d'un test : Série d'événements

À la suite du chargement de la bd et, si nécessaire, de l'ouverture d'un projet, on doit considérer tout le corps du script. Celui-ci consiste en une série d'étapes menant à l'évaluation de l'objectif principal du test, soit généralement le fonctionnement d'un événement selon un certain contexte. De manière générale, les séries d'événements produites dans les tests correspondent à celles que l'on produit directement dans l'interface utilisateur de Modeleur. Malgré tout, certains tests ont pour objectif de découvrir et de contrôler les limites de certaines opérations de telle sorte que l'on ne peut pas nécessairement reproduire une même série d'opérations dans l'interface utilisateur.

Chaque étape d'un script est précédée d'un commentaire identifiant ce qu'elle effectue et à quelle fonction elle fait appel. Il y a parfois une mise en contexte de l'appel de la fonction ainsi que l'ajout de remarques afin de clarifier certains éléments moins évidents à comprendre. De plus, pour chaque appel de fonction se trouve un message de test qui permet à l'utilisateur du banc de tests d'interpréter le résultat du test si cette fonction échoue. À la suite de l'exécution d'un test, qu'importe que le test soit un succès ou non, un fichier de résultat doit théoriquement être produit dans le même répertoire que celui qui contient le test, nommé *script_test.py*. C'est ce fichier de résultat de test qui permet de conclure si le test est un succès ou pas. En cas d'échec, une opinion de l'auteur du script sur ce qu'il croit en être la cause est présentée. Le message d'erreur ainsi que la consultation des résultats du banc de tests seront abordés de manière plus détaillée ultérieurement dans le rapport.

Au cours du dernier paragraphe, on spécifie qu'un fichier de message d'erreur doit «théoriquement» être produit puisqu'il y a toujours la possibilité de rencontrer des erreurs de précondition, de postcondition ou alors d'invariant qui font en sorte qu'aucun *fic_log.log* ne peut être produit. En effet, la lecture du code de Modeleur 2 est alors

immédiatement interrompue et il faut comprendre que ce type de situation n'est pas toujours indésirable. Par contre, l'auteur des *script_test.py* doit toujours prévoir ces erreurs afin qu'elle n'empêchent pas de récupérer le message d'erreur que chaque test doit renvoyer.

vi) Localisation d'un test

Lorsque le *main* d'un test est terminé, une ligne en commentaire est alors inscrite pour identifier l'endroit où se trouve le script dans le banc de tests. Soulignons que le nom des répertoires n'y est pas précédé du numéro de répertoire auquel il est rattaché. Certaines modifications sont parfois apportées à la numérotation du banc de tests et il serait alors beaucoup trop compliqué de devoir adapter tous ces commentaires de fin de script.

Voilà donc ce qui résume l'essentiel du contenu d'un test et la façon dont il est structuré. Ce modèle structural de test devra idéalement être respecté afin de conserver un maximum de cohérence à travers tout le banc de tests. L'utilisation des tests, que ce soit individuellement ou en groupe, ainsi que les différentes possibilités de consultation des résultats qui sont produits seront développées dans quelques unes des prochaines sections de ce rapport.

3 - Structure du banc de tests

3.1- Arborescence

La structure du banc de tests se veut simplement une arborescence de répertoires contenant entre autres les tests. Le banc de tests est un répertoire en soit dans lequel se trouvent différents modules dont les noms correspondent généralement à ceux des répertoires de code du logiciel. Essentiellement, chaque module regroupe donc les tests qui concernent une même catégorie d'entité. L'objectif de cette division du banc de tests en fonction des types d'entité est ainsi de faciliter l'accès aux nombreux tests.

