Wrapper Java d'un code C++ avec JNI

Arouna GANOU*

2017-08-07

Table de matière

*L'auteur ne saurait remercier assez Thomas Paris, Vincent Chevrier et Julien Vaubourg

Cette page est laissée volontairement blanche.

Introduction

Dans ce document, nous voulons présenter les résultats de notre stage de recherche à l'INRIA de Nancy. L'objectif du stage est de développer un wrapper en Java pour des interfaces de modèles de simulation développées en C++. Le wrapper se basera sur une bibliothèque dynamique écrite en C++ que le code Java utilisera directement. Nous allons d' abord présenter le contexte du stage et les choix que nous avons faits. Ensuite, nous expliquerons les principes que nous avons utilisés pour faire communiquer le langage Java et C++. Nous allons terminer par vous présenter l'intégration avec Mecsyco et comment rendre notre travail réutilisable et maintenable.

1 Contexte et choix

En développement logiciel, il est habituel d'écrire des bouts de codes pour pouvoir les réutiliser après soit dans le même projet, soit dans un projet absolument différent. Nous sommes habitués à ce principe pour un seul langage. Prenons un exemple, en Java ou n'importe quel autre langage de programmation d'ailleurs, c'est vu comme une très bonne pratique d'écrire une routine une fois pour toute et utiliser cette routine dans toutes les autres parties du projet qui a besoin d'utiliser cette routine. En fait, pour des projets assez complexes, il est en fait absolument utile(voire indispensable) de procéder de la sorte. En plus, souvent, il est un choix judicieux d'étendre cette habitude au niveau des langages. Donc on écrira des fonctions dans un certain langage pour pouvoir l'utiliser dans un autre. On sait que Java est un bon choix pour faire des interfaces graphiques, par contre un mauvais choix pour faire du calcul scientifique; et Fortran n'est pas mauvais pour les calculs mais ce n'est pas une bonne idée de faire des interfaces graphiques avec Fortran par rapport aux technologies existantes. Donc, au lieu de se cloisonner dans un seul langage, on en utilisera deux pour avoir des gain de performance dans les deux aspects de notre logiciel. C'est particulièrement utile voire indispensable lorsqu'il s'agit de simulation de système qui ne sont pas forcément de même nature : On peut bien vouloir faire de la co-simulation sur des applications qui ne sont pas forcément développées les mêmes langages de programmation. Dans Mecsyco, c'est exactement le cas, on a des simulateurs dont on peut contrôler le comportement à travers des modèles artefacts qui sont en C++. La bibliothèque dynamique nous permettra d'intégrer ces simulateurs dans une simulation faite en Java.

Il sera expliqué dans cette partie notre contexte particulier et les raisons de nos choix.

1.1 Contexte

Nous avons travaillé sur le logiciel Mecsyco. C'est un langage de modélisation et de simulation de systèmes complexes. L'objectif de Mecsyco est de permettre la

simulation avec des simulateurs hétérogènes et décentralisés. Le logiciel utilise une architecture multi-agent pour intégrer les différents simulateurs dans la simulation globale. On a deux versions du logiciel: l'une en Java et l'autre en C++. Nous n'avons pas les moyens nécessaires pour maintenir les deux versions à la fois. Mais, on voudrait bien faire évoluer la version Java en utilisant quand même quelques parties du logiciel qui est en C++. Une solution est de développer une couche en Java capable d'interagir directement avec les parties du logiciel en C++ qui nous intéressent en utilisant une bibliothèque wrapper.

1.2 Choix entre les wrappers existants

Pour le langage Java, on a le choix entre plusieurs mécanisme pour développer une bibliothèque wrapper. Nous pouvons citer entre autres SWIG(Simplified Wrapper and Interface Generator [1]), JNA(Java Native Access [2]) et JNI(Java Native Interface [3]). Nous n'avons pas regardé vraiment les possibilités qu'offre SWIG parce qu'on ignorait son existence au début du stage.

Nous avons commencé avec JNA. C'est une méthode relativement facile à comprendre. Ce lien [4] pourrait vous aider à prendre en main JNA. Sous Windows OS et Mac OS, le code Java ne devrait pas changer, mais l'extension de la bibliothèque change et il faut compiler différemment la source de la bibliothèque pour chacun des OS. Sur le principe d'utilisation JNA est très simple. Mais il est surtout adapté pour les codes C et non un code C++ qui met l'accent sur l'utilisation des classe. Pour garder l'état interne en C++, il fallait déclarer une variable globale pour garder l'état de l'objet ce qui n'est pas une très bonne idée parce que la variable serait donc partagée. Mais JNA pour du code C, il semble être un très bon choix. Par exemple, si on veut faire l'appel système dans un code Java, il est relativement plus simple d'utiliser JNA comme la création de répertoire, la destruction de processus, etc. C'est déjà fait en C, donc c'est une bonne idée d'utiliser JNA pour utiliser ces fonctionnalités dans son application Java sans avoir à presque rien réécrire comme code. Ce lien vous illustre cette possibilité d'appel système pour Windows [5]. Il faut aussi ajouter que JNA ne permet pas d'interaction dans le sens code natif(code C++ ici dans notre cas ici) vers le code Java. Souvent, cette interaction peut-être utile et souvent systématique.

