Institut Supérieur Industriel de Bruxelles
Rue Royale 150 — 1000 Bruxelles
Rue des Goujons 28 — 1070 Bruxelles
www.isib.be

Enseignement Supérieur de Type Long et de Niveau Universitaire

Haute Ecole Paul-Henri Spaak
Catégorie Technique

Développement d'une interface OpenMI pour interconnecter Tornado avec d'autres simulateurs de systèmes d'eau

M. Daniel VANDERPYPEN

Travail de fin d'études

Effectué au sein de l'entreprise :
model*EAU* – Université Laval
(Avenue de la médecine 1065, G1V 0A6 Québec)

Présenté en vue de l'obtention du grade
de Master en Sciences de l'Ingénieur Industriel
en Informatique

———————

Année Académique 2008-2009

———————

Numéro : ISIB-ELIN-TFE-09/09
Classification : tout public

———————

"Un standard est quelque chose que l'on pourrait améliorer et qu'on n'améliore pas ; c'est parce qu'on ne l'améliore pas que c'est un standard"

Rodnay Zaks

Résumé

La gestion environnementale au niveau de la qualité actuelle a recours à outrance à la modélisation de ses systèmes. Elle est en effet un outil puissant pour la modélisation de processus de systèmes d'eau. Comme tous les outils, ceux-ci doivent évoluer, offrir de nouvelles fonctionnalités. Il en est de même pour la plate-forme générique de simulation Tornado.

En conséquence de l'application de la directive-cadre sur l'eau lancée par l'union européenne, Tornado devait se voir implémenter une interface OpenMI. Le présent travail en relatera l'implémentation.

OpenMI, provenant de Open Modelling Interface, est donc un standard libre visant à la réalisation de modélisation intégrée. Il a pour cible l'ensemble des logiciels de modélisation dépendant du temps. Ainsi en procédant aux simulations des modèles pas à pas, le standard leur permet d'échanger leurs données calculées. Un logiciel de simulation implémentant OpenMI pourra être qualifié de composant en comportant le qualificatif d'*OpenMI-Compliant*.

Tornado étant une plate-forme générique de simulation, son interface OpenMI a donc dû être implémentée en tenant compte de cette généricité. En effet, lors de l'initialisation d'un composant, il faudra traiter l'ensemble des entrées et des sorties proposées par le modèle utilisé par le composant. Ce traitement visera à proposer les données sous forme standardisée par OpenMI en vue d'interconnecter ce composant avec d'autres.

En réalité, trois composants ont été mis au point pour Tornado. Le premier, nommé Reader, permettra l'utilisation de fichiers d'entrées de Tornado en vue d'une utilisation avec des composants hors cadre de Tornado. Le second, nommé Writer, permettra, en opposition au Reader, d'inscrire les données reçues d'un composant hors cadre de Tornado dans un fichier de sortie. Le dernier et le plus important, nommé Exec, permettra l'exécution classique des simulations Tornado tout en permettant l'utilisation d'entrées venant d'autres composants ainsi que l'exportation des résultats vers d'autres composants.

Pour finir, deux tests ont été élaborés mettant en relation les divers composants développés dans le cadre du présent travail.

Avant toute chose, je tiens particulièrement à remercier Dr. Peter Vanrolleghem pour m'avoir accepté comme étudiant stagiaire de fin de maîtrise et de m'avoir accordé ainsi un chaleureux accueil au sein de son groupe de recherche modelEAU à l'université Laval à Québec.

Mes remerciements vont à Dr. Filip Claeys qui m'a offert l'opportunité de travailler sur Tornado et de m'avoir ainsi accueilli au sein de MOSTforWATER pour m'aider à m'y préparer. Je tiens également à le remercier pour les nombreux coups de main réalisés à distance grâce à Skype.

Je ne voudrais pas oublier non plus Dr. Ing. Dirk Muschalla de modelEAU pour sa patience et son aide multilingue à propos de mes nombreux questionnements relatifs à OpenMI.

Mais ce stage n'aurait pu avoir lieu sans l'aide de deux de mes professeurs, à savoir Ir. Rudi Giot et Dr. Ing. Jacques Tichon. Je tiens à les remercier pour leurs encouragements et pour m'avoir aidé à obtenir ce stage à l'université Laval à Québec.

Evidemment, ce stage n'a pu également bien se dérouler que grâce au bon accueil de l'ensemble des membres du groupe VLAP et plus particulièrement aux membres de modelEAU. De même, je tiens à les remercier sincèrement pour leur aide de tous les jours dans mon apprentissage de la modélisation de la qualité de l'eau.

Pour finir, je ne voudrais pas non plus oublier mes collègues, amis et professeurs, qui m'ont accompagné tout au long de mes cinq années d'étude à l'I.S.I.B. et qui auront ainsi contribué à l'aboutissement du présent travail.

Juin 2009, Daniel Vanderpypen.

Table des matières

1 Introduction **1**

2 Tornado **2**

2.1 Description générale 2

2.2 Fichier de configuration de Tornado 2

2.3 Description des modèles 2

2.4 Interface de programmation (API) 3

3 Open Modeling Interface **5**

3.1 Gestion intégrée par bassins versants 5

3.1.1 Application de la directive-cadre sur l'eau 6

3.1.2 Le projet HarmonIT 7

3.1.3 Fondation de l'association OpenMI 7

3.2 Principes de base 8

3.2.1 Buts et objectifs 8

3.2.2 Description de l'espace de nom OpenMI 8

3.2.3 Les composants OpenMI 10

3.2.4 Les données échangées 10

3.2.5 Les liens 13

3.2.6 Processus d'échange des données 14

3.2.7 Déploiement de composants OpenMI 17

3.2.8 En pratique 18

4 Développement de l'interface OpenMI sur Tornado **22**

4.1 Introduction 22

4.2 Composants Tornado OpenMI 24

4.2.1 Introduction 24

4.2.2 Reader 24

4.2.3 Writer 26

4.2.4 Exec 27

4.3 Journalisation 29

4.4 Implémentation de la Wrapper class 30

4.4.1 Gestion des ExchangeItem 30

4.4.2 Gestion du temps au sein des composants 31

4.4.2.1 Echelle de temps relative et absolue 31

4.4.2.2 Méthodes relatives au temps 32

4.4.2.3 Démarrage des modèles 32

4.4.3 Initialisation 33

4.4.3.1 Introduction . . . 33
4.4.3.2 Reader . . . 34
4.4.3.3 Writer . . . 35
4.4.3.4 Exec . . . 37
4.4.4 Processus de réponse aux requètes . . . 39
4.4.4.1 Processus standard . . . 39
4.4.4.2 Particularités du Reader . . . 42
4.4.4.3 Particularités du Writer . . . 43
4.4.4.4 Particularités du Exec . . . 44
4.4.5 Phase de clôture . . . 45
4.5 Mise en garde sur les conversions d'unité . . . 45

5 Phase de test **47**
5.1 Introduction . . . 47
5.2 PredatorPrey . . . 47
5.3 ASU . . . 47

6 Conclusion **50**

A Description XML de l'expérience Tornado PredatorPrey **52**

B Fichiers omi du test Predator-Prey **55**

C Fichiers omi du test ASU **56**

Table des figures

1 Environnement d'expérience Tornado [2] 4
2 Gestion intégrée par bassins versants 5
3 Architecture de l'espace de nom OpenMI [10] 9
4 Diagramme de classe - Interface ILinkableComponent [9] 10
5 Diagramme de classe - Interfaces IExchangeItem et autres associées [9] 11
6 Diagramme de classe - Interfaces IElementSet et autres associées [9] 12
7 Illustration de différents types d'ElementSet [9] 12
8 Diagramme de classe - Interfaces IQuantity et autres associées [9] . 13
9 Diagramme de classe - Interfaces IValueSet et autres associées [9] . 13
10 Diagramme de classe - Interface ILink [9] 14
11 Communication par requète/réponse [8] 15
12 Communication par requète/réponse bidirectionnelle [8] 15
13 Processus de récupération des valeurs échangées [8] 16
14 Phases de déploiement de composants OpenMI [8] 18
15 OpenMI Configuration Editor - Construction de modèles intégrés . 19
16 OpenMI Configuration Editor - Edition des liens 20
17 OpenMI Configuration Editor - lancement de la simulation 21
18 Diagramme de classe - Architecture du standard et du SDK [12] . . 23
19 Extrait du fichier d'entrée de la simulation ASU 25
20 Diagramme de classe - CReadComponent 25
21 Diagramme de classe - CReadComponentLinkableComponent 26
22 Diagramme de classe - CWriteComponentLinkableComponent . . . 26
23 Diagramme de classe - CWriteComponent 27
24 Diagramme de classe - CExecComponentLinkableComponent 28
25 Diagramme de classe - CExecComponent 29
26 Diagramme de classe - CExecConnector 30
27 Diagramme de séquence - Méthode GetValue [12] 40
28 OpenMI Configuration Editor - test du modèle PredatorPrey 47
29 Graphique des résultats de PredatorPrey 48
30 OpenMI Configuration Editor - test du modèle ASU 48
31 Graphique des résultats de ASU 49

1 Introduction

Le présent travail de fin d'étude vise en l'implémentation d'une interface OpenMI sur la plate-forme de simulation Tornado en vue de pouvoir l'interconnecter avec d'autres simulateurs de systèmes d'eau.

La gestion environnementale devient au fur et à mesure du temps de plus en plus en plus exigeante et donc complexe à gérer. Des outils d'aide à la prise de décision deviennent indispensables et se basent avant tout sur l'utilisation de modèles mathématiques simulant ainsi les comportements écologiques à gérer. La simulation d'un système de traitements d'eaux usées permet ainsi au gestionnaire d'en mieux comprendre les fonctionnements et ainsi de pouvoir prendre les bonnes décisions de gestion.

Nous verrons ensuite que Tornado est une plate-forme générique complète et performante de simulation visant ainsi à permettre l'analyse de comportements modélisés mathématiquement.

Devant notamment fournir une aide à la décision, nous verrons que ceux-ci doivent donc suivre les besoins des gestionnaires. Nous verrons donc que l'application de la directive-cadre sur l'eau, lancée par l'Union Européenne, a conduit à de nouveaux besoins. Le présent travail relate l'implémentation d'une interface OpenMI qui n'est autre qu'une des conséquences de l'application de la directive-cadre.

Pour en comprendre l'implémentation, nous procéderons à une description d'OpenMI afin d'en faire ressortir les fonctionnements et les besoins. Ensuite nous pourrons aborder l'implémentation de l'interface en elle-même sur Tornado. Le présent document reprendra ainsi les différentes phases rencontrées lors de l'implémentation et leurs problèmes associés pourront y être développés.

Suivra alors une brève phase de test permettant de mettre à l'épreuve le travail réalisé.

Nous pourrons ainsi clôturer le présent travail sur une brève conclusion sur l'ensemble du projet réalisé.

2 Tornado

2.1 Description générale

Le domaine traitant de la qualité de l'eau est typiquement divisé en différents secteurs, à savoir les systèmes de rivières, les systèmes d'égouts et les systèmes de traitements des eaux usées. Chacun de ces sous-domaines se voyait concevoir différents modèles mathématiques et différents logiciels permettaient leurs résolutions. Au cœur de ces logiciels réside une partie souvent qualifiée de noyau et cette partie semblait fort commune à chacun d'eux. Partant de ce constat, il semblait intéressant de proposer une base générique commune pour la résolution de tous ces modèles.

Le travail sur cet utilitaire pour le domaine de la qualité de l'eau a donc pris place au sein de l'université de Gand dans l'équipe BIOMATH et ce en collaboration avec MOSTforWATER NV. Le développement de cette plate-forme a ainsi fait l'objet de la thèse de Filip Claeys [2]. La fin de ce travail a donc abouti en la plate-forme générique Tornado offrant un service complet et performant de simulation et de résolution de modèles mathématiques. Bien que développé dans le domaine de la qualité de l'eau, Tornado, de part sa conception générique, se voit exploitable dans bien d'autres domaines.

2.2 Fichier de configuration de Tornado

Afin de pouvoir être initialisé, Tornado requiert un fichier de configuration au format XML. Il reprendra tout d'abord diverses propriétés concernant Tornado, tel qu'un numéro de version, un auteur, une date, une précision de calcul, un système d'unité ou encore la verbosité du noyau. Prend place ensuite la liste des plugins que devra lancer Tornado, il s'agit essentiellement des solveurs. Le fichier se terminera ensuite par la longue liste des unités que doit prendre en compte Tornado. Chacune des ces unités reprend diverses informations, comme un nom, une catégorie, des coefficients, pour une conversion vers le Système International ou encore une équivalence de noms vers le système métrique ou impérial.

2.3 Description des modèles

Tornado, étant une plate-forme générique, permet donc la simulation de modèles différents. Chacune de ces simulations prendra donc des caractéristiques différentes,

elles seront qualifiées d'expériences dans le cadre de Tornado.

Les expériences sont détaillées dans des fichiers XML qui indiqueront ainsi au noyau ce qu'il doit calculer. Ce fichier reprendra notamment diverses propriétés concernant l'expérience en elle-même, tel qu'un nom, une description, une date ou encore un système d'unité. Le fichier reprendra ensuite la description du modèle utilisé comprenant ainsi ses divers paramètres ainsi que la déclaration de l'ensemble des variables qu'il utilise. On définira ensuite les paramètres pour les entrées de la simulation. Un fichier d'entrées peut notamment y prendre place. On notera le fait qu'en utilisant OpenMI comme fournisseur d'entrées, il ne faudra pas le spécifier dans le XML. La simple déclaration des variables dans la description du modèle suffira. Vient ensuite la déclaration des sorties avec ensuite l'éventuelle déclaration d'un fichier de sorties avec la description des variables qui doivent y être présentes. Prend place alors la définition de la plage de temps de la simulation. Le fichier se termine alors par la spécification d'un solveur et de ses divers paramètres. Un exemple de ce fichier XML est disponible pour la simulation PreadatorPrey fournie avec Tornado dans l'annexe A.

La figure 1 représente la structure de Tornado. Le cœur de Tornado, représenté par le bloc Main, est donc joint au travers d'une interface par une application permettant ainsi l'interaction entre l'utilisateur et le noyau. On prendra comme exemple WEST [3] qui sera le premier logiciel à utiliser ce noyau. Tornado va ensuite fonctionner sur deux bases. D'un côté les expériences et de l'autre les solveurs. Tandis que les solveurs vont contenir les méthodes de résolution des modèles, les expériences vont, elles, décrire ce qui est attendu par l'utilisateur. Elles se composent de variables d'entrées et de variables de sorties, étant éventuellement liées à des fichiers d'entrées ou bien de sorties. Chacune de ces expériences reprendra alors des modèles mathématiques faisant ainsi le lien entre les entrées et les sorties avec le temps comme référence. Enfin, chacune de ces expériences sera liée à un solveur.

2.4 Interface de programmation (API)

Ayant un besoin important de rapidité de calcul, Tornado a été développé dans le langage qui offrait les meilleures performances à savoir, le C++. Toutefois, nous venons de le voir, Tornado est accessible au travers d'interfaces et celles-ci sont présentes en nombre. C'est ainsi que sont présentes sur Tornado, entre autres, des interfaces C, C++, Java, Matlab et .NET.

Pour la suite du travail, c'est avant tout l'interface .NET qui nous intéressera.

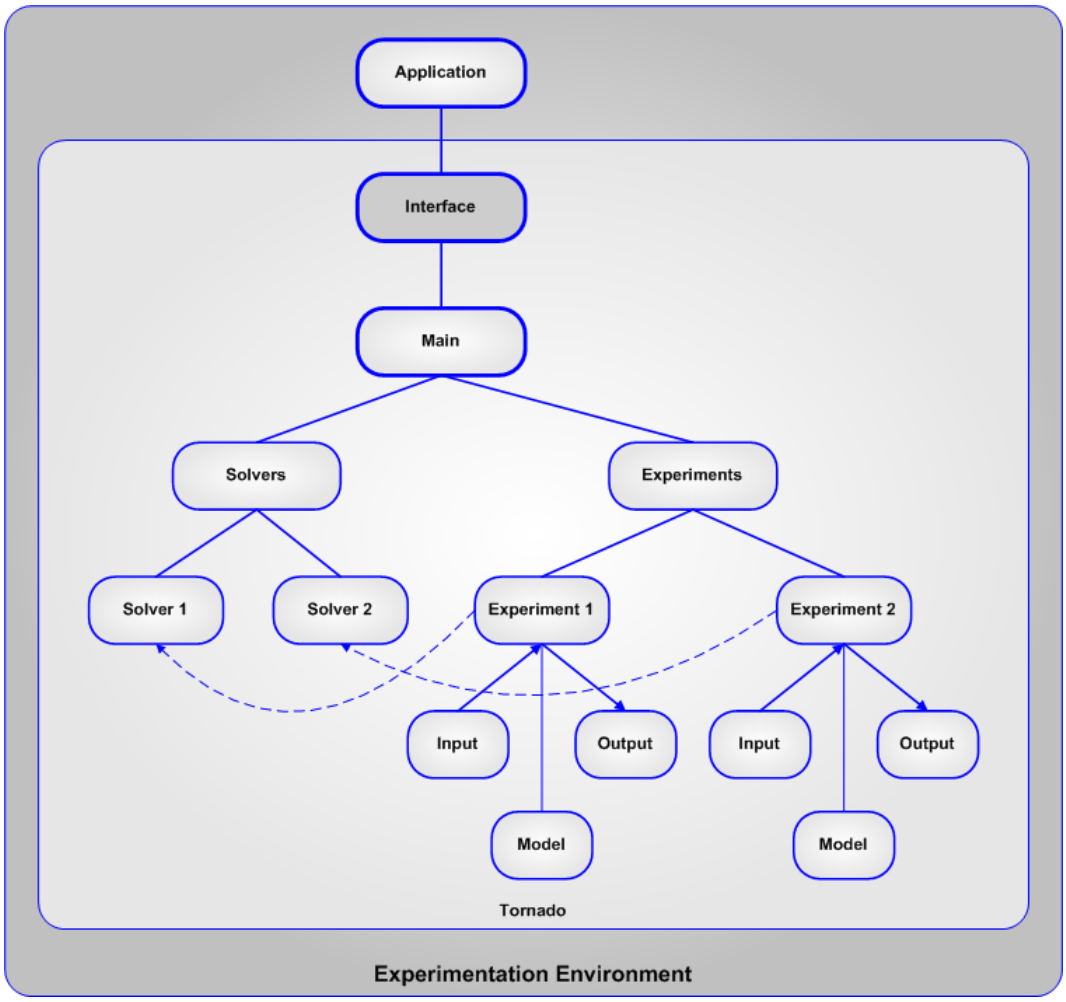

FIG. 1 – Environnement d'expérience Tornado [2]

C'est en effet au travers d'elle et à l'aide du langage C# que nous pourrons implémenter OpenMI sur Tornado.