Exception faite des répertoires *bin* et *bd*, chaque nom de répertoire est précédé par un numéro qui permet de disposer les modules dans un ordre précis et logique. Principalement, cet ordre s'avère utile lorsqu'on doit consulter les résultats. Le succès de plusieurs tests est pré-requis à la réussite de beaucoup d'autres tests. Si les premiers tests de chaque répertoire retournent des messages d'erreur, il devient illogique d'évaluer les résultats des tests qui les suivent puisqu'ils devraient eux aussi être des insuccès. La priorité est donc de corriger les événements de base puisqu'on élimine ainsi les risques d'échec attribuables à ceux-ci lors de tests plus élaborés. C'est seulement lorsque cette étape est réglée qu'il est recommandé de s'attarder aux autres tests dont la numérotation est plus élevée et qui concernent des événements plus pointus puisqu'on évalue alors les enjeux réels de chaque test.

Les modules contiennent évidemment plusieurs répertoires de tests, numérotés eux aussi selon la série auquelle ils appartiennent (ex : *6000_Module_MNT* qui contient plusieurs répertoires dont les numéros varient de 6001 à 6014). Fréquemment, ces répertoires contiennent d'autres répertoires plus spécifiques afin de catégoriser les sujets de test. Soulignons que ces sous-répertoires ne sont cependant plus numérotés mais ils sont tout de même identifiés par des noms auto-descriptifs. Globalement, cette façon de faire permet de clarifier la structure de l'arborescence.

Lorsqu'on atteint le dernier échelon de l'arborescence, le répertoire dans lequel on aboutit contient un groupe de fichiers bien précis et commun à chacun de ces répertoires. Tout d'abord, il contient le fichier *script_test.py* qui est le script écrit en Python du test en soi. Dans le même répertoire, on retrouve aussi un fichier *Modeleur_config.py,* adapté spécifiquement pour l'exécution du test contenu dans le même répertoire, ainsi qu'un fichier dénommé *makefile.mak* qui ordonne seulement la lecture du fichier *makefile.inc* ainsi que la localisation de ce fichier. Enfin, lorsque le *script_test.py* est exécuté, un fichier de message d'erreur nommé *fic_log.log* doit théoriquement être créé et placé dans le même répertoire que celui qui contient le test exécuté.

Soulignons l'existence d'un module bien à part, soit le *bin*, dans lequel on retrouve d'importants fichiers, leur importance provenant du fait qu'ils contiennent l'ensemble des fonctions utilisées pour programmer les *script_test.py*. Il existe aussi un module nommé *bd* qui contient quelques fichiers jouant un rôle capital quant à la gestion de la bd du banc de tests.

Afin que tous les fichiers requis pour l'utilisation du banc de tests soient aisément identifiables et que leur existence soit justifiée, la prochaine section décrira le rôle de chacun des différents types de fichier ainsi que leur localisation dans le banc de tests.

3.2 - Types de fichier

script_test.py :

Ce type de fichier est le plus important de tous puisqu'il contient tout le code propre à chacun des tests que l'on désire produire. Tous les tests possèdent le même nom de fichier mais il est tout de même simple de les différencier grâce à leur répertoire d'appartenance, lequel diffère évidemment pour chacun. Afin d'obtenir plus de détails sur le contenu d'un test, veuillez référer à la description apportée sur ce type de fichier dans la section « Aspect technique et structure d'un test ».

sauve_bd.bat :

Ce fichier sert simplement à sauvegarder la bd que l'on désire utilisée pour exécuter un ou plusieurs tests. Puisqu'il fait actuellement référence au serveur CABANO par l'intermédiaire de l'adresse ip, il sera donc nécessaire de changer cette adresse ip à partir du moment où un utilisateur du banc de tests devra utiliser un autre serveur. Le fichier doit être exécuté d'une fenêtre de commande et il prend en paramètres le nom de la bd ainsi que le nom du fichier qui contiendra l'information sur la bd. Noter que le programme *pg_dump.exe* doit se retrouver dans le répertoire *C:\Program Files\PostgreSQL\8.0\bin* pour être en mesure d'exécuter ce fichier.