---

[1] `https://en.wikipedia.org/wiki/SWIG`

[2] `https://en.wikipedia.org/wiki/Java_Native_Access`

[3] `https://en.wikipedia.org/wiki/Java_Native_Interface`

[4] `http://mbaron.developpez.com/tutoriels/java/executer-code-natif-jna-5-minutes`
Mais il vous demandera de créer un projet Maven. si vous êtes sur eclipse, il se peut que vous rencontrer cette exception:
Exception in thread "main"java.lang.UnsatisfiedLinkError:
Si c'est le cas, il serait une bonne idée de penser à mettre la bibliothèque qui sera dont le nom sera du type *lib***nomLibrary***.so* dans le repertoire *src/main/resources/linux-x86-64* et **nomLibrary** est la chaîne de caractère que vous passer à la fonction **Native.loadLibrary**. Ceci a été utile sous Ubuntu 16.04.

[5] `https://en.wikipedia.org/wiki/Java_Native_Access`

Nous avons choisi JNI par rapport à JNA. La première raison de notre choix c'est la gestion intrinsèque des objets C++ par cette approche. Quand nous étions en JNA, la gestion des objets a posé un problème épineux pour garder leurs états internes. Mais il est plus difficile d'utiliser JNI que JNA. JNI demande assez de temps pour pouvoir comprendre son principe. Dans la section suivante, nous allons parler du principe que nous avons suivi. Ce principe est inspiré d'une approche classique lorsqu'on utilise JNI. Mais nous allons d'abord expliquer le mécanisme de conversion des types primitifs et des strings avant de détailler dans la section suivante notre stratégie globale. Pour les types primitifs, il y a une conversion explicite qui est en général simple et ne demande pas de précautions particulières. Par exemple une variable de type **long** sera transmise en C++ en une variable de type **jlong** et les **jlong** est directement convertie en **long** C++. C'est la même technique pour les **boolean**, **char**, **byte**, **int** , **float** et **double**. Par contre pour les types par référence, on peut les convertir en C++ mais il faut prendre assez de précautions pour leurs conversions. En fait il y a deux possibilités offertes par JNI. On peut bien faire un appel de C++ vers Java sur l'objet passer en paramètres. Ce n'est pas une conversion en tant que telle mais on utilise l'objet directement dans le code natif (C++ en l'occurrence) La seconde possibilité est de convertir vraiment l'objet. C'est le cas des **String** Java qui seront justement des **jstring** en C++, dont la conversion en **string** C++ n'est pas naturelles par rapport à un code Java et qui surtout demande des précautions parce qu'il est nécessaire de libérer les resources après la conversion. Nous allons donner une illustration de cette conversion dans la section suivante.

2 Principe de notre travail

L'utilisation de JNI demande d'être assez méthodique. Dans cette section nous allons expliquer comment à partir des deux codes existants l'un en Java et l'autre en C++ avoir une communication entre les deux codes en utilisant une bibliothèque dynamique.

2.1 Préparation et adaptation des codes

Il faut effectivement dès le début savoir ce qu'on veut vraiment implémenter dans le code natif(C++ ici). Dans le cadre de notre stage, on voulait utiliser en Java des instances d'une classe C++ existante. Donc nous allons nous placer dans ce cadre pour définir notre principe. Mais le principe reste valable pour plusieurs classes en interactions en Java avec un autre groupe de classes en C++.
Nous avons au début une interface **ModelArtifact** en Java et une autre en C++ **ModelArtifact.h**. Le but précis ici, c'est de faire en sorte que toute implémentation de cette **ModelArtifact.h** puisse être appelée par notre implémentation de la bibliothèque à travers une classe **CppModelArtifact** qui est en Java. On aimerait bien pouvoir appeler toutes les fonctions qui sont dans cette interface(toutes les classes filles de l'interface ayant ces fonctions par

principe). Nous verrons que c'est une difficulté en soi.

2.2 Déclaration des appels natifs

En Java, on définit d'abord notre classe **CppModelArtifact** en héritant de l'interface **GenericModelArtifact**. En principe, on n'a pas besoin d'ajouter fonctions particulières. Mais dans notre cas, il a été nécessaire d'ajouter une fonction qui permet de sérialiser et déserialiser. Notons que cette fonction est particulière au fait que nous utilisons des types de données particuliers qui étaient définis dans Mecsyco: les tuples. On a les signatures des fonctions héritées. Mais en fait, nous avons trouvé judicieux de faire passer entre Java et C++ que les Strings comme type par référence. L'idée c'est de réduire la complexité du travail donc nous n'allons ni envoyer, ni recevoir des objets de types autre que des Strings. En plus de ces types, on envoie aussi les tableaux de Strings. Mais ce n'est pas particulièrement plus embêtant que les Strings eux-mêmes et il est très utile de retourner un tableau de string quand il s'agit d'un objet avec plus de deux attributs pour reconstruire l'objet de retour en Java. Comme nous gardons la signature de l'interface dont on hérite, il faut faire un traitement à l'intérieur de la fonction Java, pour aboutir à des types primitifs ou des strings, il faudra en faire également pour les retours, on reconstruit l'objet(au cas où ce n'est pas un type primitif) avant de le retourner. Nous avons utilisé une technologie déjà utilisée dans Mecsyco pour sérialiser et déserialiser : JSON(JavaScript Object Notation). On utilise ce mécanisme dans le code natif pour sérialiser et déserialiser en utilisant la bibliothèque **boost**. Prenons une classe simple **Compteur**(page suivante) pour illustrer ce que nous venons de dire: Nous avons choisi d'être le plus explicite possible: Le nom des fonctions natives commencent par ***n_*** et le reste du nom de la fonction native est le nom de la fonction Java qui l'utilisera.

```java
/**
 * This class is used illustrate JNI using
 * @author GANOU Arouna
 */
public class Compter {
        //pointer value of C++ object
        private long nativeObject;
        /**
         *  Constructor
         */
        public Compter() {
                // call the native constructor
                nativeObject=n_Compteur();
        }
        //start getValue function
        public double getValue() {
                //call to native function
                return n_getValue();
        } // end getValue
        //start increment function
        public void increment(double incValue) {
        //native call to native function
                n_increment(incValue);
        }// end increment function
        //Native function are declared private bellow
        //The native construction of the object.
        private native long n_Compteur();
        //for getValue function
        private native double n_getValue();
        //for increment function
        private native void n_increment(double
            incValue);
        /**
         * It's a good idea to load the shared library
         * here.
         */
        static {
                System.loadLibrary("compter");
        }
}
```