L'utilisation de Tornado au travers de l'interface .NET/C# est relativement aisée. L'instanciation de la classe TornadoNET.CTornado nous offre un accès direct à Tornado. A partir de cet objet, l'ensemble des interactions possibles avec le noyau sont établies au travers de méthodes implémentées dans l'espace de nom TornadoNET.

3 Open Modeling Interface

3.1 Gestion intégrée par bassins versants

Un bassin versant est une région qui possède un exutoire commun pour ses écoulements de surface. C'est l'équivalent d'un réservoir délimité de telle façon que toutes les précipitations qu'il reçoit, contribuent au débit de cet exutoire [1]. Leur gestion écologique s'avère complexe et peut s'observer sous différents angles.

Il n'y a pas si longtemps, on regardait les problèmes de pollution avant tout en termes d'émission. Nous avions connaissance des sources de rejet de polluants. Ces sources étaient quantifiables et des contraintes étaient fixées pour limiter la quantité de rejet de chacune de ces sources. On regardait ainsi en termes d'émission. Cependant, ces émissions se répartissant à l'échelle d'une ville entière avaient des répercutions différentes sur leur milieu, à savoir un bassin versant et un cours d'eau comme peut l'illustrer la figure 2.

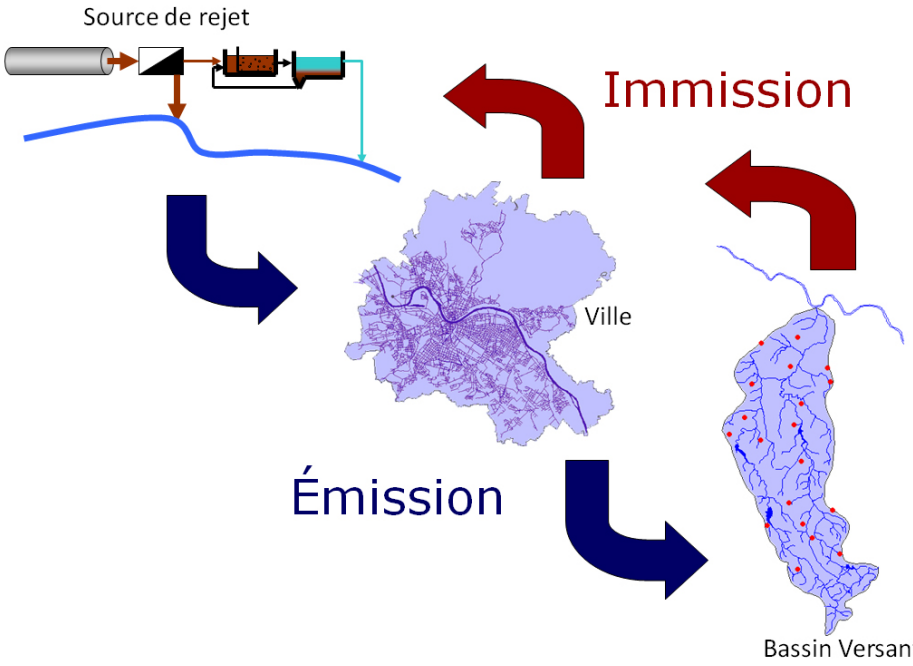

FIG. 2 – Gestion intégrée par bassins versants

Cette conception de la gestion n'était assurément pas la meilleure. On regardait en termes d'émission chacune des sources, alors qu'au final, ce qui importait était le niveau d'immissions du milieu perçu par les émissions ! Une même usine n'aura pas le même impact sur le Saint-Laurent que sur la rivière Montmorency, la capacité du milieu n'étant pas la même.

Depuis quelques années, nous adoptons donc cette nouvelle façon de gérer la qualité de l'eau en pensant en termes d'immission. Avec ce nouveau regard, la gestion de la qualité de l'eau se voit complètement bouleversée. Dorénavant, les émissions de chacune des sources seront évaluées en fonction de l'immission du bassin versant. Cette façon de gérer va donc ainsi nous apporter plus de liberté et de précision dans ce travail.

3.1.1 Application de la directive-cadre sur l'eau

"L'eau n'est pas un bien marchand comme les autres mais un patrimoine qu'il faut protéger, défendre et traiter comme tel." Tel est le premier Considérant de la directive-cadre sur l'eau [5] prise le 23 octobre 2000. Voici, qui met directement en relief l'importance de la préservation de la qualité de l'eau et du développement d'une politique intégrée concernant nos actions envers les fragiles écosystèmes aquatiques.

L'objectif de la directive-cadre sur l'eau (DCE) est de garantir un équilibre durable en établissant les fondements d'une politique de l'eau moderne, globale et ambitieuse pour l'Union Européenne. Ainsi, on allait devoir appliquer une gestion intégrée de nos bassins versants et établir notre gestion en termes d'immission telle que nous l'avons développée dans la section précédente.

La gestion environnementale se voit grandement aidée par la modélisation. La mise en équation des comportements qui nous entourent nous permet de mieux appréhender les conséquences des choix que nous faisons. Ceux-ci nous permettent en effet de prédire l'issue probable de nos décisions. Chacun des comportements est ainsi modélisable. Certains peuvent évaluer le fonctionnement des stations de traitements des eaux usées tandis que d'autres évaluent le ruissellement sur un bassin versant ou encore, certains évaluent le captage de polluants dans des bassins de rétention. En finalité, l'ensemble des comportements est modélisé à une échelle individuelle. Cependant, leurs interactions ont une importance capitale dans l'idée d'une gestion intégrée en bassin versant. Seulement pour faire face à cela, la réalisation d'un modèle unique serait tout simplement inconcevable. La seule solution se profilant était donc de relier chacun de ses modèles impliqués. Cette conception est qualifiée de modélisation intégrée [4].

3.1.2 Le projet HarmonIT

Une méthode était donc nécessaire à l'établissement des liens entres les modèles et les outils disponibles à la modélisation, afin de prendre en compte les interactions entre les processus. La commission européenne a alors lancé le projet HarmonIT pour réaliser ce travail.

Le projet cofinancé par l'Union Européenne a donc été lancé dans le courant 2001. Il s'est composé de quatorze partenaires représentant aussi bien des utilisateurs finaux que des centres de recherche ou des concepteurs de logiciels de modélisation. On notera d'ailleurs à ce propos que trois partenaires commerciaux (DHI - Water and Environment, WL - Delft Hydraulics and Wallingford Software) fournissant les systèmes de modélisation les plus utilisés, allaient devoir partager leur savoir pour la bonne élaboration du projet. C'est ainsi qu'ensemble ils ont produit OpenMI, provenant de Open Modelling Interface, à savoir une interface de modélisation libre.

Nous pouvons ainsi résumer la fonction d'OpenMI en une définition logicielle d'une interface destinée au calcul de modèle hydrologique et hydraulique. Les composants exécutant les modèles, en respectant le standard peuvent ainsi, sans programmation, être configurés pour s'échanger des données lors de leur propre exécution. Ceci signifiant donc la possibilité de conception simple de modèle intégré.

3.1.3 Fondation de l'association OpenMI

Suite à la fin de la conception du standard OpenMI, le projet HarmonIT a pris fin également, ses objectifs étant atteints.

Afin de garantir la pérennité du standard, l'association OpenMI a donc été mise sur pied en juin 2007. Ses objectifs étaient simples, elle devait réaliser la promotion du standard afin d'imposer OpenMI comme référence. L'association est également en charge de faire vivre le standard au travers de mises à jour. A l'heure actuelle, OpenMI est distribué sous sa version 1.4.

3.2 Principes de base

3.2.1 Buts et objectifs

Comme nous l'avons déjà vu, l'objectif de HarmonIT était de fournir un standard pour la gestion de la modélisation intégrée. La difficulté principale pour un tel projet, est de fournir un standard qui sera employé par la plus grande majorité. Pour ce faire, le standard devait concilier des modèles venant de différents domaines (l'hydraulique, l'hydrologie, l'écologie, la qualité de l'eau, l'économique, ...) mais également provenant d'environnements différents (atmosphérique, marin, eau douce, terrestre, urbain, rural, ...). De même, ces modèles devaient pouvoir aussi bien provenir de concept déterministe ou stochastique. Ils devaient pouvoir recourir à différentes dimensions (0, 1, 2 ou 3D) et à différentes échelles (p. ex. un modèle régional de climat à un modèle de bassin versant). De même que différentes résolutions temporelles devaient pouvoir être conciliées. Les modèles devaient également pouvoir utiliser des représentations spatiales différentes (p. ex. en grilles, réseaux ou polygones). Chaque modèle devait pouvoir également utiliser ses propres unités et enfin, chaque modèle devait pouvoir se lier à des bases de données, interfaces utilisateur ou encore à des instruments.

Nous pouvons ainsi résumer les contraintes rencontrées par HarmonIT sous six objectifs principaux. Tout d'abord, le standard devait être applicable à n'importe quelle simulation dépendant du temps. Ensuite, il devait être applicable aussi bien à l'ensemble des futurs modèles qu'aux modèles déjà existants et largement utilisés. Pour ce faire, l'implémentation du standard se devait de requérir le strict minimum de modification aux modèles déjà existants. L'objectif suivant, qui découle du précédant, est que le standard se devait d'imposer un minimum de restrictions face au modèle. Un des grands objectifs était également de fournir un standard facile d'utilisation pour l'utilisateur final. La modélisation est un milieu déjà relativement complexe. HarmonIT devait simplement fournir un outil et sans avoir la nécessité de reconsidérer l'ensemble du milieu. Pour finir, le dernier objectif principal était bien évidement de ne pas dégrader déraisonnablement les performances globales des modèles. Il est un fait que le processus d'interconnexion allait ralentir l'exécution des modèles, mais il fallait en minimiser l'impact.

3.2.2 Description de l'espace de nom OpenMI

Le standard OpenMI est défini comme une interface au sein de l'espace de nom org.OpenMI.Standard. Cette façon de procéder à la définition du standard

au travers d'interfaces va apporter ainsi une certaine flexibilité aux développeurs. L'implémentation du standard pourra se faire ainsi selon les besoins et envies des développeurs. Les composants logiciels implémentant et utilisant ces interfaces sont appelés *OpenMI-compliant*.

Afin de faciliter le travail nécessité pour rendre un simulateur fonctionnel *OpenMI-compliant*, une implémentation par défaut du standard est mise à disposition. Elle est développée au sein du framework .NET et est écrite en langage C#[1]. Cette implémentation est distribuée en tant que code source libre sous les conditions de licence GPL. Cette implémentation par défaut est appelée *OpenMI*

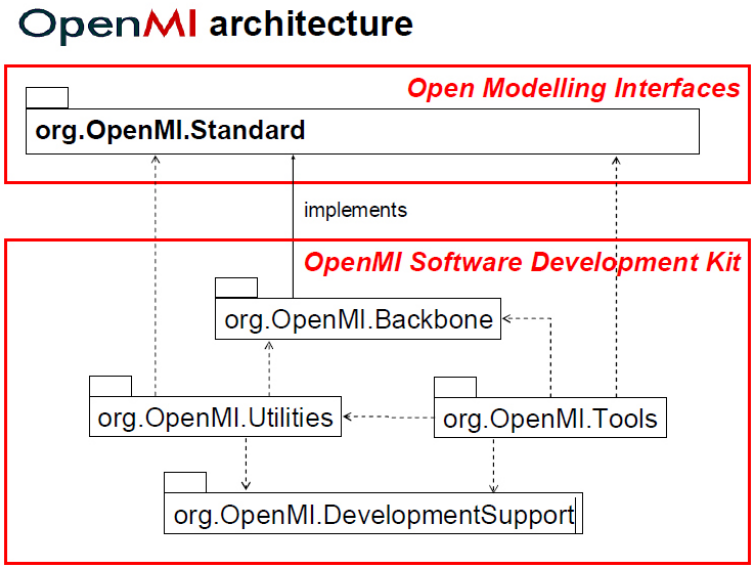

FIG. 3 – Architecture de l'espace de nom OpenMI [10]

Software Development Kit. Elle se compose de différents paquets tel que décrit par la figure 3.

- Le paquet org.OpenMI.Backbone fournit l'ensemble minimal de classe requis pour l'implémentation de l'interface du standard.
- L'espace de nom org.OpenMI.Utilities fournit des outils aidant au support de l'adaptation d'anciens codes, à manipuler les données ainsi qu'à configurer et déployer les composants.
- L'espace de nom org.OpenMI.DevelopmentSupport contient un logiciel générique permettant le parsing des fichiers XML de configuration.
- L'espace de nom org.OpenMI.Tools contient un groupe d'outils permettant des interactions avec le système.

Bien que le présent travail aura recours à ce Software Developement Kit, le standard OpenMI n'y contraint évidemment pas l'utilisation. Ce dernier est présent

[1]Une implémentation en Java devrait être distribuée prochainement

simplement pour aider le travail des développeurs et il serait dommage de réinventer la roue.

3.2.3 Les composants OpenMI

La principale nécessité d'un composant OpenMI est d'avoir une interface qui définit l'accès au composant à proprement parler. En d'autres termes, nous avons donc besoin d'accéder à l'implémentation de l'interface ILinkableComponent. Il s'agira là de notre point d'entrée OpenMI. L'implémentation de cette interface reprendra l'ensemble des fonctions de base requises tel que décrit par la figure 4.

FIG. 4 – Diagramme de classe - Interface ILinkableComponent [9]

L'ensemble des fonctionnalités, portant des noms assez significatifs, ne seront décrites que par la suite. On notera juste pour l'instant qu'elles reprennent la totalité des besoins pour une modélisation intégrée.

3.2.4 Les données échangées

Comme nous l'avons déjà évoqué, OpenMI a pour vocation d'être applicable à n'importe quel simulateur, pourvu qu'il soit temporel. Ainsi, les données qui seront échangées entre les simulateurs devront pouvoir être de tout type. Le standard prévoit donc une architecture complexe mais qui laissera ainsi une grande flexibilité aux programmeurs pour qui tous les types de données ont été prévus par le standard.

Les données échangées sont définies par l'interface IExchangeItem. Comme le montre la figure 5, cette interface hérite de deux autres interfaces qui feront la distinction entre les variables d'entrée et les variables de sortie. Les termes d'InputExchangeItem et d'OutputExchangeItem seront donc dorénavant utilisés pour définir respectivement les variables d'entrée et de sortie. Comme le décrit donc l'interface IExchangeItem, ceux-ci sont définis par une quantité et un elementSet.

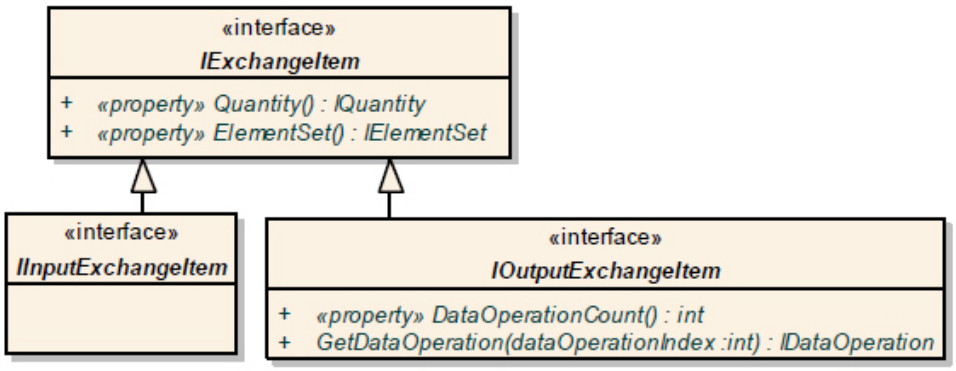

FIG. 5 – Diagramme de classe - Interfaces IExchangeItem et autres associées [9]

Dans l'optique d'une utilisation large d'OpenMI, les ExchangeItem sont caractérisés premièrement par un ElementSet, comme le décrit la figure 6. Cette caractéristique nous permettra de définir où les données s'appliquent. Dans le cadre de Tornado, cet aspect nous intéresse peu. Toutefois dans le cadre de modèles de bassins versants ou de rivières, nous pouvons avoir différents points d'entrée pour un débit par exemple. Il est bon de pouvoir les identifier ! Cette identification peut se faire suivant différents types (voir l'énumération ElementType de la Fig. 6). Vis-à-vis de Tornado, nous utiliserons uniquement le type "'IDBased"'. En comparaison, un modèle de bassin versant aura probablement plus recours au type "'XYZPolygon"' ou "'XYZPolyhedron"' et un modèle de rivière utilisera probablement simplement une "'XYLine"'. La figure 7 nous donne un aperçu de l'utilisation possible des différents types d'ElementSet. Evidemment leur utilisation dépendra des simulateurs rencontrés et des besoins de leurs modèles.

L'endroit où s'appliquent les données échangées étant établi, il faut ensuite définir à quoi correspondent ces données. L'interface IQuantity, décrite par la figure 8, se chargera ainsi de ce travail. Avant tout, OpenMI prévoit deux types de données distinctes à savoir, des scalaires ou des vecteurs. On notera, comme le décrit la figure 9, que le standard prévoit au sein de son interface IVector, des vecteurs à trois dimensions, ces dimensions faisant évidemment état d'une position spatiale à trois dimensions. Il n'est donc pas prévu de pouvoir utiliser notamment des vecteurs de concentrations tels que utilisés par différents simulateurs du traitement

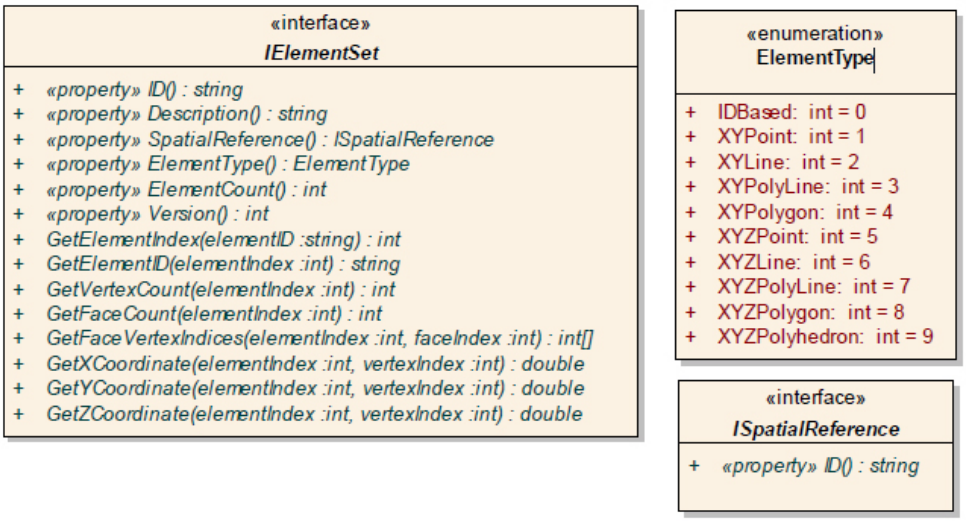

FIG. 6 – Diagramme de classe - Interfaces IElementSet et autres associées [9]

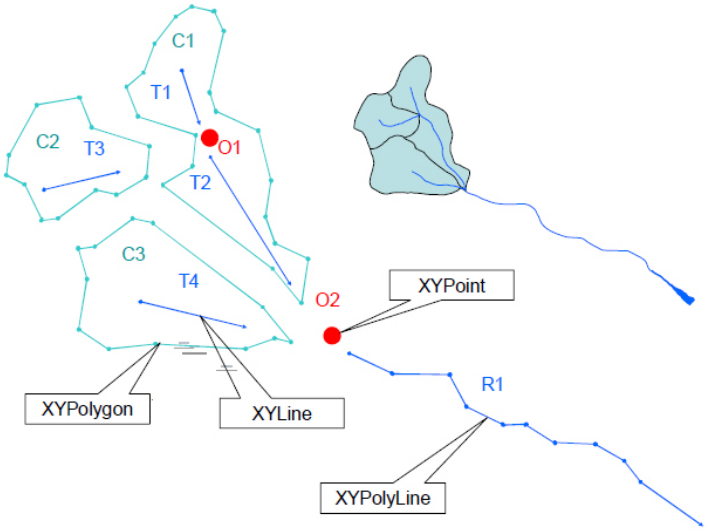

FIG. 7 – Illustration de différents types d'ElementSet [9]

des eaux. Outre cet aspect scalaire ou vectoriel, les données se voient dotées d'une dimension et d'une unité. La dimension correspond à l'unité de la donnée exprimée par rapport aux sept unités de base du système international en termes de puissance. Par exemple un volume sera donc exprimé en unité de longueur au cube, tandis qu'une vitesse sera exprimée en unités de temps exposant moins un et de longueur. Les unités quant à elles, reprennent en plus de leur description d'un facteur de conversion et d'un offset vers le système international. Ainsi, à l'aide de la dimension, OpenMI sera à même de contrôler la connexion uniquement entre données de même nature (p. ex. une vitesse avec une vitesse ou un débit avec un débit). Tandis que l'unité permettra à OpenMI de gérer la conversion entre des différentes unités de modèles différents et ce de manières transparentes pour l'utilisateur final.