Ex : *>> d:/dev/banc_test/bd/sauve_bd bd_test bd_test.txt*

Ce fichier est disponible dans le répertoire *bd* du banc de tests et il faut évidemment spécifier le chemin d'accès au moment d'écrire la commande d'exécution.

restore_bd.bat :

Pour sa part, ce fichier est le complément du précédent en ce sens qu'il sert à charger la bd ayant été sauvegardée par le *sauve_bd.bat*. Il fait lui aussi référence à l'adresse ip du serveur CABANO, ce qui pourrait possiblement nécessité du changement. Soulignons que le fichier *restore_bd.bat* prend les mêmes paramètres que le fichier précédemment décrit et qu'on le retrouve aussi dans le répertoire *bd* du banc de tests. Il est appelé au début de chaque test par la fonction *restore()* définie dans le *Module_Test_GestionnaireBD*, ce qui permet alors d'assurer l'obtention d'une bd stable et identique pour l'exécution de tous les tests. Noter que les programmes *dropbd.exe*,

createbd.exe et *psql.exe* doivent se retrouver dans le répertoire *C:\Program Files\PostgreSQL\8.0\bin* pour être en mesure d'exécuter ce fichier.

setup_test.bat :

Avant même de lancer l'exécution d'un test ou de tout le banc de tests, il est essentiel d'exécuter ce fichier *.bat* puisqu'il sert à indiquer le chemin d'accès vers le *main.exe* du logiciel Modeleur 2 ainsi que celui de quelques variables utilisées par le *makefile.inc*. Puisque le banc de tests est actuellement disposé dans le lecteur d:/ de l'ordinateur sur lequel il roule et que ce disque n'est pas toujours disponible sur tous les ordinateurs, il est évident que toutes ces variables qui servent de chemin d'accès devront être adaptées à une quelconque redirection du banc de tests vers un différent lecteur.

Puisque le fichier contient des variables servant de chemin d'accès pour plusieurs fichiers, il est évident que s'il n'est pas exécuté, il n'y a donc aucune chance qu'un quelconque test réussisse. On retrouve idéalement ce fichier dans le répertoire *bin* pour des raisons de logistique même si on peut théoriquement le disposer dans le répertoire de notre choix.

makefile.inc :

Ce fichier auquel on fait référence dans la description précédente contient tout le code qui permet d'aller lire tous les répertoires du banc de tests contenant un fichier *makefile.mak* à l'aide d'une seule commande. Ensuite, si un fichier *script_test.py* est reconnu dans ces répertoires, il est systématiquement exécuté, l'envoie de l'exécution provenant du *Modeleur_config.py*. Ce script est donc sans contredit un incontournable du banc de tests puisqu'il permet d'autoriser le roulement de tous les tests d'un seul coup. En effet, on peut ouvrir la console d'un répertoire général pour ainsi autoriser l'exécution de tous les tests qu'il contient à l'aide de la commande "*make –f makefile.mak all*" ce qui est fort utile pour l'exécution d'une longue série de tests.

makefile.mak :

Le *makefile.mak* est seulement une ligne de code qui permet au *makefile.inc* de reconnaître les différents répertoires et de les lire. Lorsqu'il n'est pas introduit dans un répertoire, l'exécution du *script_test.py* ne peut être effectuée et il n'y a alors aucun message de *fic_log.log* qui s'y écrit. Il doit donc être introduit dans tous les répertoires sans exception, soit autant ceux qui contiennent des tests que ceux qui contiennent d'autres répertoires.

bt_all.bat :

Ce fichier sert à compiler tous les résultats obtenus au cours d'un banc de tests sur une même page web. Le concept de la page web pour fin de consultation des résultats sera expliqué plus longuement dans une prochaine section de ce rapport. Pour l'instant, il s'agit seulement de comprendre que l'exécution de ce fichier *.bat* permet de lire, dans tous les sous répertoires du banc de tests contenant un test, les fichiers *fic_log.log* qui contiennent les messages d'erreur retournés lors de l'exécution des tests. Le fichier *bt_all.bat* est disponible dans le répertoire *bin*. Soulignons finalement qu'il fait appel au fichier *BT_info.py* que l'on retrouve dans le module *build* du logiciel et que ce même fichier fait appel à plusieurs autres fichiers contenus dans le *build* etqui servent à lire, compiler et afficher les résultats. Pour cette raison, il est donc essentiel de posséder ce module dans le même lecteur que celui qui contient le banc de tests.