2.2.1 Génération du fichier en-tête

Nous avons fini le travail en Java. Mais nous avons une classe qui utilisera des méthode natives. Donc il reste à implémenter ces méthodes natives. Pour commencer cette implémentation, il faut générer un fichier en-tête qui dépend

directement du nom complet de la classe, des noms de fonctions natives et de leurs signatures. Il y a deux possibilités pour générer ce fichier en-tête. On peut le faire directement en ligne de commande [6]. Cette méthode peut être automatisé en utilisant un makefile pour du code Java. La seconde possibilité que nous avons surtout utilisée est la configuration d' éclipse pour le faire automatiquement. Sous eclipse, la configuration se fait une fois pour toute et il reste à choisir une classe quelconque et lui appliquer la commande configurée. Nous expliquerons par la suite comment faire cette configuration.
1– **Localisation du javah** : Localiser le fichier javah sur votre machine en utilisant la commande *locate* comme dans l' exemple ci-dessous.

```
  > locate bin/javah
/usr/bin/javah
/usr/lib/jvm/java-8-openjdk-amd64/bin/javah
  >
```

C'est une bonne idée de prendre celui qui est sous *jvm*, les autres étant des raccourcis vers lui. Dans eclipse, aller dans la barre des taches et ensuite **run**> **External Tools**> **External Tools Configuration**, vous aurez la fenêtre de la figure ci-dessous. Ensuite, à gauche, **Program**, **clique gauche** et **New**.

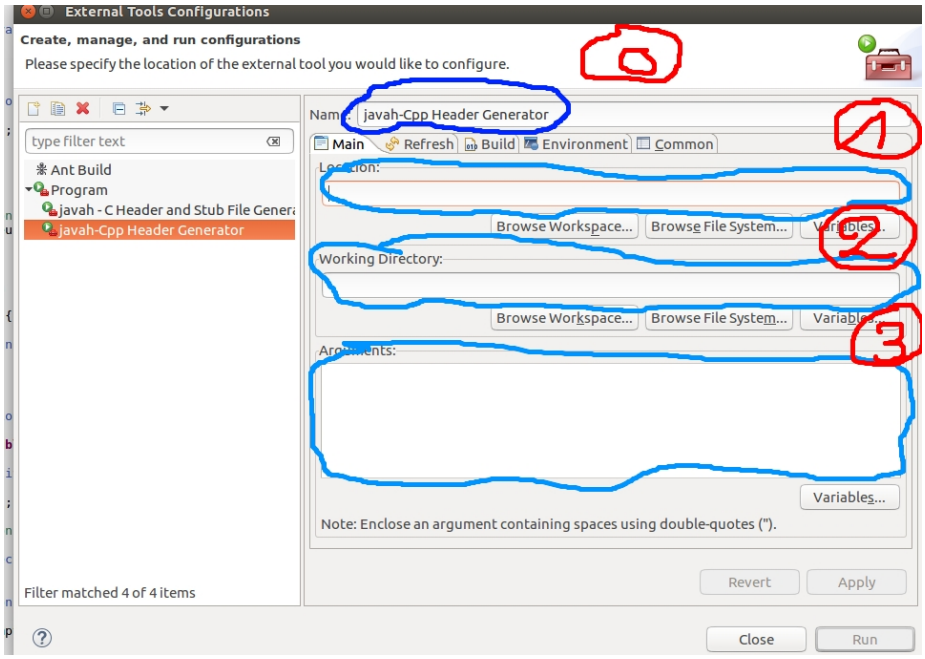

Figure 1: Configuration pour générer l'en-tête.

2– **Configuration** : Dans la figure 1, on peut donner un nom à l'outil que

[6] `https://www.jmdoudoux.fr/java/dej/chap-jni.htm`

nous sommes entrain de configurer dans **Name** 0, ensuite il faut copier le chemin complet obtenu avec la commande *locate* précédemment vers **javah** dans **Location** 1. On donne **javah** comme **Name** car la configuration que nous allons faire utilisable pour tous nos projets actuels sous eclipse. Il faut par la suite placer dans **Working Directory** 2*${workspace_ loc:/${project_ name}/bin}*. On met enfin *-classpath $ {project_ classpath} -v -d ${workspace_ loc:/$ {project_ name}/src/headersCpp} $ {java_ type_ name}* dans **Arguments** 3. Mais il faut créer un dossier **headersCpp** dans le répertoire **src**. C'est possible de mettre le fichier généré dans n'importe quel autre dossier en remplaçant *${workspace_ loc:/$ {project_ name}/src/headersCpp}* par *chemin_ de_ ce_ dossier*. On fait enfin **Apply** et **close**. On devrait avoir à la fin une image de la figure 2 Cette configuration pourra être utilisé sur toutes les classes(même sans fonc-