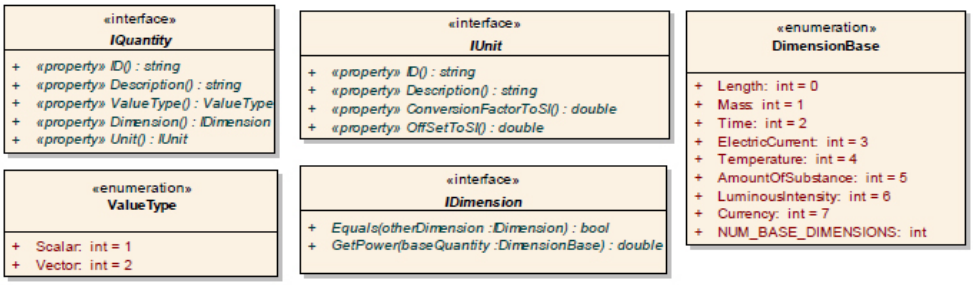

FIG. 8 – Diagramme de classe - Interfaces IQuantity et autres associées [9]

Nous avons décrit jusqu'ici des structures permettant de caractériser les données, mais les données en elles-mêmes doivent bien être gérées quelque part dans le standard également. C'est le rôle de l'interface IValueSet, décrite par la figure 9. Il s'agit simplement de la structure qui sera chargée de contenir les valeurs au travers des liens.

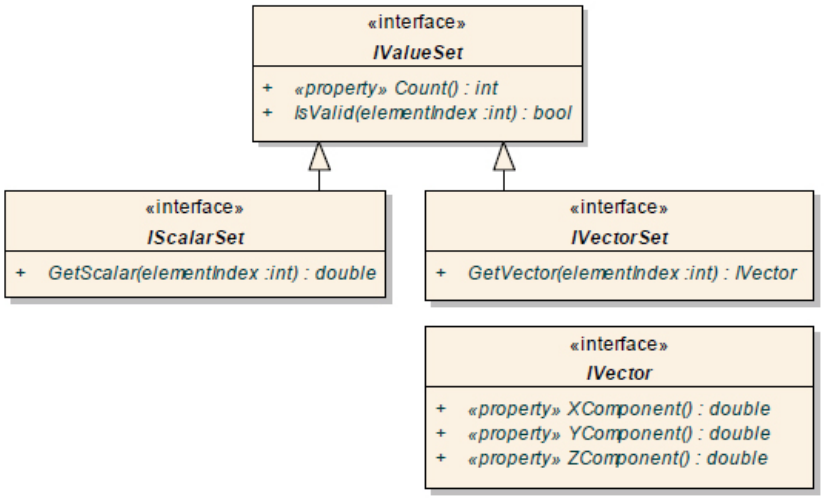

FIG. 9 – Diagramme de classe - Interfaces IValueSet et autres associées [9]

3.2.5 Les liens

Le but final du standard est de pouvoir raccorder des simulateurs entre eux. Ce raccord va se matérialiser au travers de liens comme le décrit l'interface ILink visible à la figure 10. On remarquera avant tout qu'un lien est unidirectionnel. Il se caractérise par un composant source et un composant de destination. De même, le lien comprend une Quantity et un ElementSet source ainsi qu'une Quantity et qu'un ElementSet de destination. En d'autres termes, nous pouvons dire qu'un lien se compose d'un simulateur fournissant un certain type de données provenant d'un certain endroit et d'un simulateur qui injectera cette donnée dans un de ses types de données et à un certain endroit.

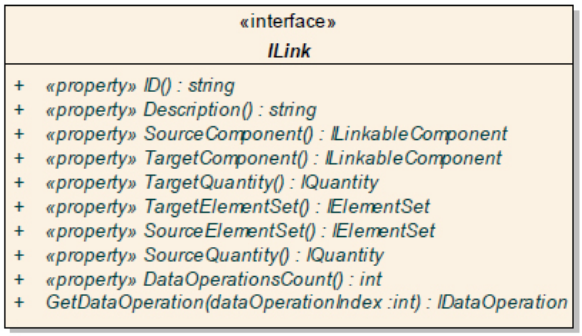

FIG. 10 – Diagramme de classe - Interface ILink [9]

3.2.6 Processus d'échange des données

Avant de détailler le processus d'échange des données, il est bon de rappeler qu'un composant OpenMI fonctionne pas à pas. Pour exécuter chacun de ses pas, le composant aura éventuellement besoin de variables d'entrées provenant d'autres modèles.

Le principe de requête/réponse a été repris par le standard OpenMI. La figure 11 illustre bien le principe. Nous partons avec quatre composants OpenMI. Des liens unidirectionnels ont été placés entre les composants. Les composants A, B et C ont besoin des résultats du composant les précédant pour calculer leurs propres valeurs de sorties. Des données seront échangées de D vers C puis vers B et enfin vers A. Le tout est géré comme le veut le standard par un trigger[2], c'est ce dernier qui va lancer le processus permettant à l'utilisateur d'arriver à ses fins.

Tout va donc démarrer par le trigger qui va donc envoyer une requête au composant A lui demandant de démarrer. Pour répondre au trigger, le composant A devra exécuter son premier pas. Seulement, pour ce faire, le composant A a besoin de résultats du composant B. Le composant A va donc lancer une requête au composant B pour obtenir les résultats dont il a besoin et va en attendre la réponse. Le même cas est rencontré par le composant B qui a besoin de résultats du composant C pour exécuter lui aussi son premier pas. Le composant B va donc à son tour envoyer une requête au composant C et attendre sa réponse. La même situation est ensuite rencontrée par le composant C et nous arrivons donc au composant D. Celui-ci est à même de calculer son premier pas. Il va donc ensuite répondre au composant C. Ce dernier va donc pouvoir reprendre et calculer lui aussi son premier pas et répondre au composant B. De même, le composant A pourra répondre au trigger qu'il a fini son travail.

[2]Un trigger est fourni par le SDK et est utilisable via le OpenMI Configuration Editor

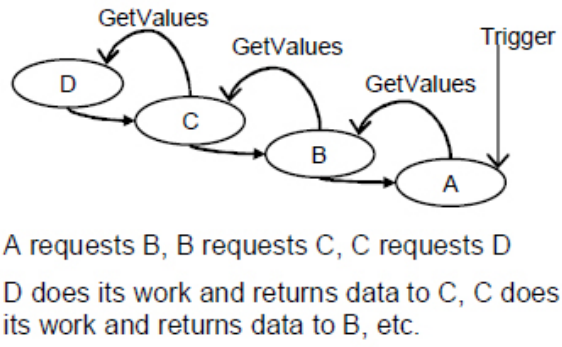

FIG. 11 – Communication par requête/réponse [8]

Nous avons vu précédemment que chacun des liens est unidirectionnel. Il est toutefois possible de créer deux liens entre deux composants, créant ainsi en quelque sorte un lien bidirectionnel. Prenons la figure 12 comme exemple. Nous n'avons ici plus que trois composants, mais un lien allant de B vers C a été ajouté.

Le début du processus sera donc identique, arrêtons-nous donc lorsque le composant C reçoit la requête du composant B. Celui-ci ne pourra répondre directement au composant B car il lui manque justement une donnée de celui-ci. Le standard a donc été conçu pour contrer ce type de blocage. A ce moment, le composant C va donc alors renvoyer tout de même une requête au composant B. Ce dernier va alors être contraint à extrapoler ses sorties pour répondre au composant C. Ensuite, le composant C pourra calculer son premier pas et répondre à B. Le composant B exécutera donc son premier pas avec ce résultat et pourra répondre alors au composant A comme dans le cas précédent.

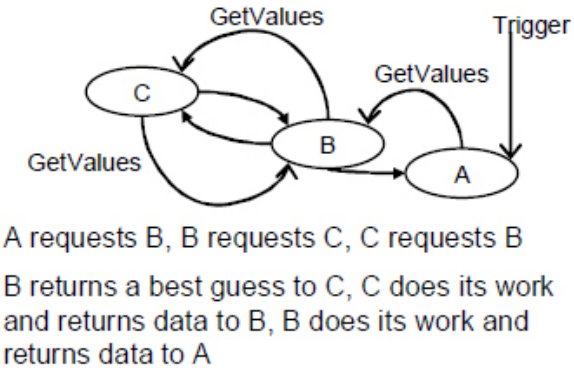

FIG. 12 – Communication par requête/réponse bidirectionnelle [8]

Sur la base de cette manière de faire, finalement très simple, le standard OpenMI va pouvoir procéder à l'échange des données. On notera toutefois que grâce à cette simplicité, l'implémentation d'OpenMI sera aisée pour les développeurs tout en garantissant la possibilité de réaliser des constructions complexes de modèles intégrés.

Voyons maintenant plus en détail comment les composants vont traiter les requêtes. La figure 13 nous illustre un peu plus précisément ce qu'il se passe lors de la réception d'une de ces requêtes par un composant utilisant le SDK.

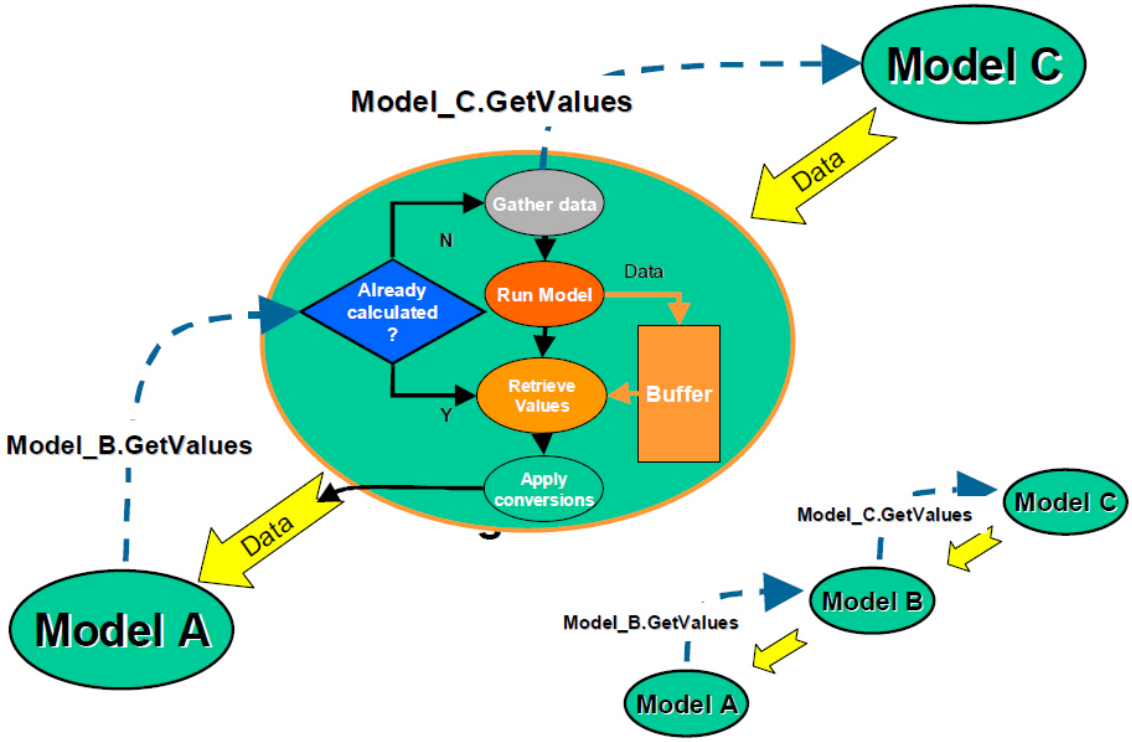

FIG. 13 – Processus de récupération des valeurs échangées [8]

Pour bien en comprendre le fonctionnement, il est important de se rappeler en premier lieu qu'un même composant peut recevoir des liens provenant de plusieurs autres composants. Dans le cadre de la figure 13, le composant B aurait pu devoir fournir des sorties à d'autres modèles. Il paraît alors évident que le composant B n'exécutera son pas de temps qu'une seule fois. La réception de la requête de A par B débutera par un premier test pour voir si le pas de temps demandé à déjà été exécuté ou non.

Partons du principe que le composant B n'a pas encore exécuté son pas de temps demandé. Nous entrons alors dans le bloc *Gather data*. Dans ce bloc, le composant B va se demander ce qu'il a besoin pour exécuter son pas de temps. C'est donc à ce moment que le composant B va rafraîchir ses liens d'entrée pour le temps requis à l'exécution du pas de temps requis. Le composant est alors en mesure de calculer son pas dans le bloc *Run Model*. Une fois que cela est fait, ses sorties sont placées dans un buffer OpenMI. Nous arrivons ainsi dans le bloc suivant *Retrieve Values*. Dans notre cas présent, les données ne font qu'y passer. Mais dans le cas où le composant n'aurait pas eu a exécuter de pas de temps, ce bloc servirait à récupérer les données dans le buffer. Le dernier bloc enfin *Apply conversions* servira

à convertir éventuellement les unités du composant B pour le composant A sur base des facteurs de conversion vers le SI[3]. La valeur ainsi directement utilisable par le composant A lui est envoyée.

3.2.7 Déploiement de composants OpenMI

Chacun des composants OpenMI va donc se comporter d'une manière standardisée. On peut décomposer le déploiement des composants OpenMI suivant six phases distinctes comme l'illustre la figure 14.

La première phase sera donc celle d'initialisation. A la fin de cette phase, le composant doit être capable de débuter des interactions avec d'autres composants. Cette phase peut se décomposer en deux étapes, à savoir l'instanciation et l'initialisation. L'instanciation du LinkableComponent se fera en utilisant l'unité logicielle telle que définie dans un fichier de configuration du composant[4]. L'étape d'initialisation quant à elle, consistera en l'exécution de la méthode initialize() telle qu'implémentée dans le composant. Cette méthode recevra des arguments provenant du même fichier de configuration dont on vient de parler. Au terme de son exécution, le composant aura chargé son modèle avec ses divers paramètres. Il devra donc avoir la connaissance de sa plage temporelle, de ses entrées et de ses sorties.

La phase suivante est celle d'inspection et de configuration. Trois étapes-clé peuvent y être observées. Tout d'abord, il sera demandé aux composants de lister leurs ExchangeItem. Ensuite des liens seront créés et ajoutés aux divers composants qui devront être mis en relation. Une étape de validation s'enchainera alors pour affirmer le statut des composants et des liens.

Vient ensuite la phase de préparation. Théoriquement, au cours de cette phase chacun des composants pourra établir d'éventuelles connexions vers une base de données, une station de surveillance ou autre. Ils pourront également ouvrir leur fichier d'entrée et de sortie, organiser leur buffer, etc. On remarquera qu'en pratique, bien des développeurs incluent ces étapes dans la phase d'initialisation.

Prend place ensuite la phase la plus importante, celle de calcul, d'exécution. Cette phase peut être assimilée à une grande boucle au cours de laquelle chacun des pas des composants sera exécuté en garantissant le bon échange des données requises entre les composants.

[3]Système International d'unités

[4]Chaque composant doit être fourni avec un fichier de configuration écrit en XML portant l'extension omi. Son comportement sera détaillé par la suite

Directement après cette phase d'exécution arrive la phase de finalisation. Durant cette phase, chaque composant pourra fermer ses buffers, ses fichiers d'entrées, de sorties, ses connexions distantes, ...

Pour terminer, vient alors la phase de désallocation. Durant cette phase, la mémoire sera déchargée de tout objet restant, provenant du déploiement.

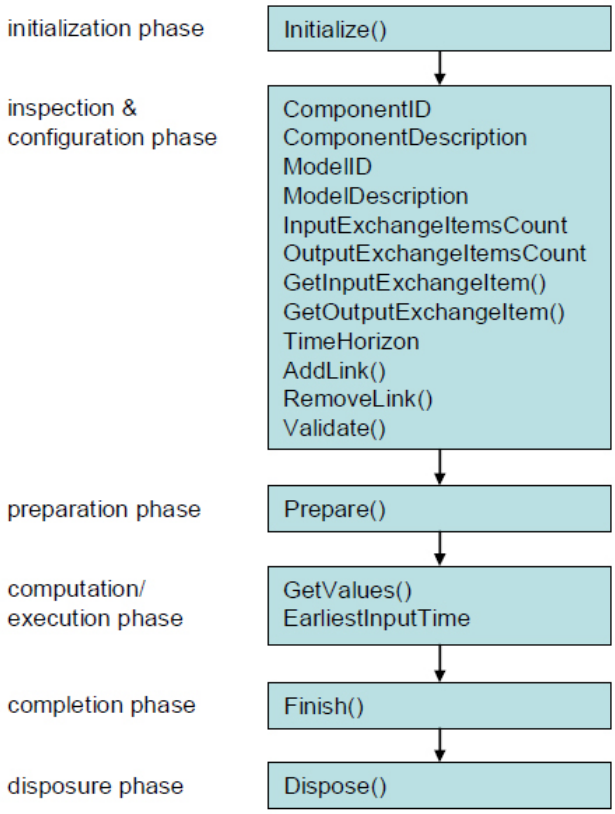

FIG. 14 – Phases de déploiement de composants OpenMI [8]

3.2.8 En pratique

Comme nous l'avions abordé précédemment, un des objectifs essentiels de OpenMI est la simplicité d'utilisation. Cette simplicité commence avant tout par l'installation. Depuis la version 1.4, l'ensemble des besoins logiciels doit être fourni par le logiciel de simulation. L'utilisateur devra donc dans un premier temps simplement installer les différents simulateurs *OpenMI-Compliant* sur sa machine, comme il l'aurait fait avec un simulateur non *OpenMI-Compliant*.