Modeleur_config.py :

Pour sa part, ce fichier a pour fonction, entre autres, d'importer des modules ainsi qu'une série de *.dll*, celle-ci devant être personnalisée pour chacun des différents tests. Évidemment, on pourrait importer toutes les *.dll* d'un seul coup et avoir le même *Modeleur_config.py* pour tous les tests, ce qui pourrait sembler plus simple. Cependant, il arrive parfois qu'un événement tente d'obtenir des informations dans une *.dll* qui ne le concerne pas, ce qui correspond à une erreur de programmation introduisant des dépendances indésirées entre les certains modules. Disposer ainsi du strict minimum en terme de *.dll* permet alors d'identifier ces événements qui font défaut par leur surplus de dépendances. Par ailleurs, l'importation d'un nombre élevé de *.dll* ralenti le temps d'exécution total d'un test, ce qui doit être évité.

En plus d'assurer la configuration du bus et l'initialisation de l'envoi d'événements, ce même fichier sert aussi à exécuter le *script_test.py* contenu dans le même répertoire.

3.3 - Fonctionnement

Le fonctionnement du banc de tests mise sur la nécessité de disposer certains fichiers dans des répertoires précis. Lors de la description des différents types de fichiers du banc de tests, il était indiqué l'endroit où l'on retrouve les différents fichiers. Ces endroits doivent absolument être respectés puisque, autrement, le banc de tests risque fort de ne pas être opérationnel. Rappelons que les répertoires contenant les principaux fichiers sont le *bin* et le *bd*.

Insistons aussi sur l'importance d'identifier clairement les répertoires et de diviser adéquatement le banc de tests de telle sorte que la consultation d'éléments précis soit la plus simple possible. Cette remarque implique que les noms des répertoires doivent indiquer le type d'action que le test vérifie. Elle implique aussi qu'il est préférable de regrouper les tests qui concernent un même événement tout en conservant une division

entre les différents modules et les différentes entités. L'arborescence doit donc être développée avec soin et demeurée la plus conviviale possible.

4 - Utilisation du banc de tests

4.1 - Rouler le banc de tests

Le code est en constant changement, ce qui implique que des erreurs de programmation peuvent survenir chaque jour. L'objectif du banc de tests est donc d'identifier aussitôt que possible ces erreurs. Pour ce faire, il est recommandé de lancer l'exécution du banc de tests le plus souvent possible, idéalement tous les jours. On s'assure alors que les erreurs ne s'immiscent pas dans le code et n'engendrent pas d'autres erreurs.

Afin d'exécuter avec succès le banc de tests, la mise en application de certaines étapes doit absolument être respectée. Puisque toutes celles précédant l'exécution même du banc de tests ont été présentées plus tôt dans ce rapport, résumons simplement la liste de ces étapes dans l'ordre de leur exécution :

i) Construire une bd portant le nom «bd_test»
ii) Sauvegarder cette bd dans un fichier texte nommé *bd_test.txt* à l'aide du fichier *sauve_bd.bat*
iii) Initialiser les chemins d'accès requis pour plusieurs fichiers en exécutant le *setup_test.bat*
iv) Lancé l'exécution du banc de tests à l'aide de la commande ''*make –f makefile.mak all*''

Actuellement, tous les *script_test.py* sont programmés de telle sorte que le schéma que l'on utilise, portant le nom de « bd_test », soit détruit au tout début d'un test, soit lors de l'exécution du test en soi. L'information de la bd sauvegardée dans le fichier *bd_test.txt* est ensuite chargée pour reconstruire le schéma et, alors, on peut poursuivre l'exécution d'un test. Il est important de souligner que ces détails empêchent d'exécuter plus d'un test à la fois. Cette restriction est absente lorsqu'on n'a pas besoin de charger une bd. Il serait donc possible de retirer le segment de chargement de la bd des *script_test.py* afin de permettre l'exécution simultanée de plusieurs tests. Cette option permet de gagner du temps mais elle empêche de garder intacte la banque de données que l'on doit construire spécialement pour le banc de tests.