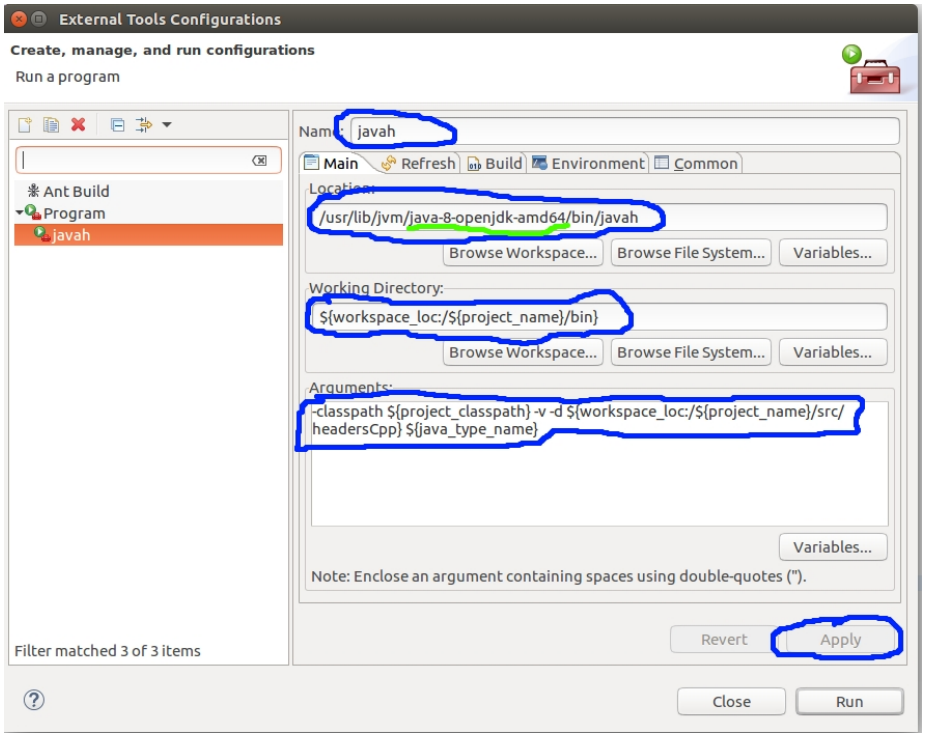

Figure 2: Commande javah configurée.

tions natives) sauf que le fichier en-tête généré n'aura aucune fonction native. Choisissez une classe, allez dans la barre d'outils et ensuite **run>External Tools>javah**. C'est bien le nom que nous avons donné pendant la configuration. Si tout se passe bien, on aura un message similaire à celui là :

```
[Creating file RegularFileObject
[/home/arek/javaWorkspace/JNI_Simple3/src/headersCpp/testjni_Compter.h]]
```

Nous allons commenter le fichier qui a été généré avant d'aller le reste du travail en C++. En principe, le fichier genéré aurait le même contenu que celui là à l'exception de *testjni* qui dépendra du nom complet de la classe qui dépend effectivement du package. Ici la classe se trouve dans le package *testjni*. Ce fichier ne doit pas être modifié comme c'est indiqué dans le fichier. Les types de retour sont entre les mots *JNIEXPORT* et *JNICALL*.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class testjni_Compter */

#ifndef _Included_testjni_Compter
#define _Included_testjni_Compter
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     testjni_Compter
 * Method:    n_Compteur
 * Signature: ()J
 */
JNIEXPORT jlong JNICALL
   Java_testjni_Compter_n_1Compteur
  (JNIEnv *, jobject);

/*
 * Class:     testjni_Compter
 * Method:    n_getValue
 * Signature: ()D
 */
JNIEXPORT jdouble JNICALL
   Java_testjni_Compter_n_1getValue
  (JNIEnv *, jobject);

/*
 * Class:     testjni_Compter
 * Method:    n_increment
 * Signature: (D)V
 */
JNIEXPORT void JNICALL
   Java_testjni_Compter_n_1increment
  (JNIEnv *, jobject, jdouble);

#ifdef __cplusplus
}
#endif
```

```
#endif
```

Tout au début il y a un **include** de *jni.h*. C'est dans ce fichier source que sont definis les types des deux premiers paramètres des fonctions: ***JNIEnv*** et ***jobject***. Notons que n'importe quelle fonction qui apparaît dans ce fichier doit avoir au moins les deux types en paramètres.
JNIEnv : C'est un lien vers la fonction *main* qui appelle la fonction dans la bibliothèque. Donc c'est l'environnement appelant la fonction.
jobject : C'est l'objet java qui appel la méthode dans la bibliothèque.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class testjni_test_Compter */

#ifndef _Included_testjni_test_Compter
#define _Included_testjni_test_Compter
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     testjni_test_Compter
 * Method:    n_Compteur
 * Signature: ()J
 */
JNIEXPORT jlong JNICALL
   Java_testjni_test_Compter_n_1Compteur
  (JNIEnv *, jclass);

/*
 * Class:     testjni_test_Compter
 * Method:    n_getValue
 * Signature: (J)D
 */
JNIEXPORT jdouble JNICALL
   Java_testjni_test_Compter_n_1getValue
  (JNIEnv *, jclass, jlong);

/*
 * Class:     testjni_test_Compter
 * Method:    n_increment
 * Signature: (JD)V
 */
JNIEXPORT void JNICALL
   Java_testjni_test_Compter_n_1increment
  (JNIEnv *, jclass, jlong, jdouble);

#ifdef __cplusplus
```

```
}
#endif
#endif
```

Il se peut qu'au lieu de ***jobject***, on ait ***jclass***, c'est le cas lorsqu'on declare les méthodes *static native*. Mais dans ce cas, il y a un troisième argument nécessaire qui est la valeur du pointeur de l'objet C++ sur lequel on interagit pour les fonctions autres que le constructeur. Cette manière de faire les choses est un raccourcis qui nous permet d'avoir directement la valeur du pointeur sans avoir le rechercher nous même en C++.