Le simulateur doit être fourni avec un fichier de configuration OpenMI. La constitution de ce fichier sera abordée plus en détails par la suite. On notera juste pour l'instant que ce fichier reprendra un lien vers l'exécutable du modèle

ainsi que divers paramètres nécessaires au bon fonctionnement du modèle au sein d'OpenMI et que ces derniers diffèrent en fonction des simulateurs utilisés. Ce fichier est sous la forme de fichier XML et est donc facilement modifiable par un utilisateur lambda[5]. Ce fichier sert de point d'entrée au composant OpenMI .

Le *OpenMI Software Development Kit* inclut alors un éditeur graphique de configuration appelé OpenMI Configuration Editor [6]. Ce dernier permet la construction et l'exécution de modèle intégré.

La première étape lors de l'élaboration de modèle intégré à l'aide du OpenMI Configuration Editor consiste au chargement des différents composants OpenMI. Ce chargement se fait en sélectionnant le fichier de configuration XML du composant. Une fois chargé dans le OpenMI Configuration Editor, on est en mesure d'ajouter les liens unidirectionnels entre les différents composants. Cette étape, telle que la figure 15 en montre le résultat, déterminera simplement quel composant enverra des données vers tel autre composant. En cas de besoin de lien bidirectionnel, deux liens devront être créés entre les composants.

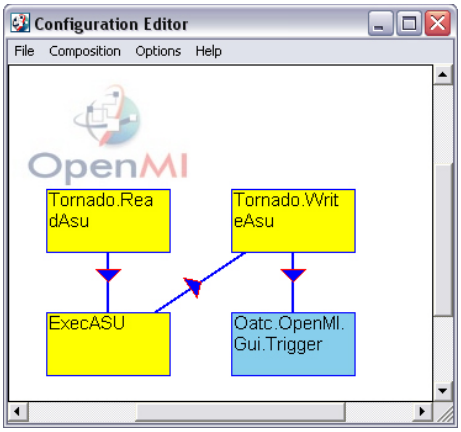

FIG. 15 – OpenMI Configuration Editor - Construction de modèles intégrés

L'étape suivante, illustrée par la figure 16, consiste en l'édition des liens entre les composants. Les composants sources et les composants récepteurs ayant été définis à l'étape précédente, il conviendra de relier les sorties du composant source avec les entrées du composant récepteur. Le OpenMI Configuration Editor fournit donc un bon affichage graphique nous permettant de visualiser chacune des variables en spécifiant les diverses caractéristiques les définissant (Unités,

[5]L'expression d'utilisateur lambda fait référence à un utilisateur semblable à la majorité dans son comportement. Le plus souvent, on désigne par là un utilisateur qui ne fait pas usage de fonctionnalités avancées, qui ne cherche pas à comprendre le fonctionnement du système, ou qui n'a pas une connaissance poussée dans le domaine concerné [6].

[6]Les fichiers d'installation pour cet éditeur peuvent être déchargés depuis l'adresse suivante : http ://sourceforge.net/projects/openmi/

dimensions, géo-référencement, ...). Dans le cadre de données géo-référencées, une représentation graphique de leur emplacement est même prévue.

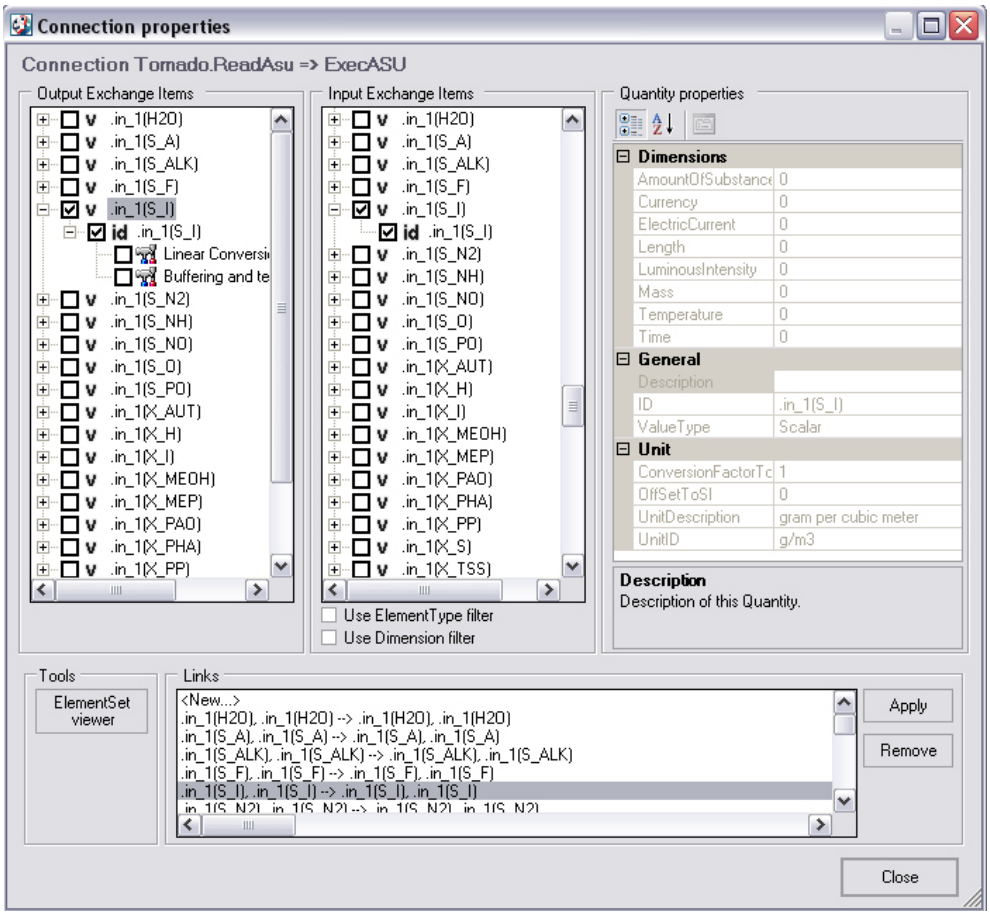

FIG. 16 – OpenMI Configuration Editor - Edition des liens

Une fois que l'ensemble de liens requis est configuré, on peut procéder au lancement des simulations. La figure 17 illustre l'ensemble des options disponibles au lancement. La plupart d'entre elles concerneront la gestion des évènements utiles au débogage. On notera au passage que les accès disques étant très lents, l'utilisation de la journalisation affectera fortement les performances globales des simulations. L'option *Invoke Trigger At* nous permettra de fixer le temps de fin de la simulation[7].

Le OpenMI Configuration Editor propose évidemment de sauvegarder l'ensemble de la configuration réalisée. Les modèles ainsi que les liens et leurs contenus qui les relient pourront ainsi être conservés, de même que les paramètres du lancement de la simulation. Cette configuration sera sauvegardée sous format XML avec l'extension opr.

[7]On rappellera que chacun des composants doit être en mesure de s'initialiser et de se rendre à la valeur qu'on leur demande

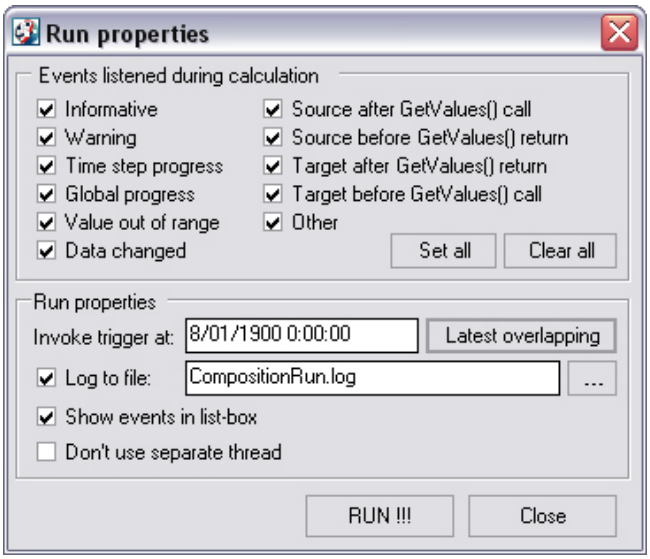

FIG. 17 – OpenMI Configuration Editor - lancement de la simulation

4 Développement de l'interface OpenMI sur Tornado

4.1 Introduction

Maintenant que nous en savons plus sur le simulateur Tornado et sur le standard OpenMI, nous allons pouvoir aborder la partie essentielle du présent document, à savoir l'implémentation d'une interface OpenMI sur Tornado.

Il est important de rappeler que le standard en lui-même ne se compose que d'interfaces pour uniquement nous décrire une manière de faire, sans nous dire comment le faire. La charge de travail peut sembler de prime abord relativement conséquente. Toutefois, le SDK que nous avons déjà évoqué précédemment va nous aider dans notre tâche. Celui-ci va reprendre en effet une implémentation par défaut de la plupart des interfaces dont nous avons besoin. Les interfaces IElementSet, IQuantity et IValue par exemple requièrent somme toute une implémentation assez générique, peu importe les simulateurs. Il conviendra donc de très bien utiliser leur implémentation.

On serait alors en droit de se demander dans quel but une distinction a été apportée entre le standard et le SDK. La réponse est pourtant simple et se rapporte une fois de plus au besoin de généricité et de flexibilité de OpenMI afin de pouvoir s'appliquer à tout simulateur temporel. Lorsqu'un développeur n'est pas satisfait de l'implémentation d'une partie d'OpenMI dans le SDK, libre à lui d'en faire sa propre implémentation. Mais le carcan imposé par les interfaces du standard assurera une pleine compatibilité entre les composants utilisant l'entièreté SDK, une partie seulement ou ne l'utilisant pas du tout.

L'implémentation de l'interface pour Tornado ne nécessitant pas d'adaption particulière, le SDK y sera utilisé largement. La figure 18 nous illustre ainsi l'imbriquement de la partie principale du SDK dans le standard.

Le point de départ requis pour être *OpenMI-compliant* est d'implémenter l'interface ILinkableComponent. Pour y parvenir, la documentation [8] nous suggère d'écrire une classe qui sera le composant OpenMI. Cette classe doit hériter de la classe LinkableEngine. Le seul besoin de cette classe est que la méthode SetEngineApiAccess y soit redéfinie. Cette méthode ne sert qu'à attribuer à notre composant un *Engine*. Ce dernier correspond à une autre classe que nous devrons implémenter et qui devra implémenter l'interface IEngine. L'implémentation de cette interface est très importante puisqu'elle devra reprendre la définition des

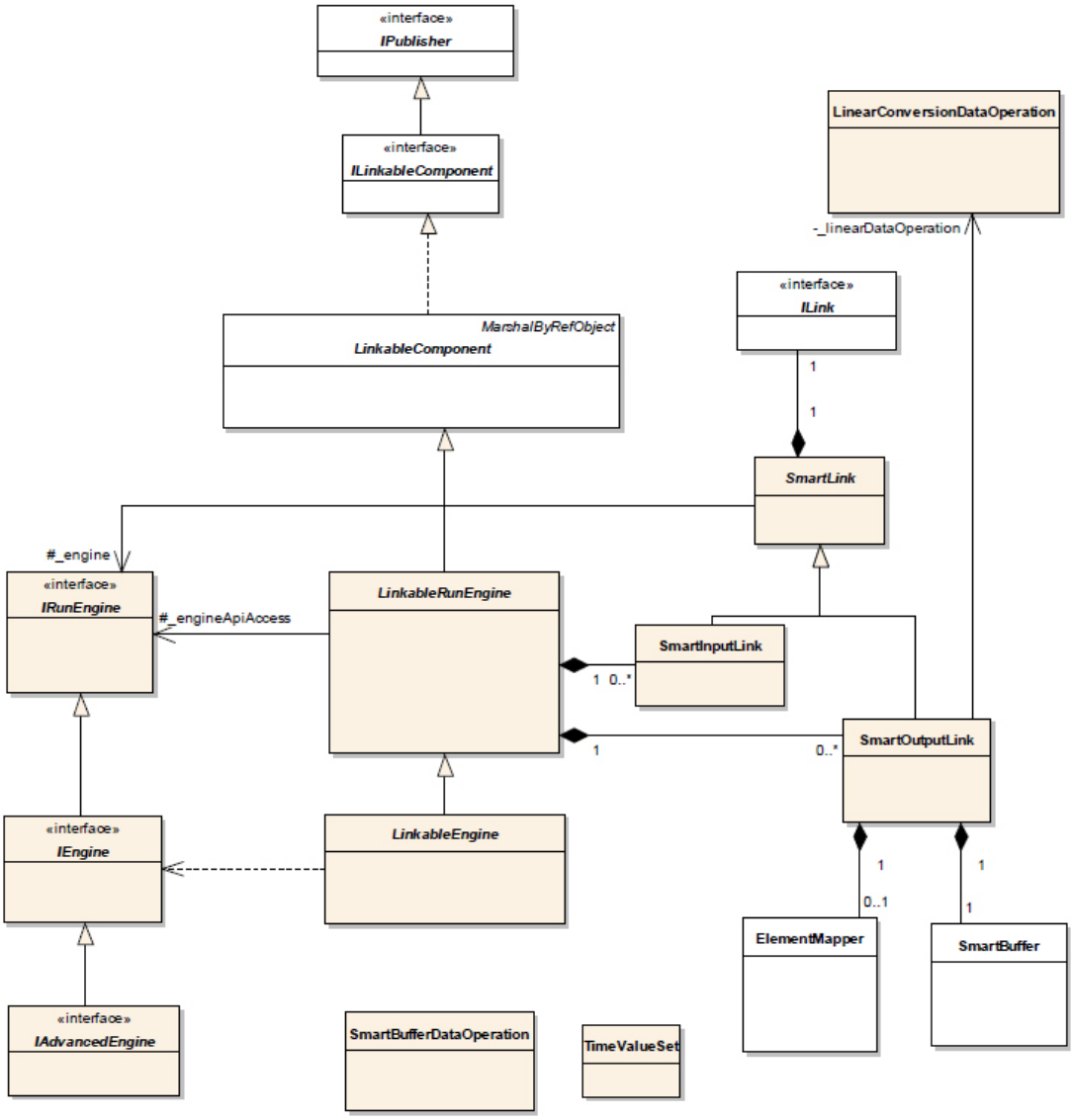

FIG. 18 – Diagramme de classe - Architecture du standard et du SDK [12]

méthodes permettant le fonctionnement de Tornado en tant que composant OpenMI. Les méthodes permettant de récupérer les données de sorties et de fixer des données d'entrées devront notamment y être définies. Cette classe est souvent qualifiée de *Wrapper class*. On peut la résumer en quelque sorte en une version standardisée de Tornado, elle sert de relais entre le fonctionnement d'OpenMI et du noyau Tornado.

Le cheminement depuis l'interface ILinkableComponent jusqu'à cette implémentation de IEngine est justifié plus par des raisons historiques que techniques et ne sera donc pas abordé dans le présent document.

De cette architecture, nous retiendrons donc que l'implémentation d'une interface OpenMI avec l'aide du SDK se résume au développement de deux classes.

La première est celle qui devra être référencée par le fichier de configuration OpenMI. Elle doit hériter de LinkableEngine[8] et simplement redéfinir la méthode SetEngineApiAccess. La seconde classe à implémenter correspond quant à elle à l'*Engine* qui sera attribuée à la classe précédente. Cette seconde classe doit implémenter l'interface IEngine[9].

4.2 Composants Tornado OpenMI

4.2.1 Introduction

En réalité, ce n'est pas un composant OpenMI qui a été développé sur Tornado mais trois composants bien distincts. Chacun d'entre eux a pris naissance pour répondre à une tâche bien définie.

Le premier vise uniquement à charger des fichiers d'entrées formatés pour Tornado. Le second se charge d'écrire un fichier de sorties avec le contenu qu'il recevra. Le troisième est, quant à lui, le plus important, car il permettra l'exécution des modèles.

4.2.2 Reader

Tornado utilise bien évidemment ses propres fichiers d'entrées. Ceux-ci reprennent le nom des variables ainsi que leurs unités. La figure 19 reprend un extrait du fichier d'entrées de la simulation ASU proposée avec Tornado, seules les six premières entrées parmi les vingt, y sont représentées. Le composant Reader aura donc comme objectif de charger un fichier d'entrées dans un buffer de Tornado et d'en récupérer le contenu. Ce composant pourra ainsi communiquer les valeurs du fichier d'entrées à d'autres composants OpenMI suivant le standard. Grâce à ce composant les fichiers d'entrées de Tornado peuvent être exploités par d'autres simulateurss*OpenMI-compliant*. Cette exploitation pourra se faire ainsi en profitant notamment des processus de conversion et d'interpolation assumés par le standard.