Plusieurs méthodes sont disponibles pour utiliser le banc de tests. Tout d'abord, il est possible de rouler un seul test à la fois. Il s'agit simplement d'ouvrir une console spécifique à un test. Son exécution peut être appelée de deux façons différentes. Afin de seulement exécuter le test, on peut utiliser la commande ''*make –f makefile.mak*''. Cette méthode ne renvoie cependant pas de description détaillée d'une erreur de précondition, de postcondition ou d'invariant dans la fenêtre de commande. Principalement, il est intéressant de consulter le message d'erreur qui survient pour ces types d'erreur puisque, si l'auteur des scripts ne les a pas prévu, il n'y a alors aucun fichier de message produit pour en consulter la source d'erreur. La seconde commande, soit ''*main –c Modeleur_config*'', permet pour sa part de renvoyer ces commentaires sur une éventuelle erreur de précondition ou autre, et sur l'endroit où elle se trouve. Ces informations sont

évidemment très appréciées puisqu'elles permettent d'identifier très rapidement le type d'erreur rencontré, ce qui s'ensuit généralement de corrections rapides.

Évidemment, une raison existe pour expliquer l'utilisation des deux méthodes d'exécution. Il s'avère que la seconde s'applique seulement pour l'exécution d'un test individuel et qu'aussitôt qu'on désire contrôler tout un répertoire de tests, il faut alors nécessairement utiliser la première méthode tout en ajoutant le mot « *all* » à la fin de la ligne de commande. Cet ajout permet de lancer tous les tests contenus dans le répertoire auquel fait référence la console, en suivant bien sûr l'ordre présenté par l'arborescence. Bien sûr, on doit alors utiliser davantage les fichiers *fic_log.log* dans lesquels s'écrivent tous les messages d'erreur reçus par le code du logiciel, en plus du message d'erreur que chaque *script_test.py* renvoie. Ces messages sont aussi complets que ceux obtenus dans une fenêtre de commande, et même parfois davantage. Seulement, la consultation de ces messages par l'ouverture des fichiers *fic_log.log* est un peu plus longue. Voilà qui nous amène donc à discuter de l'affichage des résultats.

4.2 - Catégories de résultats

Quatre catégories de résultats ont été conçues afin de regrouper les différents cas. Voici la liste de ces catégories :

- Succes
- Echec
- Etapes Prealables
- Configuration

Il faut comprendre que le succès d'un test peu correspondre autant à l'échec d'un événement qu'à sa réussite, selon ce qui est attendu et désiré par l'auteur du test. Un échec indique que l'objectif principal du test n'est pas atteint, tandis que la catégorie « Etapes Prealables » indique plutôt qu'un problème affecte un événement considéré secondaire par rapport à l'objectif réel du test. Ce problème fait la plupart du temps l'objet d'un test préalable selon l'arborescence proposé, d'où le nom de la catégorie, pour lequel on observe un échec. Voilà pourquoi il est recommandé de toujours analyser les messages d'erreur en respectant l'arborescence du banc de tests selon sa numérotation. Enfin, lorsqu'on rencontre la catégorie « Configuration », on fait alors face à une erreur sur l'assignation du nom d'une entité, ce nom étant incorrectement écrit ou n'identifiant pas l'entité dont on a besoin pour le test.

4.3 - Afficher et consulter les résultats

On peut consulter les résultats des tests un à un en ouvrant chacun des fichiers *fic_log.log* dans lesquels on retrouve le message d'erreur complet retourné par un test. Cependant, cette méthode s'avère surtout favorable lorsqu'on désire consulter un seul résultat. Lorsqu'on est plutôt intéressé par l'ensemble des résultats, il est alors recommandé d'exécuter le fichier *bt_all.bat* qui est chargé de lire tous les messages de

test et de les transcrire sur une page web. La consultation des résultats est ainsi plus rapide et accessible à tous.