2.3 Préparation du code C++ et implémentation des appels natifs

On suppose que nous avons déjà une classe C++ existante [7] qui s'appelle Compteur dont l'entête de la classe est le code ci-dessous

```
// Compter.h //
#ifndef COMPTER_H_
#define COMPTER_H_
class Compter {
private:
  double _val; // la valeur qui nous permet de garder
     l'etat interne
public:
  // Constructeur
  Compter();
  // Observer
  double getValue();
  // Increment function
  void increment(double incValue);
};
#endif
```

Pour faire le lien entre l'objet Java et l'objet C++, on ajoute deux attributs supplémentaires à la classe. En fait, ce sont ces attributs qui permettent de garder l'état de l'objet C++ dans la pile du code Java. Donc l'idée simple est d'hériter de la classe C++ existante et ajouter ces deux attributs supplémentaires et ajouter ou ajuster les constructeurs au besoin. C'est exactement ce que nous ferons comme on le voit sur le code de la page suivante.
Une fois la classe C++ ajustée pour faire un appel depuis Java, on implémente l'en-tête qui a été généré avec eclipse dans la section précédente. C'est exactement le même entête que nous utilisons sans rien modifier à l'intérieur, ni les noms des fonctions. Les noms de fonctions contiennent en effet le nom complet

[7] Le projet C++ dont les codes sont cités ici peut être obtenu dans repositoire suivant : `https://github.com/ggas18/compteur`

de la classe Java qui l'utilisera. Une modification des noms peut empêcher la classe Java de se retrouver dans l'appel des fonctions de la bibliothèque.

```
// "JCompteur.hpp"
#include <jni.h> // pour JNIenv et jobject
#include "Compter.h" // pour la classe compteur qu'on
   veut heriter
class JCompter : public Compter {
private:
  JNIEnv *_env;// le programme en cours d'execution
     appelant
  jobject _obj;

public:
  JCompter(JNIEnv *env, jobject obj) : Compter() {
    // instancier l'attribut env
    this->_env = env;
    // creer une reference globale vers l'objet pour
       qu'on puisse l'acceder
    // apres
    this->_obj = env->NewGlobalRef(obj);
  }
};
```

Dans le code, on note qu'on met l'en-tête *jni.h*. Mais pouvoir l'utiliser comme tel dans ce fichier, il est intéressant de créer des raccourcis vers les fichiers *jni.h* et *jni_ md.h*(ce fichier dépend de la machine). C'est que ce qu'on a fait. On utilise la commande *locate* pour retrouver les chemins des fichiers et ensuite créer des liens symboliques vers eux dans le répertoire */usr/include/*avec la commande *ln* avec l'option *-s*

```
ln -s chemin_complet_de_{jni.h ou jni_md.h}
   /usr/include/{jni.h ou jni_md.h\}
```

Il est également possible de localiser le chemin complet de *jni.h* et *jni_ md.h* et mettre ces chemins complets complet dans le code. Cette option n'est pas très commande car votre code dépendra de votre machine alors que la première option est justement la méthode habituelle d'insertion des en-têtes. On pourra également utiliser le code actuel sur la machine aucune modification concernant les l'insertion de l'en-tête *jni.h* Voici l'implémentation de l'en-tête générée avec eclipse.

```
// implementation de l'en-tete generee avec eclipse.
#include "testjni_Compter.h" // c'est l'en-tete dont
   nous essayons d'implementer les fonctions

#include "JCompter.hpp" // Pour la classe JCompter,
   c'est l'objet que nous manipulons depuis java
```

```
JNIEXPORT jlong JNICALL
   Java_testjni_Compter_n_1Compter(JNIEnv
   *env,jobject obj) {
  jlong jcompPtrVal = 0;
  JCompter *jcompPtr = new JCompter(env, obj); // on
     cree l'objet
  // on caste la valeur du pointeur sur  l'objet C++
  jcompPtrVal = reinterpret_cast<jlong>(jcompPtr);
  return jcompPtrVal;}

JNIEXPORT jdouble JNICALL
   Java_testjni_Compter_n_1getValue(JNIEnv
   *env,jobject obj) {
  double ret_val = 0.0;
  // prendre la classe
  jclass cls=env->GetObjectClass(obj);
  // prendre l'identifiant du champ de l'attribut
     NativeObject
  jfieldID id=env->GetFieldID(cls,
     "nativeObject","J");
  // prendre la valeur de l'attribut NativeObject,
     donc la valeur du pointeur
  jlong pointerVal=env->GetLongField(obj, id);
  JCompter
     *jCompt=reinterpret_cast<JCompter*>(pointerVal);
  return jCompt->getValue();}

JNIEXPORT void JNICALL
   Java_testjni_Compter_n_1increment(JNIEnv
   *env,jobject obj,jdouble incVal) {
  jclass cls=env->GetObjectClass(obj); // prendre la
     classe
  // prendre l'identifiant du champ de l'attribut
     NativeObject
  jfieldID id=env->GetFieldID(cls,
     "nativeObject","J");
  // prendre la valeur de l'attribut NativeObject,
     donc la valeur du pointeur
  jlong pointerVal=env->GetLongField(obj, id);
  JCompter *jCompt=reinterpret_cast<JCompter
     *>(pointerVal);
  jCompt->increment(incVal);}
```