Ainsi le composant Reader aura recours aux deux classes suivantes :

- TornadoOMI.ReadLinkableComponent[10] qui hérite de Oatc.OpenMI.Sdk.Wrapper.LinkableEngine
- TornadoOMI.CReadComponent[11] qui implémente l'interface Oatc.OpenMI.Sdk.Wrapper.IEngine

[8] Provenant du SDK à l'emplacement suivant : Oatc.OpenMI.Sdk.Wrapper.LinkableEngine
[9] Provenant du SDK à l'emplacement suivant : Oatc.OpenMI.Sdk.Wrapper.IEngine
[10] voir figure 21
[11] voir figure 20

```
%%Version3.3
%%BeginHeader
t    in_1(H2O)    in_1(S_I)    in_1(S_O)    in_1(S_N2)   in_1(S_A)    in_1(S_F)
d    m3/d     g/m3     g/m3     g/m3     g/m3     g/m3     g/m3     g/m3
%%EndHeader
0    21477    75   0.001    0.001    62.46546     124.93092
0.01042 21474    75   0.001    0.001    53.74197     107.48394
0.02083 19620    75   0.001    0.001    54.44886     108.89772
0.03125 19334    75   0.001    0.001    50.06769     100.13538
0.04167 18978    75   0.001    0.001    46.35457     92.70914
0.05208 18321    75   0.001    0.001    44.35819     88.71638
0.0625  17855    75   0.001    0.001    42.57931     85.15862
0.07292 17237    75   0.001    0.001    40.00033     80.00066
0.08333 16453    75   0.001    0.001    40.38873     80.77746
```

FIG. 19 – Extrait du fichier d'entrée de la simulation ASU

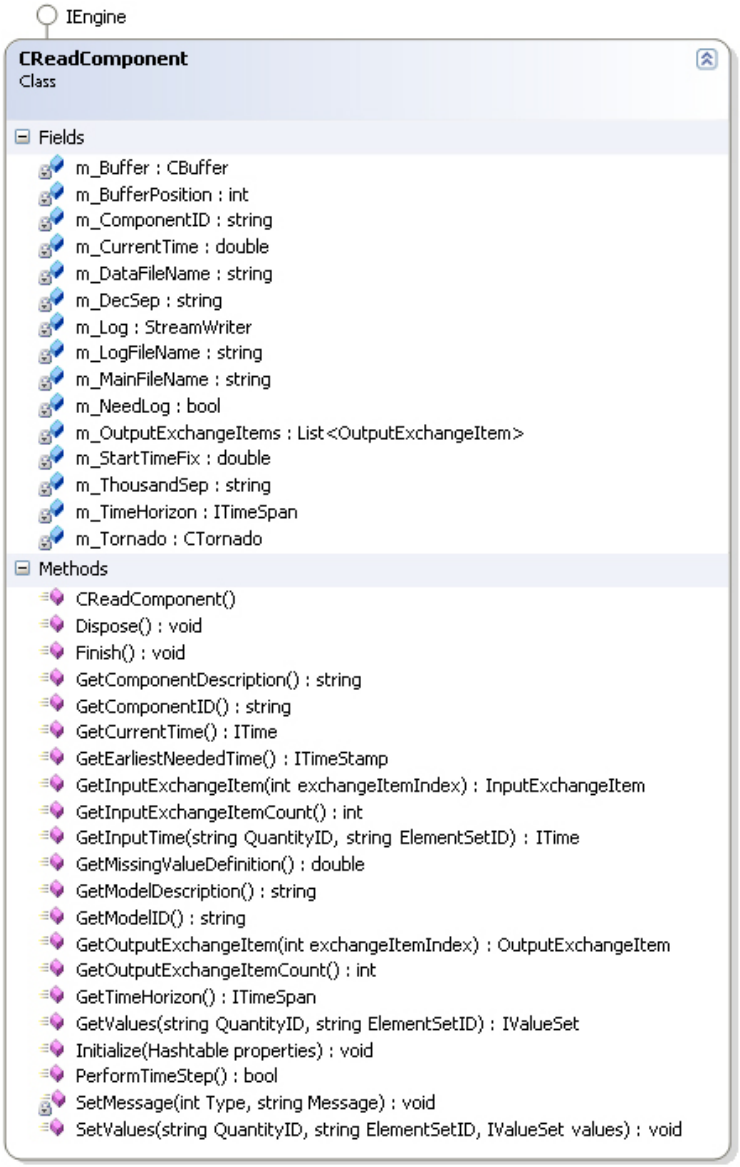

FIG. 20 – Diagramme de classe - CReadComponent

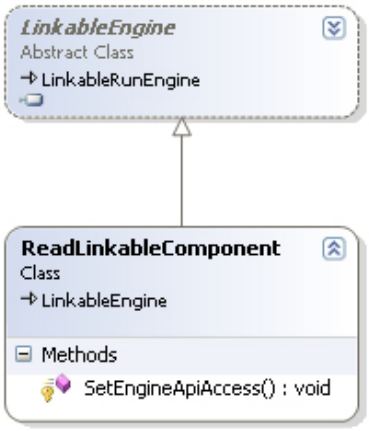

FIG. 21 – Diagramme de classe - CReadComponentLinkableComponent

4.2.3 Writer

Le composant Writer répond pour sa part à un besoin probablement moins important mais fournira ainsi aux utilisateurs un service complet. Son but est donc de pouvoir retranscrire ses entrées dans un fichier de sorties. Il s'exécute suivant un pas fixé par l'utilisateur et c'est ce pas qui fixera l'intervalle entre chaque valeur du fichier de sorties. De part sa nature de simple retranscription, il ne nécessite pas l'utilisation de Tornado. Comme nous le verrons par la suite, ce composant ne permet toutefois pas de conversion d'unité.

Ainsi le composant Writer aura recours aux deux classes suivantes :

- TornadoOMI.WriteLinkableComponent[12] qui hérite de Oatc.OpenMI.Sdk.Wrapper.LinkableEngine
- TornadoOMI.CWriteComponent[13] qui implémente l'interface Oatc.OpenMI.Sdk.Wrapper.IEngine

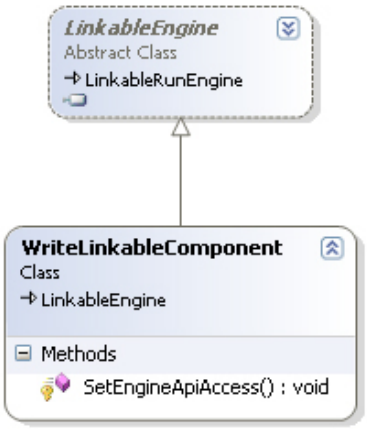

FIG. 22 – Diagramme de classe - CWriteComponentLinkableComponent

[12] voir figure 22
[13] voir figure 23

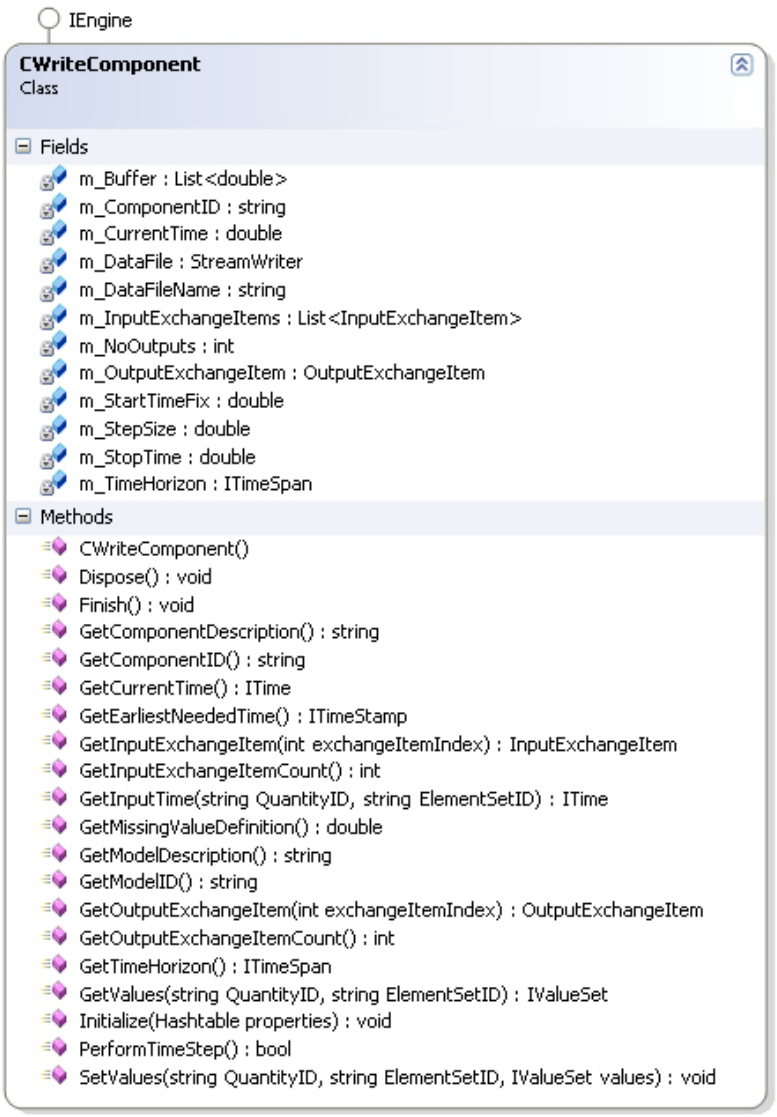

FIG. 23 – Diagramme de classe - CWriteComponent

4.2.4 Exec

Le composant Exec correspond pour sa part à la partie essentielle du travail. Il a été développé dans le but de pouvoir exécuter les modèles des utilisateurs. Il offre ainsi les flexibilités des modèles utilisés classiquement par Tornado. Ses variables d'entrées peuvent ainsi provenir de fichier d'entrées ou d'autres composants OpenMI. De même, ses variables de sorties peuvent se voir exploitées par un fichier de sorties, par d'autres composants OpenMI ou bien par les deux. Comme nous l'avions déjà abordé, Tornado utilise un fichier XML servant à la description de la simulation. Dans le cas d'utilisation d'autres composants comme source, aucune spécification d'entrée ne doit être fournie dans le fichier XML concernant les variables d'entrées. Concernant les fichiers de sorties, l'utilisation du fichier XML reste identique à l'utilisation de Tornado hors du cadre d'OpenMI .

Dans le but de simplification, le comportement de OpenMI et de Tornado a été cloisonné dans deux classes distinctes. Comme dans les autres cas, le point d'entrée de notre interface sera la classe héritant de LinkableEngine provenant du SDK et utilisera l'implémentation de IEngine. Cependant, l'implémentation de IEngine utilisera une classe de connexion avec Tornado pour le faire fonctionner. Le code propre à OpenMI et à l'API de Tornado a pu être ainsi séparé et rendre la maintenance plus aisée.

Ainsi le composant Exec aura recours aux trois classes suivantes :

- TornadoOMI.ExecLinkableComponent[14] qui hérite de Oatc.OpenMI.Sdk.Wrapper.LinkableEngine
- TornadoOMI.CExecComponent[15] qui implémente l'interface Oatc.OpenMI.Sdk.Wrapper.IEngine
- TornadoOMI.CExecConnector[16] qui sert de liaison avec le noyau Tornado

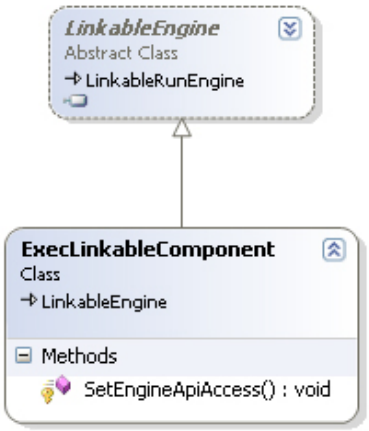

FIG. 24 – Diagramme de classe - CExecComponentLinkableComponent

[14]voir figure 24
[15]voir figure 25
[16]voir figure 26

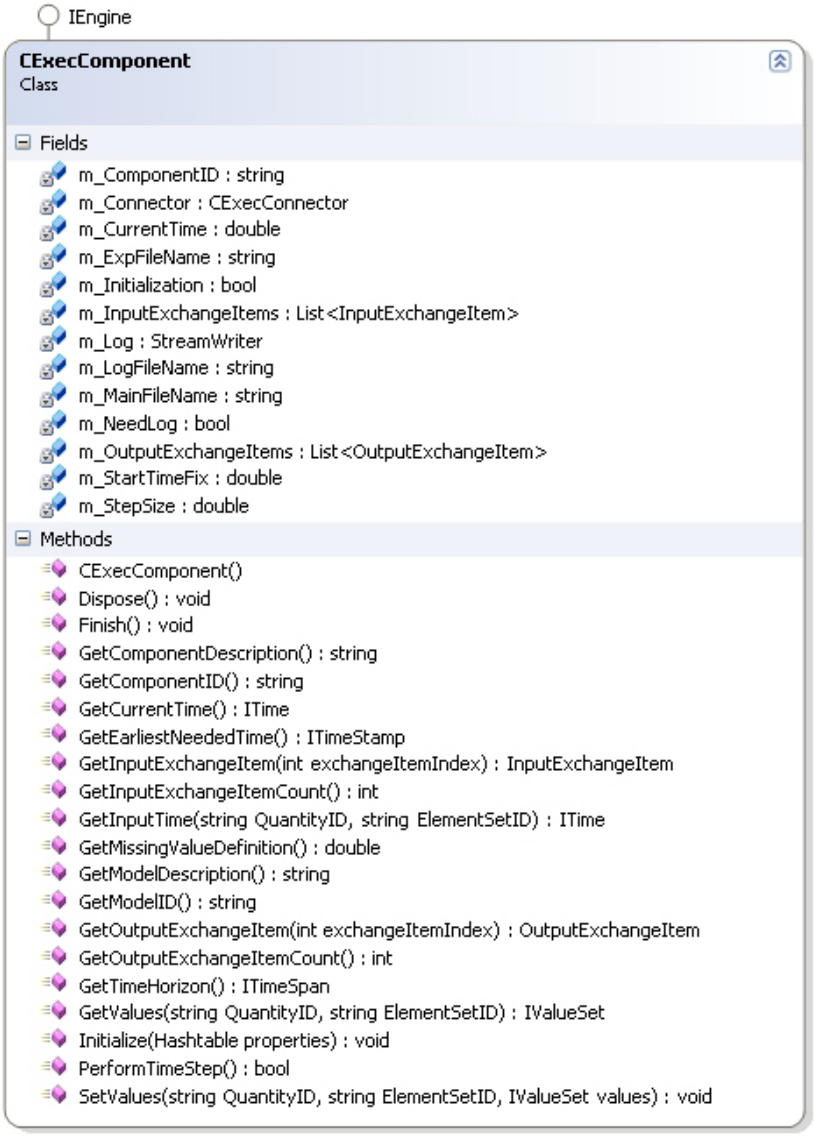

FIG. 25 – Diagramme de classe - CExecComponent

4.3 Journalisation

Dans son implémentation, Tornado propose la génération d'évènements. Ceux-ci peuvent être lancés dans le but d'informer, d'avertir ou de signaler une erreur. Ces évènements permettent ainsi à l'utilisateur de mieux comprendre ce qui s'est passé lors de l'exécution de la simulation. Ils ne sont produits que dans le cadre de Tornado. L'utilitaire de configuration graphique, OpenMI Configuration Editor , reprend quant à lui sa propre gestion d'évènements liés au processus OpenMI.

Partant du constat que la journalisation des évènements Tornado ne serait éventuellement utile qu'en cas de développement de la simulation et que les accès disque sont extrêmement lents, la journalisation de ces évènements est proposée en option. Ainsi, dans le fichier de configuration omi, un champ pour spécifier un nom

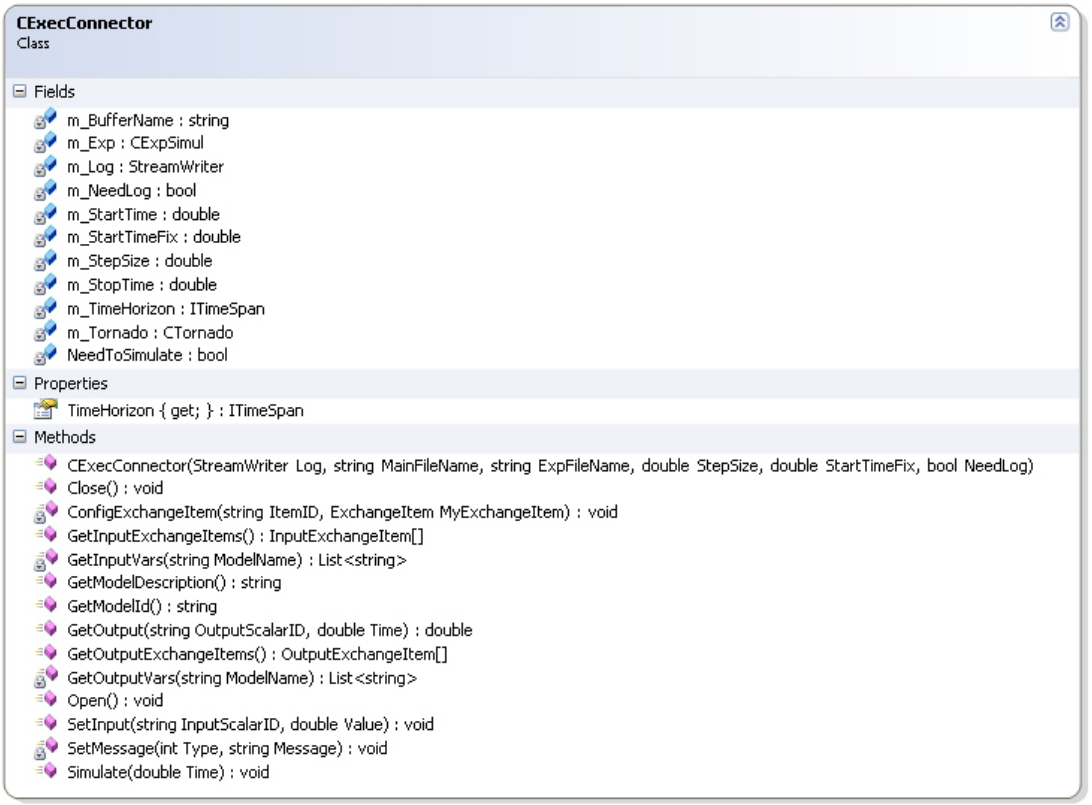

FIG. 26 – Diagramme de classe - CExecConnector

de fichier pour le journal est prévu. S'il est présent lors de la phase d'initialisation, la journalisation des évènements Tornado sera activée. Dans le cas contraire, les évènements ne seront pas journalisés.

4.4 Implémentation de la Wrapper class

4.4.1 Gestion des ExchangeItem

Un des premiers rôles que doit remplir la Wrapper class est la gestion des ExchangeItem. La classe reprend donc en attribut une liste d'InputExchangeItem et une liste d'OutputExchangeItem. Ces deux listes nous permettent ainsi d'implémenter les quatre méthodes requises par l'implémentation de l'interface IEngine à savoir :

- int GetInputExchangeItemCount()
- InputExchangeItem GetInputExchangeItem(int exchangeItemIndex)
- int GetOutputExchangeItemCount()
- OutputExchangeItem GetOutputExchangeItem(int exchangeItemIndex)

La première et la troisième méthode sont chargées de retourner respectivement le nombre d'entrées et de sorties gérées par le composant OpenMI. La seconde et la quatrième méthode quant à elles, sont chargées de retourner respectivement l'InputExchangeItem et l'OutputExchangeItem correspondant à l'index demandé. L'implémentation de ces méthodes est très aisée grâce aux deux listes membres de la classe.

On notera cependant une distinction entre les différents composants développés. Le Reader par principe n'aura pas besoin d'InputExchangeItem. Ses méthodes auront donc été adaptées en tenant compte de ces différences. Le Writer pour sa part n'aurait besoin d'aucun OutputExchangeItem. Pour toutefois pouvoir être connecté avec le trigger, le Writer a été pourvu d'un OutputExchangeItem vide dans ce but.

De part le caractère dynamique de Tornado, qui permet la simulation de modèles différents, c'est au moment de la phase d'initialisation que la liste des InputExchangeItem et des OutputExchangeItem sera construite.