L'affichage des résultats, disponible sur le site internet de l'équipe de recherche, soit http://www.intranet.gre-ehn.inrs-ete.uquebec.ca/, se fait dans la rubrique *Modeleur 2* de la section *Rapport de test*. La lecture des résultats sur le site s'oriente comme suit : pour chaque répertoire est affiché,

- dans la première colonne, le nombre de tests pour lesquels aucun fichier de résultat n'a été produit, situation généralement causée par une erreur de précondition, de postcondition ou d'invariant.
- dans la deuxième colonne, le nombre de tests pour lesquels le fichier de résultat indique soit un message de test de type erroné, que ce soit de type « configuration », de type « étapes préalables » ou bien de type « erreur ».
- dans la troisième colonne, le nombre total de sous-répertoires de tests que le répertoire en question contient

Les résultats, tout comme le nom des répertoires, sont affichés sous forme d'hyperliens pour permettre de consulter les résultats contenus dans chacun de ces répertoires. Lorsqu'un répertoire se présente comme le dernier échelon d'une chaîne d'entre eux menant vers un message, le nom du répertoire est alors inscrit en italique afin d'identifier clairement qu'il contient un message.

Enfin, un code de couleur très simple a été mis en place afin de faciliter la consultation des résultats du banc de tests. L'hyperlien vers un ou plusieurs résultats est vert seulement à partir du moment où tous les résultats qu'il contient sont des succès. Autrement, aussitôt qu'un fichier de message indique une erreur, l'hyperlien devient rouge.

4.4 - Structure d'un fichier de message de test

Tout d'abord, on identifie deux segments de message à l'intérieur d'un fichier *fic_log.log*, soit le fichier de message de test. Le premier segment est un message réalisé par l'auteur du script. Il précise d'une part la catégorie du message parmi les choix présentés dans la section 4.2. Ce message contient aussi une petite description de ce que l'auteur du script croit être en cause s'il y a un quelconque type d'insuccès et, en cas de succès, il rappelle l'objectif du test en expliquant quel est le succès enregistré.

Le second segment est celui du code C++ de Modeleur 2. Il est lui aussi catégorisé et il est généralement accompagné d'une brève explication. Il est important de souligner qu'il arrive parfois que le message de l'auteur soit un succès et que le message de Modeleur 2 soit un échec. Ce genre de situation est observée lorsqu'on évalue les limites des événements que l'on vérifie et leur réaction selon divers contextes possibles pour un même événement. Cet aspect qu'est le contexte est déterminant puisque différentes combinaisons de paramètres mènent souvent dans des blocs de codes

différents. La production de groupes de scripts évaluant ces différentes combinaisons est donc la seule façon de s'assurer qu'un événement est véritablement fonctionnel. Ainsi, lorsque l'auteur vérifie l'échec d'un événement grâce à certains paramètres, cet échec correspond alors à un succès dans le message de l'auteur.

En somme, le message fourni par l'auteur du script donne généralement un bon aperçu du type de résultat obtenu et de ce qui doit être corrigé si le besoin y est. Cependant, le message C++ ajoute parfois plus de détails sur la raison d'un échec. Ces deux messages sont donc complémentaires et une consultation de chacun est toujours recommandée, surtout lorsque l'un affiche une erreur tandis que l'autre affiche un succès. En effet, il arrive que les bons résultats ne soient pas obtenus pour les bonnes raisons. Il faut alors simplement s'assurer que le message d'échec de Modeleur est accompagné par la bonne raison et qu'il est cohérent avec le type d'erreur attendu.

5 - Entretien du banc de tests

5.1 - Général

Cette section vise à dresser une liste des opérations régulières qui devront être effectuées afin de maintenir à jour le banc de tests, soit le code et la bd. La plus évidente des opérations est bien sûr celle de la mise à jour des *script_test.py*. On doit comprendre que de nombreuses modifications dans le code C++ de Modeleur 2 impliquent la correction de certains scripts et parfois même de certaines fonctions. Il faut donc prendre soin de corriger tous les scripts concernés par les corrections, mais aussi voir à ce que les commentaires qui accompagnent chaque test demeurent valides. Si les corrections méritent d'être commentées, il serait apprécié qu'elles le soient de telle sorte que la consultation ultérieure d'un test soit faciliter.