Nous allons commenter ce code en distinguant deux types de fonction.
Les fonctions de création : C'est typiquement les fonctions que les construc-

teurs en Java utilisent. Ces fonctions doivent retourner la valeur en entier long de l'objet C++ créé. Ici, on a en l'occurrence *Java_ testjni_ Compter_ n_ 1Compter*: On crée un pointer sur l'objet C++, ensuite on caste en entier long que l'on retourne.
Les fonctions d'instance : L'utilisation des fonctions d'instance suppose que l'objet est déjà créé. La routine d'appel est vraiment la même chose pour ces fonctions. On cherche d'abord la valeur du pointeur et ensuite on caste le pointeur en l'objet. Pour trouver la valeur du pointeur, on doit avoir d'abord la classe de l'objet avec la fonction **GetObjectClass**(**jobject**) qui est une méthode d'instance de JNIenv. Notons que cet appel est un appel en sens inverse c'est-à-dire de C++ vers Java. Avec la classe trouvée, on cherche l'identifiant **jfieldID** de l'attribut stocké en Java comme valeur du pointeur que l'on a affecté avec l'appel d'une fonction de création avec la méthode **GetFieldID**(**classe, "nom_de_latribut_java", "type"**) *classe* c'est le retour de la fonction **GetObjectClass**(**jobject**), *"type"* c'est toujours *"J"* pour **jlong**. Mais pour le nom de le **"nom_de_lattribut_java"**, il faut que le nom soit exactement la même chose et c'est sensible à la casse. Sinon l'application peut cracher totalement, voici un exemple de crash en utilisant un mauvais nom de l'attribut Java en C++.

```
#
# A fatal error has been detected by the Java Runtime
   Environment:
#
#  SIGSEGV (0xb) at pc=0x00007f983f419c84, pid=11578,
   tid=0x00007f98711d2700
#
# JRE version: OpenJDK Runtime Environment
   (8.0_131-b11) (build
   1.8.0_131-8u131-b11-2ubuntu1.16.04.3-b11)
# Java VM: OpenJDK 64-Bit Server VM (25.131-b11 mixed
   mode linux-amd64 compressed oops)
# Problematic frame:
# C  [libcompteur.so+0xc84]  Compter::getValue()+0xc
#
# Failed to write core dump. Core dumps have been
   disabled. To enable core dumping, try "ulimit -c
   unlimited" before starting Java again
#
# An error report file with more information is saved
   as:
#
   /home/arek/javaWorkspace/JNI_Simple3/hs_err_pid11578.log
#
# If you would like to submit a bug report, please
   visit:
```

```
#   http://bugreport.java.com/bugreport/crash.jsp
# The crash happened outside the Java Virtual Machine
   in native code.
# See problematic frame for where to report the bug.
#
```

On a la valeur du pointeur avec la fonction **GetLongField**(**object, jfieldID**) et **jfieldID** etant l'identifiant trouvé précédemment. On a l'objet C++ en faisant un cast avec la fonction *reinterpret_ cast*[8]. Dès qu'on a l'objet en C++, on fait tous les appels définis en C++ pour cet objet. Notons que ces trois appels du code C++ vers Java est vraiment systématique au début de chaque fonction. On peut également appeler des méthodes d'objet créer en Java. Par exemple, si on a une table de hachage rempli en Java, on peut effectivement accéder au valeur directement en C++ avec une approche similaire sur la méthode. Dans ce cas, la méthode sera traitée comme l'attribut ici, mais il y a des éléments supplémentaires comme la signature de la méthode Java. Jusqu'ici nous n'avons traité que les types primitifs qui n'exigent pas d'attention particulière pour leurs conversions. Nous allons analyser maintenant les chaînes de caractères.
Pour une chaîne de caractère **String** en signature, dans l'en-tête générateur la fonction aura un **jstring**. Cette procédure du code ci-dessous permet transformer une chaine **jstring** en **string** C++(sachant que de la chaîne **string** C++ on peut avoir les chaînes C).

```
// On suppose qu'on a la chaine jchaine qui
   est un jstring
const char *chaineC; // une chaine de
   caractere type C
chaineC=env->GetStringUTFChars(jchaine, NULL);
string  chaineCpp(chaineC) ;// c'est un appel
   C++ d'un          constructeur de la
   classe string de C++
env->ReleaseStringUTFChars(jchaine,chaineC);
   // C'est important cet appel pour liberer
   les resources.
```

La conversion des **string** C++ en **jstring** se fait facilement: On suppose qu'on a une chaine **string** C++,

```
char *chaineC1=new char[chaineCpp.length()+1];
std::strcpy(chaineC1,chaineCpp.c_str()); //
   [string].c_str() retourne une chaine C du
   type: const char *
jstring jchaine=env->NewStringUTF(chaineC1);
delete chaineC1; // pour liberer l'espace
   allouee precedemment.
```