4.4.2 Gestion du temps au sein des composants

4.4.2.1 Echelle de temps relative et absolue

Un des aspects principaux du développement d'une interface OpenMI est la standardisation du fonctionnement de son simulateur. Le lien, parmi chacun des composants mis en relation, est le temps. Celui-ci se voit donc attribué une importance capitale. Afin de garantir simplicité pour le plus grand nombre de simulateurs, OpenMI utilise un temps absolu. Il se caractérise ainsi en années, mois, jours, heures, minutes et secondes. La garantie d'une bonne synchronisation se fera grâce à l'utilisation du temps de manière absolue. En opposition, Tornado utilise un temps relatif. Ses modèles s'exécutent donc toujours en commençant du temps zéro pour se finir au temps défini par l'utilisateur, configuré généralement en nombre de jours.

Un facteur de conversion était donc nécessaire pour fixer une correspondance entre les temps d'OpenMI et ceux de Tornado. Ce facteur étant dépendant des cas d'utilisation, il ne pouvait être fixé dans l'interface. Un paramètre a donc été prévu dans le fichier de configuration omi du composant. Ce paramètre a pour nom "StartTimeFix" et devra sous forme de chaîne de caractères reprendre une date telle qu'utilisée par OpenMI. Cette date définira ainsi le facteur de translation entre le temps zéro de Tornado avec une date absolue. Chaque

interaction temporelle entre OpenMI et Tornado devra donc subir une conversion. La classe TornadoOMI.DateTimeConverter a donc été écrite pour se charger de cette conversion. Elle possède une méthode pour convertir une date d'OpenMI en Tornado et de même une autre méthode pour faire le travail inverse.

4.4.2.2 Méthodes relatives au temps

Après cette mise au point sur l'échelle de temps, il faut en comprendre sa gestion. Habituellement, les modèles sont conçus pour fonctionner sur des plages de temps bien fixées. Certains modèles sont obligés de commencer leur simulation à un temps bien précis. Chaque composant OpenMI doit donc être caractérisé par une plage fréquentielle. Ainsi la méthode GetTimeHorizon doit être implémentée dans la Wrapper class. Cette méthode retourne un objet de type TimeHorizon implémenté par le SDK. Il contient la date de début et la date de fin de l'exécution du modèle. Cette méthode utilisera bien évidemment un attribut de la classe et cet attribut sera construit lors de la phase d'initialisation. On notera également que le composant Writer pourrait très bien utiliser l'échelle de temps absolue comme le fait OpenMI , étant donné que comme nous le verrons, il n'utilise pas le noyau Tornado. Faisant toutefois partie de l'ensemble de composant Tornado, il semblait important qu'il en reprenne les mêmes principes, dont celui du temps relatif.

Une autre méthode importante, relative au temps, est la méthode GetCurrentTime. Dans le cadre d'OpenMI, nous avons vu que les simulateurs étaient exécutés pas à pas. Il était donc important d'avoir une méthode permettant à tout moment de savoir où se situe le simulateur sur sa plage temporelle.

4.4.2.3 Démarrage des modèles

Un aspect particulièrement important, relatif au temps et surtout au TimeHorizon d'un composant, est son utilisation lors du processus de lancement du composant par le SDK. Il est en effet attendu par le standard que chaque composant soit en mesure de s'initialiser seul.

En rapport à cela, revenons juste un instant sur le TimeHorizon. Ce dernier définit donc un espace temporel borné. La propriété du composant CurrentTime se promènera donc sur cet espace sans jamais pouvoir en sortir sous peine de générer une exception. Par définition, le CurrentTime prendra comme première valeur la borne inférieure du TimeHorizon pour avancer par la suite pas à pas jusqu'à la borne supérieure. Avant de démarrer l'exécution propre du modèle et partant du principe que chacun des composants doit être en mesure de s'initialiser seul, il sera

demandé au composant de fournir une première valeur de sortie pour chacun de ses OutputExchangeItem pour la valeur de temps initial du composant. Le but de ce processus est de permettre au SDK d'initialiser une première fois ses buffers avec une valeur.

Voilà un élément qui va à l'encontre du fonctionnement de Tornado. En effet, la simulation d'un modèle dans Tornado débute au temps relatif zéro. Seulement, Tornado a besoin de lire les valeurs d'entrées pour le temps zéro afin de pouvoir calculer les valeurs de sorties pour ce temps zéro. Pour résoudre le problème, le TimeHorizon du composant a été reculé d'un pas par rapport à la description du modèle exécuté par Tornado. De plus, la méthode GetValue a été adaptée pour retourner arbitrairement la valeur zéro lors de son appel en cours d'initialisation et pour le temps plus petit d'un pas par rapport au temps de démarrage décrit dans le modèle.

La relation entre les modèles ne se verra ainsi pas affectée par cette adaptation. La première valeur placée dans le buffer par les composants Tornado ne sera pas demandée par les autres composants mis en relation. En effet, les utilisateurs auront conçu leurs composants en fonction de la plage de temps référencée dans le descriptif du modèle Tornado.

En rapport à ceci, il est donc important de comprendre que lors de l'exécution des différents composants, l'un d'entre eux peut-être interrogé sur une de ses sorties pour un temps correspondant au milieu de sa plage de temps. Le composant doit pouvoir ainsi se rendre au moment demandé en débutant depuis la borne inférieur de son TimeHorizon.

4.4.3 Initialisation

4.4.3.1 Introduction

Comme nous l'avons vu jusqu'à présent, la phase d'initialisation est particulièrement importante dans l'aspect dynamique de Tornado. Cette phase va se constituer par la simple exécution de la méthode initialize de la Wrapper class. Elle constituera notre seule occasion de construire les propriétés propres du composant à savoir sa plage temporelle, ses InputExchangeItem et ses OutputExchangeItem. La méthode initialize sera également celle qui récupérera les paramètres contenus dans le fichier de configuration omi.

La composition de cette méthode diffèrera fortement dans son implémentation

en fonction des composants, sa description sera donc décomposée en trois sous-chapitres.

4.4.3.2 Reader

La première étape de la méthode consistera en l'analyse des paramètres récupérés du fichier de configuration omi, voir par exmple le listing 1. En voici la liste pour le Reader :

- ComponentID : identifiant du composant OpenMI
- MainFileName : lien vers le fichier de configuration de Tornado
- LogFileName : lien optionnel vers le fichier journal Tornado
- DataFileName : lien vers le fichier d'entrées
- DecSep : indicateur de séparateur des décimales
- ThousandSep : indicateur de séparateur des milliers
- StartTimeFix : facteur d'échelle temps relatif/absolu

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.ReadLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="Tornado.ReadAsu"/>
    <Argument Key="MainFileName" ReadOnly="true" Value="Tornado.Main.xml"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogReadAsu.txt"/>
    <Argument Key="DataFileName" ReadOnly="true" Value="ASU.Simul.in.txt"/>
    <Argument Key="DecSep" ReadOnly="true" Value="."/>
    <Argument Key="ThousandSep" ReadOnly="true" Value=","/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
  </Arguments>
</LinkableComponent>
```

Listing 1 – Fichier omi pour la lecture du fichier d'entrées d'ASU

Un traitement par expression régulière sera alors exécuté sur la propriété StartTimeFix afin d'en convertir la valeur en double.

Ensuite, le noyau de Tornado sera instancié et initialisé à l'aide de son fichier de configuration fourni dans les paramètres. Après cela, un buffer sera ajouté au noyau de Tornado, ce qui permettra d'y charger le fichier d'entrées. En fonction de la présence ou non d'un lien vers un fichier de journalisation dans les paramètres, un écouteur d'évènements fixé sur le noyau de Tornado et sur le buffer sera ajouté et se verra attribué la méthode SetMessage. Cette dernière sert à l'écriture de ces évènements dans le journal.

Une fois ce travail effectué, nous pourrons récupérer sur base du contenu du buffer la plage de temps du fichier d'entrées. L'API .NET de Tornado prévoit cela très simplement. L'attribut privé m_CurrentTime de la classe pourra alors se voir attribuer la valeur du premier temps du buffer, à savoir zéro, étant donné que Tornado utilise une échelle de temps relative. La méthode GetCurrentTime pourra ainsi être opérationnelle et se charger de la conversion entre le temps relatif et absolu en fonction du facteur de conversion calculé précédemment.

La dernière étape de la méthode initialize consistera en la conception des ExchangeItem. Comme nous l'avons abordé précédemment, le reader n'a pas besoin d'entrée. Cette étape ne concernera donc que les OutputExchangeItem. Une fois de plus, l'API .NET de Tornado nous fournit une série de méthodes permettant l'analyse du contenu du buffer. Grâce à celles-ci, nous allons pouvoir récupérer le nom des variables ainsi que le nom de leurs unités, ce qui nous permettra la construction des ExchangeItem. Pour construire ces derniers, nous avons donc besoin de construire un ElementSet et une Quantity pour chacun d'entre eux. En raison de l'absence de nécessité de référence spatiale dans Tornado, les ElementSet seront construits de manière très basique. Ils seront de type IDBased[17] et porteront le nom de la variable comme identifiant. De même la Quantity recevra le nom de la variable comme identifiant mais devra recevoir une unité. Comme expliqué précédemment, le nom de celle-ci est récupérée depuis le buffer. Une fonction de l'API .NET de Tornado permet ensuite à partir de ce nom de récupérer les informations relatives à cette unité, à savoir un facteur et un offset vers le Système International[18]. Sous ce principe sera donc construit l'ensemble des OutputExchangeItem ajoutés à la liste membre de la classe.

4.4.3.3 Writer

Tout comme pour le Reader, la première étape de la méthode consistera en l'analyse des paramètres récupérés du fichier de configuration omi, voir par exemple le listing 2. En voici la liste pour le Writer :

- ComponentID : identifiant du composant OpenMI
- DataFileName : lien vers le fichier de sorties
- StepSize : taille du pas en nombre de jours
- StopTime : borne temporelle supérieure du composant en nombre de jours
- NoOutputs : nombre de sorties à inscrire dans le fichier de sorties
- StartTimeFix : facteur d'échelle temps relatif/absolu

[17] Type d'ElementSet dépourvus de coordonnées spatiales en opposition par exemple à un XY-Point ou une XYPolyline.

[18] Mise en garde importante relative à cette conversion disponible à la section 4.5 à la page 45

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.WriteLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="Tornado.WriteAsu"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogWriteAsu.txt"/>
    <Argument Key="DataFileName" ReadOnly="true" Value="WriteAsu.Out.txt"/>
    <Argument Key="StepSize" ReadOnly="true" Value="0.01"/>
    <Argument Key="StopTime" ReadOnly="true" Value="100000"/>
    <Argument Key="NoOutputs" ReadOnly="true" Value="28"/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
  </Arguments>
</LinkableComponent>
```

Listing 2 – Fichier omi pour l'écriture des résultats de ASU

Cette fois-ci également, un traitement par expression régulière sera exécuté en premier lieu sur la propriété StartTimeFix afin d'en convertir la valeur en double.

Contrairement au Reader, le Writer ne requiert pas l'utilisation de Tornado. Il ne sera donc évidemment pas instancié et initialisé et aucun paramètre contenant un lien vers son fichier de configuration n'est requis.

Un flux d'écriture sera ensuite ouvert à partir du lien fourni dans les paramètres. C'est dans celui-ci que seront retranscrites les entrées du Writer.

Après cela, la page temporelle du writer sera fixée. Bien que n'utilisant pas Tornado, le Writer a été conçu dans le cadre de Tornado. Il utilisera donc un temps relatif également. Conformément à ce qui a été développé dans la section 4.4.2.3 de la page 32, la plage de fonctionnement du composant commencera donc un pas avant le temps zéro. Il prendra ensuite fin selon la valeur renseignée parmi les paramètres.

La dernière étape consiste en la création des ExchangeItem. Comme nous l'avons déjà abordé, le Writer comportera un OutputExchangeItem dans l'unique but de l'interconnecter avec un trigger. Ce dernier sera donc créé selon une construction par défaut et portant comme identifiant la référence *TriggerOutput*. La création des InputExchangeItem est quant à elle légèrement plus problématique. En effet le nombre d'entrées du composant dépendra uniquement des besoins rencontrés par l'utilisateur. Le paramètre NoOutput reprend ainsi le nombre d'InputExchangeItem qu'il conviendra de mettre en place. Seulement, à cet instant, le composant ne peut savoir ce qui sera connecté à ses InputExchangeItem. Ceux-ci se voient alors attribués également une construction par défaut et portant le numéro de leur entrée. De part cette construction par défaut, aucune possibilité de conversion d'unité ne

sera proposée à l'utilisateur.

Suite à la construction des InputExchangeItem, une en-tête sera ajoutée au fichier de sorties reprenant le numéro des variables.

4.4.3.4 Exec

Tout comme pour les précédents composants, la première étape de la méthode consistera en l'analyse des paramètres récupérés du fichier de configuration omi, voir par exmple le listing 3. En voici la liste pour l'Exec :

- ComponentID : identifiant du composant OpenMI
- MainFileName : lien vers le fichier de configuration de Tornado
- LogFileName : lien optionnel vers le fichier journal Tornado
- ExpFileName : lien vers le descriptif de l'expérience Tornado
- StepSize : taille du pas en nombre de jours
- StartTimeFix : facteur d'échelle temps relatif/absolu