Par ailleurs, ces mêmes modifications dans le code C++ peuvent occasionnellement provoquer la reconstruction complète ou partielle de la bd, particulièrement lorsque de nouvelles colonnes sont ajoutées à certaines tables, ou alors que de nouvelles tables doivent être produites lors de la création d'un schéma. Le seul cas pour lequel on peut considérer la possibilité d'insérer manuellement une information à l'aide d'une requête SQL est lorsque la table qui contient l'information est seule et indépendante des autres tables. Pour tous les autres cas, il est recommandé de reconstruire entièrement la bd. Par exemple, l'ajout d'une nouvelle table implique très souvent des liaisons avec plusieurs autres tables et il devient aventureux de tenter de reconstruire le schéma de manière à respecter le nouveau code. Il ne faut jamais oublier que le principal objectif du banc de tests est d'identifier les erreurs de programmation commises dans le code C++. Selon cette perspective, on ne peut se permettre de commettre une erreur en adaptant un bd existante en fonction de changements majeurs ayant lieu dans le code.

Évidemment, l'opération de reconstruction de la bd s'avère longue mais on s'assure ainsi que le banc de tests fournira des résultats correspondant au code mis en place et que les résultats ne seront pas faussés.

5.2 - Reconstruction de la bd

L'opération de reconstruction de la bd du banc de tests mérite certainement de s'y attarder puisqu'elle s'avère actuellement méthodique et laisse peu de marge de manoeuvre à son constructeur. En effet, un ordre précis dans la création des diverses entités qui constituent la *bd_test* doit être respecté.

La mise en place de cette bd débute, une fois le projet créé, par la création de tous les champs analytiques requis et se poursuit par la création des partitions de maillages, la création des couches qu'elles contiennent et l'opération de maillage sur certaines de ces couches. Cet ordre est établi en raison de la nécessité de disposer de champs analytiques

afin de sauvegarder les nouvelles couches de maillage que l'on ajoute. De plus, les maillages sont requis afin de construire les différents champs ef de la bd.

Les champs ef doivent actuellement être créés avec les tests de sauvegarde sous disponibles dans le répertoire 3000_GestionChampEF. Il faut évidemment modifier les noms d'entité assignés dans chacun des scripts. Si l'étape initiale du *restore* est conservée, il faut alors s'assurer qu'à la suite de l'exécution de chacun des tests de sauvegarde sous, la bd soit sauvegardée à l'aide du fichier *sauve_bd.bat*. Si on préfère éviter cette sauvegarde successive de la bd, il est aussi possible de placer en commentaire de manière temporaire l'étape du *restore,* évitant ainsi que le schéma de la bd soit détruit au début de chaque test.

Lorsque tous les champs ef sont créés, on peut alors produire les modèles numériques de terrain (mnt) ainsi que toutes les partitions et les couches de topographies qu'ils contiennent puisque ces couches requièrent l'assignation de champs ef. Enfin, avant de sauvegarder ces couches, il est recommandé de les éditer de telle sorte qu'elles ne soient pas superposées comme elles le sont au moment de leur création.

On résume donc les nombreuses étapes de reconstruction dans l'ordre suivant :

- Création et sauvegarde des champs analytiques
- Création des partitions de maillages
- Ajout de couches de maillage sur ces partitions
- Édition de ces couches
- Sauvegarde des partitions de maillage
- Opération de maillage sur certaines couches de maillage
- Création et sauvegarde de champs ef grâce aux scripts de sauvegarde sous du module 3000
- Création de mnt
- Ajout de partitions de topographie sur ces mnt
- Ajout de couches de topographie sur ces partitions de topographie
- Édition des couches de topographie
- Sauvegarde des mnt
- Création des semis de points topographie

Globalement, il est clair que cette procédure est complexe et rigide même si elle fonctionne. Voilà pourquoi il serait approprié de considérer la possibilité de construire une série de scripts SQL, ou alors de fichiers Python similaires à ceux produits dans le banc de tests, qui auraient pour seul objectif de reconstruire systématiquement toute la bd grâce à la simple exécution de ce groupe de scripts. L'opération de reconstruction de la bd deviendrait plus rapide et elle serait simplifier.