[8] http://en.cppreference.com/w/cpp/language/reinterpret_cast

Il a été également utile de faire la conversion de tableau de **string** C++ en tableau de chaîne **jstring**. Notons que les tableaux de **jstring** sont des tableaux de types génériques **jobjectArray**. Le bout de code suivant permet justement de faire la conversion d'un tableau de chaînes **string** C++ en tableau de chaînes **jstring**. On suppose qu'on a un tableau de **string** C++ cppStringArray

```
        // traitement de la position 0
        //On cree un object jobjectArray qui
           contiendra des types String
    //donc il faut lui preciser la classe:
       "java/lang/String"
        jobjectArray jobArray=
           env->NewObjectArray(length,env->FindClass("java/lang/String"),
           0);
        //On cree ensuite le jstring
    char *chaineC1=new
       char[cppStringArray[0].length()+1];
    std::strcpy(chaineC1,cppStringArray[0].c_str());

    jstring jstr3=env->NewStringUTF(chaineC1);
    env->SetObjectArrayElement(jobArray,0,jstr3);//
       on place l'objet jstring a la place 0
    env->DeleteLocalRef (jstr3);// on libere les
       resources pour le jstring
        // on refait exactement la meme chose pour la
           position 1 du tableau
    char *chars4=new
       char[cppStringArray[1].length()+1];
    std::strcpy(chars4,cppStringArray[1].c_str());

    jstring jstr4=env->NewStringUTF(chars4);
    env->SetObjectArrayElement(jobArray,1,jstr4);
    env->DeleteLocalRef (jstr4);
    // liberation de la memoire allouee pour les deux
       chaines.
    delete[] chars3;
        delete[] chars4;
```

Le principe qu'on vient d'expliquer est une approche assez générale. Dans Mecsyco, on utilisait des types de données particuliers comme les SimulData et les SimulEvent. Les SimulData sont des chaînes de caractères qu'on sérialise et déserialise. Donc la difficulté de manipulation des objets de ces type n'est pas liée à JNI mais plutôt à la technique de déserialisation qu'on utilise. Par contre pour les SimulEvents, on a procédé différemment. Comme le SimulEvent est constitué d'un SimulData et d'un double, pour les fonctions ayant des SimulEvent en paramètre, leurs fonctions natives ne prendra pas le SimulEvent directement mais le SimulData(qui est un String) et un double, donc le problème est résolu pour

les paramètres des fonctions. Pour les retours de fonctions, on est contraint du fait qu'une fonction ne retourne qu'une seule valeur. Nous avons contourné le problème en retournant un tableau. Mais dans un tableau il faut que les éléments soient du même type. Comme le SimulData est un String, nous convertissons le double en String et on se retrouve avec un tableau de chaîne de caractères. Dès qu'on a un tableau de ce genre, on peut facilement reconstruire l'objet en gérant les exceptions qui vont avec. Une chose importante quand on travaille avec des doubles qu'on transforme des fois en chaîne de caractères ou dans le sens inverse, il est utile de configurer les paramètres globaux du programme. Si on ne le fait pas, le séparateur des réels est une virgule et ça peut poser des problèmes lors de la déserialisation. On peut le faire avec la classe suivante avec une déclaration de variable globale de la classe dans l'implémentation de l' entête généré.

```
/**
 * classe qui permet de configurer les parametres
    globaux. On a
 * un seul parametre globale a configurer: Les
    *separateurs des decimaux doit etre des points.
 */
class ConfSystem {
public:
  ConfSystem(string params) {
     std::locale::global(std::locale(params)); }
}; // Fin de la classe ConfSystem

/**
 * configurer le separateur des doubles en "."; la
    configuration sera
 * independante
 * de la localite
 */

string params("en_US.UTF8");
ConfSystem configurationDuSystem(params);
```

3 Intégration avec Mecsyco et résultats

3.1 Point de départ

Pour implémenter ce wrapper sur le modèle artefact en C++, il faut ajouter les jars suivants au projet Java :

```
libs
├── mecsyco_re_201.jar
└── jackson-annotations-2.6.1.jar
```

```
├── jackson-core-2.2.3.jar
└── jackson-databind-2.1.4.jar
```

L'archive *mecsyco_re_201.jar* pour les types de données propres à Mecsyco que nous avons utilisés et *jackson-*.jar* pour la sérialisation et la déserialisation.

```
// utile pour la serialisation/deserialisation
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import com.fasterxml.jackson.databind.type.*;

// utile pour les types definis dans mecscyco
import mecsyco.core.type.*;
```

Ce n'est pas utile de tout importer de cette manière. D'ailleurs il n'a pas été fait de la sorte dans le code réel. On l'a fait ici juste pour la présentation dans le texte. Ce modèle artifact marchera en principe pour tous les modèles artifact C++ qui échange des Tuple1 de double. Donc ce wrapper Java peut être utilisé pour les simulations de chacun des coordonnées du système de Lorenz et c'est le cas aussi pour le modèle de Voltera. Mais pour l'adapter à un échange d'un autre type, il faudra refaire les étapes de sérialisation et déserialisation. Il faut un travail similaire en C++ dans ce cas. Dans le projet C++, il est nécessaire d'avoir les dossiers suivants qui sont déjà dans la version C++ de mecsyco.

```
core
├── exception
├── model
├── serialization
└── type
```

On utilisera pas forcément tous les fichiers de ces répertoires. D'ailleurs, pendant le stage nous avons copié les fichiers qui nous sont nécessaire pour l'intégrer dans notre projet C++. Il est également utile d'avoir **boost** sur son ordinateur. L'ajout des dépendances dynamique à la bibliothèque est géré directement par **cmake**.