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.ExecLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="ExecASU"/>
    <Argument Key="MainFileName" ReadOnly="true" Value="Tornado.Main.xml"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogExecAsu.txt"/>
    <Argument Key="ExpFileName" ReadOnly="true" Value="ASU.Simul.Exp.OMI.xml"/>
    <Argument Key="StepSize" ReadOnly="true" Value="0.01"/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
  </Arguments>
</LinkableComponent>
```

Listing 3 – Fichier omi pour simuler ASU

Une fois de plus la méthode commencera par un traitement par expression régulière sur la propriété StartTimeFix afin d'en convertir la valeur en double.

Ensuite sera instanciée la classe CExecConnector servant à faire le lien entre l'implémentation de IEngine et Tornado. Le constructeur de CExecConnector se chargera dans un premier temps d'instancier et d'initialiser Tornado à l'aide de son fichier de configuration spécifié dans les paramètres. Une fois fait, à l'aide du lien vers le descriptif de l'expérience, cette dernière sera chargée par le noyau Tornado. Ensuite en fonction de la demande de l'utilisateur pour une journalisation des évènements Tornado, une méthode d'écriture sera attribuée à l'écouteur d'évènement du noyau et de l'expérience Tornado. Cette méthode se chargera d'écrire dans le fichier journal les évènements informatifs, d'avertissements

ou d'erreurs. Puis, un buffer sera ajouté au noyau Tornado et chacune des variables de sorties de l'expérience sera attribuée au buffer. Après, la date de début et de fin de l'expérience sera récupérée de Tornado pour construire le TimeHorizon du composant. En accord avec ce qui a été mis en évidence à la section 4.4.2.3 de la page 32, le TimeHorizon débutera un pas avant celui de l'expérience. On notera que le TimeHorizon fera partie de la classe CExecConnector, la méthode getTimeHorizon de la classe CExecComponent se chargera de récupérer le TimeHorizon de son instance de CExecConnector. Dans l'optique du fonctionnement pas à pas qui sera demandé à Tornado, la date de fin de l'expérience présente dans le noyau Tornado sera modifiée pour prendre la même valeur que celle de début. L'avancement pas à pas de Tornado sera géré en effet par la modification de ces paramètres dans l'expérience au sein de Tornado. Pour finir cette instanciation de CExecConnector, l'expérience au sein de Tornado sera initialisée. Tornado étant ainsi préparé, le reste de l'initialisation du composant peut alors prendre place.

L'étape suivante consiste en la construction des ExchangeItem. Le lien qui sera présent lors de la récupération d'information concernant les variables d'entrées/sorties et la configuration des ExchangeItem entre Tornado et OpenMI sera très intime. Cet état de fait a alors encouragé à procéder à cette manœuvre au sein de la classe CExecConnector. Ce processus se déploie sur différentes méthodes. La démarche suivante illustre le processus pour les sorties, mais elle est identique pour les entrées.

1. Initialize de la classe CExecComponent va appeler la méthode GetOutputExchangeItem de son instance de CExecConnector
2. la méthode GetOutputExchangeItem va appeler deux méthodes pour construire les OutputExchangeItem
   (a) GetOutputVars de la classe CExecConnector va énumérer la liste des variables de sorties de l'expérience Tornado.
   (b) ConfigExchangeItem de la classe CExecConnector va configurer les ExchangeItem
3. la méthode GetOutputExchangeItem va pouvoir renvoyer une liste des OutputExchangeItem à l'instance de CExecComponent
4. l'instance de CExecComponent va affecter cette liste à son attribut reprennant la liste des OutputExchangeItem

On notera que lors de l'étape 2a, l'ensemble des variables de l'expérience sera énuméré. Il faut comprendre par là, l'ensemble des variables du modèle principal composant l'expérience Tornado ainsi que de l'ensemble de ses sous-modèles associés. On notera également que l'étape 2b se chargera de construire un

ElementSet et une Quantity pour l'outputExchangeItem. L'ElementSet sera de type IDBased[19]. Le noyau Tornado sera également interrogé quant à l'unité de la variable de manière à pouvoir configurer correctement la Quantity. Ensuite les identifiants de l'ElementSet et de la Quantity prendront le nom de la variable comme valeur.

Pour terminer l'initialisation, la méthode open de CExecConnector sera appelée afin d'ouvrir l'expérience dans le noyau Tornado. Cette méthode va en réalité finir le travail de préparation de Tornado pour la simulation de l'expérience.

4.4.4 Processus de réponse aux requètes

4.4.4.1 Processus standard

La phase d'exécution du modèle est la partie clé de OpenMI. Son comportement est complexe et il est essentiel de bien le comprendre pour implémenter efficacement IEngine. Le mieux pour en comprendre le fonctionnement est certainement de reprendre le diagramme de séquence fourni dans la documentation officielle de OpenMI [12] et présent à la figure 27.

Les instances représentées sur le diagramme sont au nombre de sept et s'expliquent comme suit :

- Model A : ILinkableComponent : Composant recepteur
- Model B : ILinkableComponent : Composant source
- Engine : IAdvancedEngine : Implémentation de IEngine du composant B
- Buffer : SmartBuffer : Buffer interne OpenMI du composant B
- elementMapper : ElementMapper : Objet du SDK traitant l'éventuelle géo-localisation
- InputLink : SmartInputLink : Implémentation du lien d'entrée du composant B
- OutputLink : SmartOutputLink : Implémentation du lien de sortie du composant A

On notera donc que seul les trois premières instances ont du être développées, les quatre autres étant fournies par le SDK.

[19]Type d'ElementSet dépourvu de coordonnées spatiales en opposition par exemple à un XY-Point ou une XYPolyline.

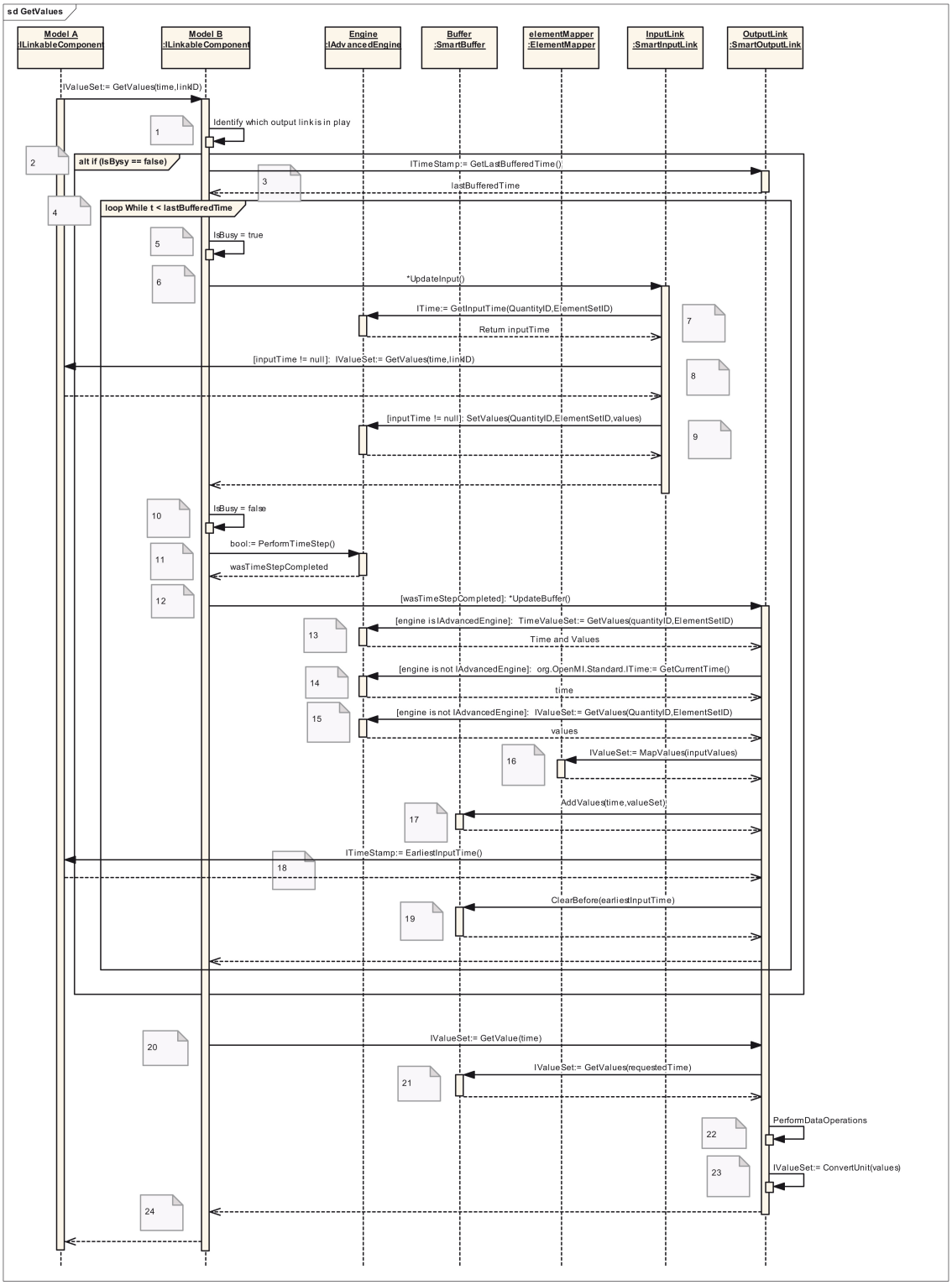

FIG. 27 – Diagramme de séquence - Méthode GetValue [12]

Reprenons alors les différentes étapes suggérées par la documentation et appliquons les dans le cadre de Tornado. La toute première étape non numérotée consiste en un appel GetValues propre au SDK, reprenant ainsi une implémentation différente que la méthode GetValues implémentée dans IEngine. La méthode

GetValues venant du SDK se charge de mettre à jour toutes les variables d'échange d'un lien pour un moment donné. En opposition, la méthode GetValues qui sera implémentée dans IEngine servira uniquement à extraire du simulateur une variable de sortie pour un moment donné.

1. Le SDK va regarder quel lien de sortie du composant B s'est vu interroger.
2. Un drapeau est mis à jour dans le composant B pour déterminer qu'il est en attente sur une valeur d'entrée. Son but est d'empêcher les blocages dans le lien bidirectionnel[20]. Si le composant B est interrogé alors qu'il est déjà en attente d'entrée, on passera directement à l'étape 20 pour procéder à l'extrapolation.
3. Le lien de sortie, étant donc identifié, est interrogé quant au dernier temps présent dans son buffer.
4. Sur cette base le composant B sera capable de déterminer s'il doit procéder à l'exécution d'un pas de temps supplémentaire. Dans le cas contraire, la variable de sortie sera disponible dans le buffer de sortie pour le temps requis et on pourra sauter à l'étape 18.
5. Pour éviter encore une fois des blocages, un nouveau drapeau indique que le composant est en phase de récupération de valeurs d'entrée.
6. Sur chacun des liens d'entrée du composant B sera exécutée la méthode UpdateInput() du SDK qui au travers de différentes méthodes implémentées dans IEngine se chargera de rafraîchir les entrées pour le temps requis.
7. La méthode GetInputTime implémentée dans IEngine du composant B va être invoquée pour chacune de ses variables d'entrée. Dans le cas où le composant n'a pas besoin de cette entrée pour exécuter le pas, la valeur ***null*** est retournée. Autrement, le temps pour lequel l'entrée est requise sera retourné. Dans le cas de Tornado, supposons que notre composant se trouve au temps x. Pour calculer le pas suivant, il a besoin des valeurs d'entrée du pas suivant. Les composants Tornado utilisant des entrées retourneront toujours comme valeur le temps courant incrémenté d'un pas.
8. Dans le cas où ***null*** n'est pas retourné, le composant B lancera des requêtes GetValue pour récupérer les valeurs demandées pour le temps retourné à l'étape précédente. Ceci aura pour effet de lancer dans un composant C le même processus que nous sommes en train de décrire.
9. Les valeurs ainsi récupérées sont alors attribuées au composant B.
10. Le drapeau du processus de récupération d'entrée est alors relâché.

[20]voir figure 12 15

11. La méthode PerformTimeStep de l'implémentation de IEngine du composant B peut alors être invoquée et ainsi on peut procéder à la simulation du modèle jusqu'au pas suivant. En cas d'erreur, la valeur ***null*** sera retournée, reconduisant le processus à l'étape 4.
12. Le pas étant calculé, la méthode UpdateBuffer du SDK va se charger de mettre à jour le buffer de sortie du composant B.
13. L'étape présente n'est pas applicable dans le cadre de l'implémentation réalisée sur Tornado. Elle correspond en réalité à ce qu'effectueront les deux étapes suivantes.
14. Le composant B est interrogé sur son temps courant.
15. Le composant B est interrogé sur ses valeurs de sorties afin d'être associé au temps récupéré par l'étape précédente dans son lien de sortie.
16. Si le lien est géo-référencé l'ElementMapper associé au lien effectuera les éventuelles manipulations spatiales requises. Aucune manipulation ne sera donc appliquée ici, Tornado utilisant des ElementSet de type IDBased.
17. Les valeurs sont ensuite placées dans le buffer associé au lien de sortie.
18. Le composant B va interroger le composant A sur sa propriété EarliestInputTime afin de savoir quel sera le premier temps requis par le composant A. Cette requête vise avant tout l'optimisation des buffers du composant B.
19. Sur base de cette réponse, le composant peut vider ses buffers des temps qui ne seront plus sollicités.
20. Les valeurs demandées au début du processus, étant présentes sur le lien de sorties, sont demandées.
21. Les valeurs sont donc demandées par le lien de sorties à son buffer associé. Les problèmes d'interpolation ou d'extrapolation seront gérés ici par le SDK.
22. Les éventuelles opérations sur les données configurées par l'utilisateur seront appliquées[21].
23. Les éventuelles conversions d'unités seront effectuées pour que les données du composant B soient directement exploitables par le composant A.
24. Le composant A reçoit finalement les données voulues.

4.4.4.2 Particularités du Reader

Certaines spécificités vont s'appliquer dans l'implémentation de IEngine pour le Reader. Ses fonctions étant réduites, certaines de ces méthodes pourront être allégées.

[21] voir section 4.5 de la page 45

En relation avec le temps, la méthode GetInputTime retournera systématiquement la valeur ***null*** étant donné que le composant n'a pas de variable d'entrée. La méthode GetEarliestNeededTime est quant à elle également inutile pour les mêmes raisons. Cependant, des vérifications de la part du SDK seront appliquées notamment au travers de cette méthode. Devant donc retourner un temps entrant dans le TimeHorizon du composant, cette méthode retournera le temps courant du composant. La méthode SetValue est pour sa part vide. Le Reader n'ayant pas d'entrée, cette méthode ne sera jamais invoquée.

La méthode PerformTimeStep sera pour sa part bien simple. En effet elle consistera en l'incrémentation du temps courant vers le prochain temps connu du buffer créé à la lecture du fichier d'entrée.

La méthode GetValue se chargera simplement de la récupération de la valeur dans le buffer de Tornado.

4.4.4.3 Particularités du Writer

Les besoins du Writer sont quelque peu en opposition avec le Reader. On n'aura pas vraiment besoin de sorties mais par contre nous aurons besoin d'entrées.

Par rapport au temps, pour effectuer le pas de temps suivant, le Writer aura toujours besoin des valeurs de ses entrées pour le temps du pas prochain. Ainsi la méthode GetInputTime retournera le temps courant du Writer incrémenté d'un pas de temps. De même, le Writer n'aura jamais besoin de temps précédant celui-là. La méthode GetEarliestNeededTime retourne donc cette même valeur.

Comme nous l'avons abordé, le Writer ne possède qu'une seule variable de sortie et ce uniquement pour pouvoir être lié avec le trigger. La méthode GetValue retournera donc un simple ScalarSet construit par défaut. On aurait pu retourner la valeur ***null***, ce qui aurait été plus performant, mais le SDK procède à une vérification du type de valeur retournée. L'utilisation de la valeur ***null*** générerait donc une exception.

Le principe du Writer est donc l'écriture de ses variables d'entrées dans un fichier et ce sans utiliser Tornado. L'implémentation de IEngine possède donc une donnée membre qui servira de buffer interne au composant. Ce buffer a été implémenté sous forme de liste de double. La méthode SetValue procédera donc à l'ajout de la valeur récupérée dans ce buffer. Ensuite, la méthode PerformTimeStep sera appelée et se chargera de rajouter une ligne au fichier de sortie après avoir incrémenté son temps courant d'un pas. Pour remplir le fichier de sortie, il inscrira en premier lieu

son nouveau temps courant et écrira ensuite chacune des valeurs du buffer sur la même ligne puis en terminant par un retour de charriot. Pour terminer la méthode PerformTimeStep, le buffer sera vidé pour laisser place aux valeurs du prochain pas de temps.

Le processus qui vient d'être développé est possible grâce à la conception du SDK. En effet, on est sûr que celui-ci procédera systématiquement en l'appel de la méthode SetValue dans le même ordre et en l'occurrence dans l'ordre alphabétique des identifiants des variables. C'est ainsi que nous pouvons avoir la certitude que les valeurs seront à chaque pas de temps écrites dans la bonne colonne du fichier de sortie.

4.4.4.4 Particularités du Exec

La gestion du temps sera identique au composant Writer et ce pour les mêmes raisons. Les méthodes GetInputTime et GetEarliestNeededTime retourneront toutes deux le temps courant incrémenté d'un pas.

La méthode PerformTimeStep sera quant à elle particulière puisqu'elle devra faire fonctionner Tornado pas à pas. A l'entrée de cette méthode sera donc incrémenté le temps courant comme pour les autres composants. Ensuite, la méthode Simulate de CExecConnector sera appelée avec comme paramètre le nouveau temps courant. Cette méthode Simulate se chargera donc de faire avancer la simulation d'un pas. Au début de cette méthode, sera donc incrémenté le temps de fin de la simulation dans le noyau Tornado. Ce dernier lancera la simulation qui se passera donc sur un intervalle commençant à l'ancien temps courant et finissant au nouveau. Après la simulation de ce pas, la valeur du temps de début de la simulation dans le noyau de Tornado sera fixée à son temps de fin, étant prêt ainsi pour un prochain pas.

De part cette façon de refaire partir la simulation, la méthode SetValue a été implémentée de manière à utiliser la méthode ExpSetInitialValue de l'API .NET de Tornado pour ajouter les nouvelles entrées requises par le modèle.

La méthode GetValue se chargera alors de récupérer les valeurs requises dans le buffer de Tornado. Pour ce faire, elle utilisera la méthode GetValue de l'API .NET de Tornado et ce s'appliquant sur le buffer de Tornado créé pour OpenMI. Comme nous l'avions développé au cours de la section 4.4.2.3 de la page 32, lors du démarrage de la simulation, le composant va devoir remplir une première fois ses buffers de sorties sans avoir l'occasion d'effectuer le processus développé dans le diagramme de séquence de la figure 27. Ainsi donc, nous avions développé le fait

de décaler le début de la plage de fonctionnement du composant. Cette décision va donc avoir une conséquence dans l'implémentation de la méthode GetValue qui en temps normal ira chercher dans le buffer de Tornado la valeur demandée. Avant de procéder au premier pas, un drapeau devra indiquer que la méthode GetValue devra retourner la valeur zéro. Au premier passage dans la méthode PerformTimeStep, ce drapeau sera relâché.

4.4.5 Phase de clôture

La phase de clôture est très simple en soi. Elle consiste uniquement en l'appel des méthodes Finish et Dispose. Dans le cadre de Tornado, seule la méthode Finish a été utilisée, étant suffisante.

Pour le Reader, elle consistera en la fermeture de l'éventuel fichier journal. Pour le Writer, elle consistera en la fermeture du fichier de sorties. Pour l'Exec, elle consistera en la fermeture de l'éventuel fichier journal ainsi qu'en la fermeture de l'expérience Tornado au travers de la méthode Close de ExecConnector.

4.5 Mise en garde sur les conversions d'unité

Dans le cadre de l'implémentation de OpenMI sur Tornado, une mise en garde concernant la conversion automatique des unités est importante. Telle que le reprenait la figure 8 de la page 13, les données échangées au sein d'OpenMI sont caractérisées notamment au travers d'une Quantity. Cette dernière reprenant l'unité et la dimension de la variable échangée.

Tout d'abord un premier problème se pose avec la dimension, Tornado ne renseigne nulle part cette information. Il est donc impossible sur base du seul nom de l'unité d'en déterminer la dimension. Le standard ne force toutefois pas les composants à fournir cette information. Théoriquement, l'information sur la dimension permettait une vérification pour empêcher de connecter une vitesse avec un volume.

Un second problème est également présent, cette fois-ci avec les unités. Celles-ci sont donc caractérisées par un facteur de translation et un coefficient permettant de les convertir vers le SI[22]. Ces deux seules caractéristiques seront utilisées par OpenMI pour vérifier si une conversion est nécessaire et pour en cas de besoin pouvoir la faire. L'API .NET de Tornado prévoit effectivement des méthodes pour

[22]Système International d'unités

obtenir ces deux facteurs. Cependant, il semble que Tornado n'utilise pas tout à fait le SI. On notera par exemple que Tornado considère le gramme comme étant la référence SI pour les unités de masse alors qu'il s'agit en réalité du kilogramme.

L'utilisateur mis en garde pourra toutefois s'en sortir aisément. En effet, le standard définit au travers de l'interface IDataOperation la possibilité à l'utilisateur de procéder à une opération quelconque sur les données avant de procéder à une éventuelle conversion d'unité. L'opération peut être d'aspect spatiale, temporelle ou tout autre. Evidemment ces opérations doivent être prévues initialement par le développeur. Dans le cadre de Tornado, aucune opération n'a été prévue en raison de la nature dynamique des modèles utilisés. Toutefois, le SDK implémente par défaut une opération disponible sur chacun des OutputExchangeItem. Il s'agit d'une conversion linéaire. Ainsi lors de la composition des liens entre les entrées et sorties des composants, l'utilisateur pourra choisir dans l'OpenMI Configuration Editor d'ajouter une conversion linéaire sur la variable de sortie. Sur base donc des facteurs affichés au niveau des unités, l'utilisateur pourra fixer de nouveau coefficient correcteur. Bien évidemment, il s'agit ici d'une solution de secours, le standard prévoyant des conversions automatiques.

5 Phase de test

5.1 Introduction

En plus d'une évaluation continue de l'ensemble du code, deux tests de situations pratiques ont été mis en place. Le premier étant la mise en place des composant Exec et Writer ensemble pour exécuter le modèle Prédateur - Proie. Le second test met en lien quant à lui le jeu complet des trois composants développés pour Tornado.

5.2 PredatorPrey

Le modèle prédateur-proie est un grand classique dans le monde de l'informatique et une expérience Tornado le mettant en pratique est fournie par défaut avec son installation. Il consiste simplement en l'évolution de la quantité de prédateurs et de proies sur base de valeurs initiales de prédateurs et de proies. Le composant Exec a donc été chargé d'exécuter le modèle et le composant Writer a pour sa part écrit les résultats dans un fichier de sorties. Pour réaliser le test, le descriptif de l'expérience est resté identique à la version fournie par défaut avec Tornado et donc avec la configuration interne d'un fichier de sorties également. La comparaison des deux fichiers de sorties a donc pu nous confirmer de part leur correspondance le bon fonctionnement des composants.

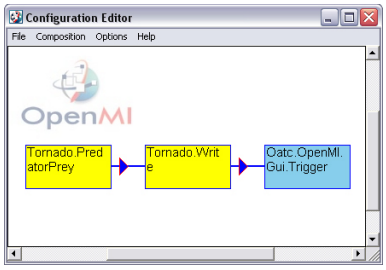

FIG. 28 – OpenMI Configuration Editor - test du modèle PredatorPrey

La figure 29 illustre les résultats produits par la simulation.

5.3 ASU

Le test ici présent a été réalisé sur l'expérience basée sur le modèle de boues activées ASU fournie également par défaut avec Tornado. L'expérience va cette

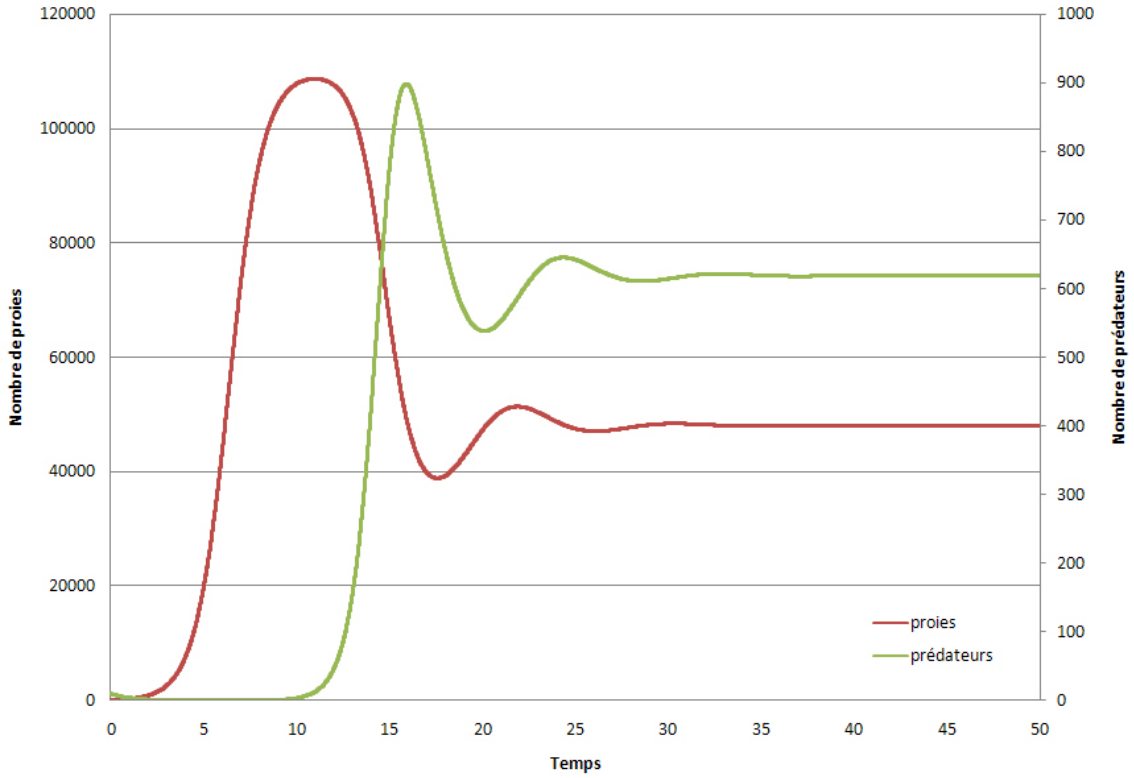

FIG. 29 – Graphique des résultats de PredatorPrey

fois-ci calculer des concentrations de sorties sur base de concentrations d'entrées. L'expérience a d'abord été simulée une première fois hors du cadre de OpenMI pour en avoir un fichier de sorties conforme.

Ensuite la composition telle que présentée à la figure 30 a été mise en place dans le OpenMI Configuration Editor. Les trois composants ont été mis en relation cette fois-ci. Le Reader se charge de récupérer les entrées pour les donner à Exec. Celui-ci va alors pouvoir calculer les concentrations de sorties à envoyer au Writer. Une fois encore, la comparaison du fichier de sortie du Writer correspondait à la version conforme.

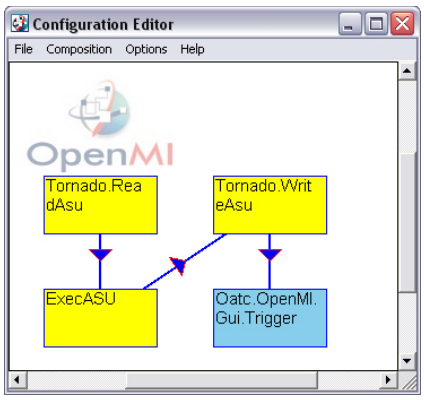

FIG. 30 – OpenMI Configuration Editor - test du modèle ASU

La figure 31 illustre une partie des résultats ASU. Seul le débit en eau et la concentration de deux composés ont été illustrés.

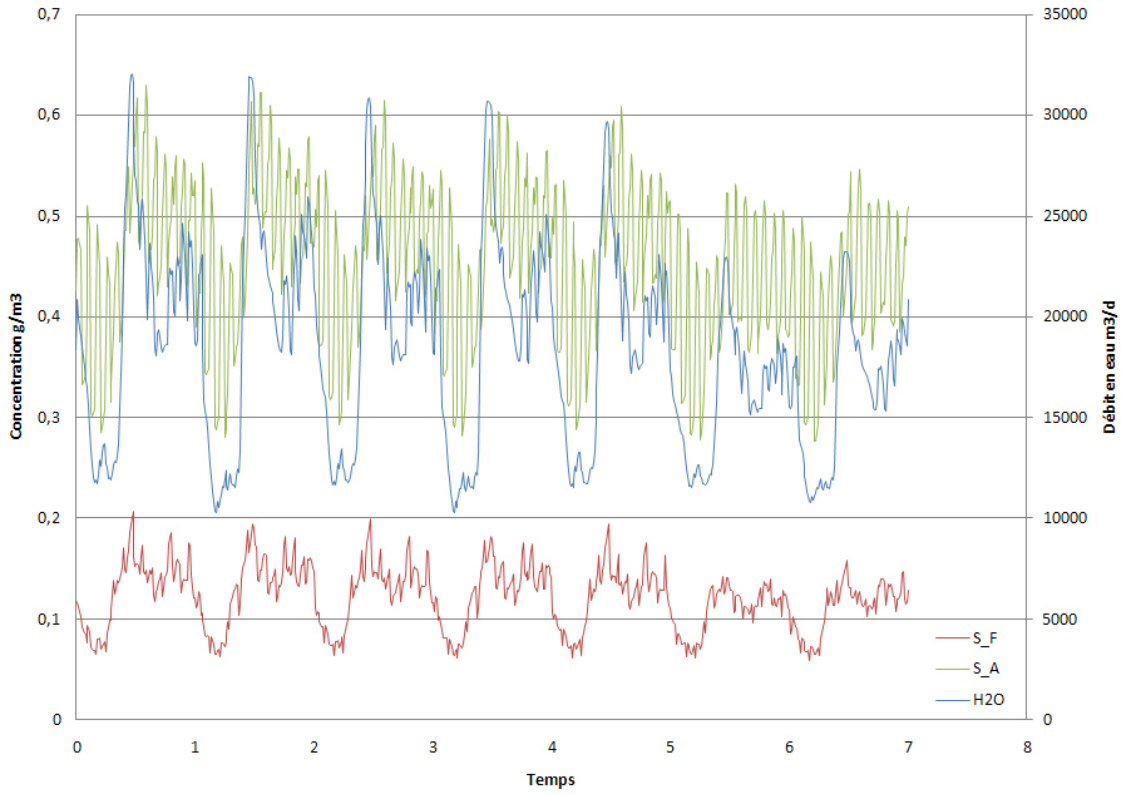

FIG. 31 – Graphique des résultats de ASU

6 Conclusion

Nous pouvons donc conclure au travers de ce travail que le standard OpenMI a bien répondu à ses consignes. Le standard développé par HarmonIT et promu par l'OpenMI Association est en effet puissant et flexible. Le standard en lui-même prend en considération l'ensemble des modèles basés sur le temps. Chacun d'entre eux y a été prévu tel que les modèles géo-référencés. La décomposition d'OpenMI en son standard et son SDK nous offrant ainsi la flexibilité. La définition du standard au travers d'interfaces nous offre en effet beaucoup de flexibilité et le SDK nous permet tout de même une bonne base de travail.

On relèvera toutefois qu'OpenMI est relativement complexe à appréhender. Son besoin de prendre en compte l'ensemble des modèles basés sur le temps lui confère une structure volumineuse qui peut sembler confuse au premier abord. Toutefois, lorsque le standard est bien pris en main, son implémentation s'en voit relativement simple.

Comme nous l'avons vu, OpenMI est le produit d'un besoin créé par la directive-cadre sur l'eau. Son implémentation devient ainsi un net avantage pour un logiciel moderne de simulation. De plus, dans son besoin de prendre en compte tous les besoins, Tornado se devait de proposer une porte vers OpenMI.

Enfin, nous pouvons constater qu'au travers des trois composants OpenMI développés, une solution vraiment complète a été offerte aux futurs utilisateurs de Tornado. Chacun des éventuels besoin a été pris en compte.

Références

[1] Actil F., Rousselle J., Lauzon N. (2005). *Hydrologie, cheminements de l'eau*, Presses Internationales Polytechniques, Montréal, Canada

[2] Filip Claeys (2008) A *generic software framework for modelling and virtual experimentation with complex environmental systems.* PhD. Thesis. Faculty of Bioscience Engineering. Ghent University, Belgium.

[3] Vanhooren H., Meirlaen J., Amerlinck Y., Claeys F., Vangheluwe H. and Vanrolleghem P.A. (2003) *WEST : Modelling biological wastewater treatment.* J. Hydroinformatics, 5, 27-50

[4] Vanrolleghem P.A., Benedetti L. and Meirlaen J. (2005) Modelling and real-time control of the integrated urban wastewater system. *Environmental Modelling and Software* 20, 427-442.

[5] CEC, 2000. Directive 2000/60/EC of the European Parliament and of the Council establishing a framework for the Community action in the field of water policy.

[6] Article Wikipédia sur l'utilisateur lambda `http://fr.wikipedia.org/wiki/Utilisateur_lambda`

[7] OpenMI Association (2007) *Scope for the OpenMI.* Part A of the OpenMI Document Series

[8] OpenMI Association (2007) *Guidelines.* Part B of the OpenMI Document Series

[9] OpenMI Association (2007) *The org.OpenMI.Standard interface specification.* Part C of the OpenMI Document Series

[10] OpenMI Association (2007) *The org.OpenMI.Backbone technical documentation.* Part D of the OpenMI Document Series

[11] OpenMI Association (2007) *The org.OpenMI.DevelopmentSupport technical documentation.* Part E of the OpenMI Document Series

[12] OpenMI Association (2007) *The org.OpenMI.Utilities technical documentation.* Part F of the OpenMI Document Series

Annexe

A Description XML de l'expérience Tornado PredatorPrey