5.3 - Bd construite dans la vie courante

Il est important de mentionner que l'ordre de construction de la bd requise pour rouler le banc de tests ne correspond pas à l'ordre des étapes de traitement de données

topographiques. En effet, l'utilisateur de Modeleur 2 disposera théoriquement de données de topographie qu'il devra importer. Il pourra alors les modifier, ces modifications consistant principalement à sectionner son regroupement de données topographiques et à éliminer les données aberrantes. Il pourra ensuite publier les jeux de données obtenus. Cette publication génère automatiquement des champs ef scalaires pour chacun des jeux de données ainsi que des maillages ef qui leur sont associés. Soulignons que l'un des deux jeux de données sera considéré prioritaire sur l'autre (au moment de l'assemblage). Lorsque cette étape est terminée, il s'agit alors de créer des couches de topographie correspondant aux différents champs ef de topographie générés et de construire un maillage commun à tous ces champs que l'on désire traiter à l'aide de couches de maillage. Grâce à la fonction d'assemblage d'un mnt sur un maillage, il est alors possible de générer un champ ef afin de regrouper en quelque sorte les différentes données de topographie sur un même champ.

Puisque cette logistique de traitement de données correspond davantage à ce qui devra être respecté dans la réalité, une courte série d'entités a été produite dans la bd du banc de tests. Elle se situe à la toute fin du tableau d'entités du banc de tests et elle permet d'effectuer un test intégrateur qui vérifie que le champ produit par l'événement d'assemblage d'un mnt sur un maillage tient adéquatement compte des priorités de couches. Essentiellement, le test consiste à effectuer une requête de valeur sur un noeud du champ ef produit et commun aux deux jeux de données initiaux afin de vérifier si la valeur qu'il retourne correspond à peu près à celle du jeu de données prioritaire.

Conclusion

En somme, le banc de tests est un lourd répertoire de tests qui permet d'augmenter notre niveau de confiance envers les différentes fonctionnalités de Modeleur 2. L'ajout ou la modification de code peut du même coup être immédiatement validé, évitant de traîner des erreurs de programmation qui, en l'absence de cette structure, seraient possiblement rencontrées que bien plus tard dans les étapes de conception.

La structure du banc de tests n'est pas très complexe mais elle est certainement quelque peu rigide. En effet, le retrait de certains fichiers peut rendre le banc de tests inutilisable et c'est pourquoi il faut être prudent. De plus, la mise en place de quelques modules regroupant l'ensemble des fonctions utilisées par les tests devra autant que possible être respectée lors de l'élaboration de futurs tests et de futures fonctions afin de demeurer cohérent avec l'approche procédurale employée, soit une approche orientée sur l'utilisation de fonctions. Il s'agit simplement d'écrire les fonctions dans les modules de fonctions et d'importer les modules requis dans chacun des tests. Cette méthode allège le code et permet d'obtenir un format de script simple et facile à consulter.

Ce banc de tests sera possiblement poursuivi par un futur stagiaire afin d'en enrichir le contenu et de le maintenir à jour en fonction des changements qui seront apportés au code du logiciel. Il sera donc important que cette personne tente de respecter la structure mise en place ainsi que son fonctionnement au moment d'apporter quelques modifications afin que le tout demeure cohérent et fonctionnel. Par la suite, il sera convié à tenter de construire un groupe de scripts SQL ayant pour tâche de reconstruire la bd requise pour l'exécution du banc de tests. De cette façon, il pourra éliminer les complications qu'impliquent les changements apportés au schéma d'un projet.

Pour conclure, soulignons que Modeleur 2 met à disposition de l'utilisateur de nombreux événements afin de modéliser les cours d'eaux et d'effectuer des simulations. Pour cette raison, il faut comprendre que le banc de tests actuel regroupe seulement que l'essentiel des événements, soit les événements de base ainsi que certains événements plus particuliers qui concernent les entités de base. Afin de compléter le travail accompli jusqu'à maintenant, les ajouts des quelques modules manquants tels que celui sur les séries seront assurément les bienvenus.