3.2 Adaptation au simulateur

Supposons que nous avons un simulateur en C++. Il nous reste à ajuster le modèle artefact qui lui est lié pour pouvoir le contrôler depuis Java. Nous avons utiliser un décorateur comme le modèle artefact est juste une interface et on voulait faire un code le plus générique qu'on puisse. Lorsqu'on voudra utiliser ce modèle artefact, il suffit de le faire savoir dans le décorateur qui est la classe JModelArtifact. Il s'agit précisément de cette partie du code qu'il faut modifier:

```
JModelArtifact::JModelArtifact(JNIEnv *env, jobject
    jObject){

    this->jObject=env->NewGlobalRef(jObject);
    this->env=env;
    // Il faudra modifier la ligne suivante en
```

```
    //fonction du modele artifact qu'on veut
    //utiliser.
    this->modelArtifact= new
        ModelArtifactSimulateurX();

}
```

Pour des échanges de doubles, on n'aura plus besoin de modifier l'implémentation de l'entête généré sous eclipse pour les méthodes natives. Pour contourner ce problème, il serait une bonne idée de faire une classe Factory en C++.

3.3 Compilation et assemblage final

Avec les techniques de la section précédente, on a un code source qu'il faudra compiler. Il est indispensable de lier la bibliothèque **boost** pour pouvoir compiler. Voici le *CMakeLists.txt*:

```
#you can change the minimum version of cmake below
cmake_minimum_required(VERSION 3.5.1)

project (mecsycoWrapper )

set(PROJECT_VERSION "1.0.0")

set(CMAKE_CXX_STANDARD 11)

set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++1y -pedantic -Wall
-Wextra -Wcast-align -Wcast-qual -Wctor-dtor-privacy -Wdisabled-optimization
-Wformat=2 -Winit-self -Wlogical-op -Wmissing-declarations -Wmissing-include-dirs
-Wnoexcept    -Woverloaded-virtual -Wredundant-decls -Wsign-conversion
-Wstrict-null-sentinel -Wstrict-overflow=5 -Wswitch-default -Wundef -Werror -Wno-unused")

file(GLOB SOURCES "src/*.cpp")

# for boost
find_package(Boost)
IF (Boost_FOUND)
include_directories(${Boost_INCLUDE_DIR})
endif()

set (Boost_USE_STATIC_LIBS OFF) # linking boost dynamically

find_package (Boost COMPONENTS REQUIRED system thread)


#you can change the target name, by you have to set the same name in add_library(...) and
#target_link_libraries(...)
```

```
add_library(lorenzZcpp SHARED ${SOURCES})

target_link_libraries (lorenzZcpp ${Boost_LIBRARIES})
```

Voici la structure type de l'arborescence du projet pour pouvoir utiliser correctement le fichier CMakeLists.txt [9] donné:

```
mecsycoLib
├── src
│   ├── code_source1.cpp
│   ├── code_source1.h
│   ├── code_source2.cpp
│   └── code_source2.h
└── CMakeLists.txt
```

Tout le code sources doit se trouver dans le repertoire *src*. Il suffit de creer un dossier *build*, entrer dans le repertoire *mecsycoLib* et compiler.

```
mecsycoLib> mkdir build
mecsycoLib> cd build
mecsycoLib/build> cmake .. && make
```

Vous aurez dans le répertoire *build* la bibliothèque dynamique avec le nom **mecsyco** et c'est le nom qu'il faut utiliser pendant le chargement de la bibliothèque dynamique, car la JVM se chargera de mettre les préfixes et suffixes standard pour avoir le nom complet **lib***mecsyco.so*. On doit maintenant configurer notre projet sous eclipse pour pouvoir utiliser cette bibliothèque dynamique. Dans un projet Maven, il suffit de copier le fichier *build/libmescyco.so* et le mettre dans le répertoire *src/main/resources/linux-x86-64*. Dans un projet Java, pour configurer le chemin dans la bibliothèque native, il faut aller dans les propriétés du projet, ensuite dans **Java Build Path** et sous **JRE System Library** et modifiez **Native Library Location** en précisant le chemin complet du dossier qui contient la bibliothèque native.
Une autre façon plus flexible de lier la bibliothèque à une classe(contenant une fonction main) utilisant la bibliothèque est de changer les arguments de la JVM. On peut le faire en sélectionnant la classe contenant la fonction main qui utilisera la bibliothèques et faire un clic droit, **Run as->Run Configurations**, à droite aller sous l'onglet **(x)Argument**, ensuite **VM Arguments**: Si l'option **-Djava.library.path** contient déjà un chemin ajouter deux points **:** à la fin de ce chemin et ensuite ajouter le chemin complet contenant la nouvelle bibliothèque dynamique. Si l'option n'est pas utilisée alors **-Djava.library.path=chemin/vers/dosssier/contenant/bibliothèque** Nous arrivons à simuler un système de Lorenz dont la dynamique de la variable Z est implémentée en C++ et celle des variables X et Y en Java. Pendant la simulation il a été nécessaire de limiter le nombre de chiffres après la virgule car il

[9] Si vous voulez copier ce fichier pour utiliser, tenez compte de cette remarque: Pour rendre certaines lignes visibles j'ai fait des sauts à la ligne qu'il faudra ajuster au besoin lorsque vous allez utiliser la commande cmake. Dans tous les cas, chaque projet a son propre CMakeLists.txt donc vous n'aurez pas besoin de le copier dans ce document.

y avait souvent des problèmes de représentation des nombres pour les données reçues de C++.