```
<?xml version="1.0" encoding="UTF-8"?>
<Tornado>
  <Exp Version="1.0" Type="Simul">
    <Props>
      <Prop Name="Author" Value="PCFC9400\FC"/>
      <Prop Name="Date" Value="Wed Jan 28 14:12:04 2009"/>
      <Prop Name="Desc" Value="PredatorPrey simulation experiment"/>
      <Prop Name="FileName" Value="PredatorPrey.Simul.Exp.xml|.Tornado"/>
      <Prop Name="DisplayName" Value=""/>
      <Prop Name="UnitSystem" Value="Model-based"/>
    </Props>
    <Simul>
      <Model Name="PredatorPrey" CheckBounds="false" StopWhenBoundsViolation="
          false" StopWhenSteadyState="true" SteadyStateTol="0">
        <Quantities>
          <Quantity Name=".c1" Value="0.001"/>
          <Quantity Name=".c2" Value="0.9"/>
          <Quantity Name=".c3" Value="0.0001"/>
          <Quantity Name=".c4" Value="1.1"/>
          <Quantity Name=".c5" Value="1e-005"/>
          <Quantity Name=".b" Value="0.02"/>
          <Quantity Name=".pa.in_1" Value="52904.594"/>
          <Quantity Name=".pa.in_2" Value="52904.736"/>
          <Quantity Name=".pa.p" Value="100"/>
          <Quantity Name=".ps.in_1" Value="595.46729"/>
          <Quantity Name=".ps.in_2" Value="595.46946"/>
          <Quantity Name=".ps.p" Value="10"/>
          <Quantity Name=".c4pa.c" Value="1.1"/>
          <Quantity Name=".c4pa.in_1" Value="48095.085"/>
          <Quantity Name=".c5papa_plus_c1paps.in_1" Value="29773.364"/>
          <Quantity Name=".c5papa_plus_c1paps.in_2" Value="23131.372"/>
          <Quantity Name=".c5papa.c" Value="1e-005"/>
          <Quantity Name=".c5papa.in_1" Value="48095.085"/>
          <Quantity Name=".c5papa.in_2" Value="48095.085"/>
          <Quantity Name=".c1paps.c" Value="0.001"/>
          <Quantity Name=".c1paps.in_1" Value="619.05212"/>
          <Quantity Name=".c1paps.in_2" Value="48095.085"/>
          <Quantity Name=".bc1paps.c" Value="0.02"/>
          <Quantity Name=".bc1paps.in_1" Value="29773.364"/>
          <Quantity Name=".c2ps_plus_c3psps.in_1" Value="557.14691"/>
          <Quantity Name=".c2ps_plus_c3psps.in_2" Value="38.322552"/>
          <Quantity Name=".c3psps.c" Value="0.0001"/>
          <Quantity Name=".c3psps.in_1" Value="619.05212"/>
          <Quantity Name=".c3psps.in_2" Value="619.05212"/>
          <Quantity Name=".c2ps.c" Value="0.9"/>
          <Quantity Name=".c2ps.in_1" Value="619.05212"/>
        </Quantities>
      </Model>
      <Inputs Enabled="false">
        <Input Name="*Calc*">
```

```
      <CalcVars Enabled="true">
      </CalcVars>
    </Input>
  </Inputs>
  <Outputs Enabled="true">
    <Output Name="*Calc*">
      <CalcVars Enabled="true">
      </CalcVars>
    </Output>
    <Output Name="*Ext*">
      <Ext Name="" Enabled="true">
        <Props>
          <Prop Name="CommInt" Value="0.5"/>
          <Prop Name="Interpolated" Value="true"/>
          <Prop Name="UseDisplayUnits" Value="false"/>
          <Prop Name="StartTime" Value="-INF"/>
          <Prop Name="StopTime" Value="+INF"/>
        </Props>
        <Quantities>
          <Quantity Name=".pa.out_1">
            <Props>
            </Props>
          </Quantity>
          <Quantity Name=".pa.p">
            <Props>
            </Props>
          </Quantity>
          <Quantity Name=".ps.out_1">
            <Props>
            </Props>
          </Quantity>
          <Quantity Name=".ps.p">
            <Props>
            </Props>
          </Quantity>
        </Quantities>
      </Ext>
    </Output>
    <Output Name="File">
      <File Name="PredatorPrey.Simul.out.txt" Enabled="true">
        <Props>
          <Prop Name="CommInt" Value="0"/>
          <Prop Name="CommIntType" Value="Linear"/>
          <Prop Name="Interpolated" Value="true"/>
          <Prop Name="UseDisplayUnits" Value="false"/>
          <Prop Name="StartTime" Value="-INF"/>
          <Prop Name="StopTime" Value="+INF"/>
          <Prop Name="DecSep" Value="."/>
          <Prop Name="Precision" Value="8"/>
        </Props>
        <Quantities>
          <Quantity Name=".pa.out_1">
            <Props>
            </Props>
          </Quantity>
          <Quantity Name=".ps.out_1">
            <Props>
            </Props>
          </Quantity>
```

```
            </Quantities>
          </File>
        </Output>
      </Outputs>
      <Time>
        <Props>
          <Prop Name="StartTime" Value="0"/>
          <Prop Name="StopTime" Value="100"/>
        </Props>
      </Time>
      <Solve>
        <Integ Method="RK4ASC">
          <Props>
            <Prop Name="MaxNoSteps" Value="0"/>
            <Prop Name="InitialStepSize" Value="1e-005"/>
            <Prop Name="MinStepSize" Value="1e-005"/>
            <Prop Name="MaxStepSize" Value="1"/>
            <Prop Name="Accuracy" Value="1e-008"/>
            <Prop Name="CheckMinStepSize" Value="false"/>
          </Props>
        </Integ>
      </Solve>
    </Simul>
  </Exp>
</Tornado>
```

B Fichiers omi du test Predator-Prey

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.ExecLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="Tornado.PredatorPrey"/>
    <Argument Key="MainFileName" ReadOnly="true" Value="$(TORNADO_ROOT_PATH)/etc/
        Tornado.Main.xml"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogPredatorPrey.txt"/>
    <Argument Key="ExpFileName" ReadOnly="true" Value="$(TORNADO_DATA_PATH)/
        PredatorPrey/PredatorPrey.Simul.Exp.xml"/>
    <Argument Key="StepSize" ReadOnly="true" Value="0.02"/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
   </Arguments>
</LinkableComponent>
```

Listing 4 – Fichier omi du composant Exec

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.WriteLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="Tornado.Write"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogWritePredatorPrey.txt"/>
    <Argument Key="DataFileName" ReadOnly="true" Value="WritePredatorPrey.Out.txt"
        />
    <Argument Key="StepSize" ReadOnly="true" Value="0.01"/>
    <Argument Key="StopTime" ReadOnly="true" Value="100000"/>
    <Argument Key="NoOutputs" ReadOnly="true" Value="2"/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
   </Arguments>
</LinkableComponent>
```

Listing 5 – Fichier omi du composant Write

C Fichiers omi du test ASU

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.ReadLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="Tornado.ReadAsu"/>
    <Argument Key="MainFileName" ReadOnly="true" Value="$(TORNADO_ROOT_PATH)/etc/
        Tornado.Main.xml"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogReadAsu.txt"/>
    <Argument Key="DataFileName" ReadOnly="true" Value="ASU.Simul.in.txt"/>
    <Argument Key="DecSep" ReadOnly="true" Value="."/>
    <Argument Key="ThousandSep" ReadOnly="true" Value=","/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
   </Arguments>
</LinkableComponent>
```

Listing 6 – Fichier omi du composant Read

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.ExecLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="ExecASU"/>
    <Argument Key="MainFileName" ReadOnly="true" Value="$(TORNADO_ROOT_PATH)/etc/
        Tornado.Main.xml"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogExecAsu.txt"/>
    <Argument Key="ExpFileName" ReadOnly="true" Value="ASU.Simul.Exp.OMI.xml"/>
    <Argument Key="StepSize" ReadOnly="true" Value="0.01"/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
   </Arguments>
</LinkableComponent>
```

Listing 7 – Fichier omi du composant Exec

```
<?xml version="1.0"?>
<LinkableComponent Type="TornadoOMI.WriteLinkableComponent" Assembly="..\..\bin\
    Release\M4W.Tornado.OMI.dll">
  <Arguments>
    <Argument Key="ComponentID" ReadOnly="true" Value="Tornado.WriteAsu"/>
    <Argument Key="LogFileName" ReadOnly="true" Value="LogWriteAsu.txt"/>
    <Argument Key="DataFileName" ReadOnly="true" Value="WriteAsu.Out.txt"/>
    <Argument Key="StepSize" ReadOnly="true" Value="0.01"/>
    <Argument Key="StopTime" ReadOnly="true" Value="100000"/>
    <Argument Key="NoOutputs" ReadOnly="true" Value="28"/>
    <Argument Key="StartTimeFix" ReadOnly="true" Value="1/1/1900 0:00:00"/>
   </Arguments>
</LinkableComponent>
```

Listing 8 – Fichier omi du composant Write