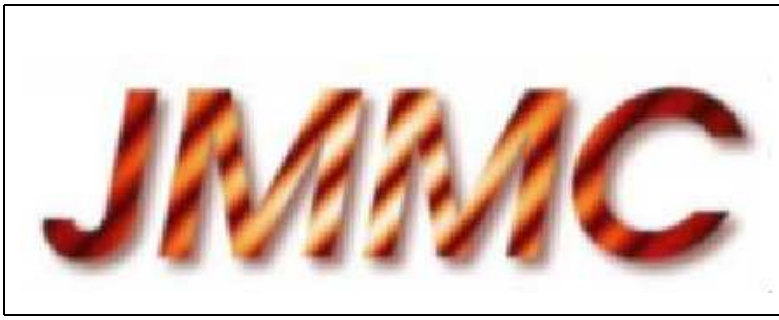

JMMC-MEM-2910-0002

Revision 1.0

Date: 3/09/2004

JMMC

Documentation technique du serveur de fonctions

Authors:
Thierry Stein <Thierry.Stein@obs.ujf-grenoble.fr> — LAOG/JMMC

Author: Thierry Stein Institute: LAOG/JMMC	Signature: Date: 3/09/2004
Approved by: Gérard Zins Institute: LAOG/JMMC	Signature: Date: 3/09/2004
Released by: Guillaume Mella Institute: LAOG/JMMC	Signature: Date: 3/09/2004

Change record

Revision	Date	Authors	Sections/Pages affected
	Remarks		
1.0	3/09/2004	G.Zins	all
	Adapted from EII/JRA4 Latex style		

Table des matières

Introduction

Ce document a pour but de présenter un synthèse de mon travail depuis avril, c'est à dire le début de mon stage.

Brièvement, le sujet consistait en la création d'un générateur de serveur de fonction, afin de pouvoir tester et utiliser une librairie de fonctions C ou C++ via un certain canal de communication (Stdin/stdout ou TCP/IP).

Dans le cadre de ce travail, j'ai découvert le logiciel SWIG, qui permet d'interfacer des fonctions C ou des classes C++ par l'intermédiaire d'un langage de programmation interprété (Tcl ou Python, par exemple). Après avoir testé ce logiciel, il m'est paru intéressant d'essayer de l'intégrer à mon propre programme, puisqu'il permettait de simplifier largement un grand nombre de fonctionnalités du générateur proprement dit.

Mon programme prend de préference en entrée un fichier header (mais un fichier source est aussi toléré).L'execution se déroule en trois étapes, décrites dans la suite de ce document :

- Création du fichier xml
- Ecriture du fichier interface
- Compilation de la librairie dynamique

1 Création du fichier XML

Dans cette première étape, nous prenons en entrée un fichier header (par exemple test.h) contenant l'ensemble des prototypes des fonctions que l'on souhaite tester. A l'aide de SWIG, nous transformons le fichier test.h en un fichier test_wrap.xml, à l'aide de la commande suivante :

swig -xml -includeall -I... -ignoremissing -module "nom_module" test.h

Comme ceci n'est pas forcément très clair de prime abord, voici le détail des différentes options :

- **includeall** permet d'ajouter dans l'arbre xml le contenu de fichiers supplémentaires, situés dans le répertoire spécifié par l'option -I. Par exemple, supposons que le fichier test.h inclus un fichier def.h. Le fichier test.h utilise des structures ou des typedef définis dans def.h. A l'aide de cette option, on ajoute dans l'arbre xml le contenu de def.h, ce qui permet de récupérer la définition des typedef utilisés dans test.h.
- **ignoremissing** va permettre d'ignorer tous les fichiers inclus qui ne sont pas trouvés. Elle s'utilise en parallèle avec l'option includeall. Ceci trouve son intérêt dans le cas, par exemple, ou test.h utilise le fichier stdio.h. On a aucun intérêt à recopier le fichier stdio.h dans l'arbre xml, et on ne spécifie donc pas son emplacement avec l'option -I. L'execution du programme continue donc normalement, en précisant juste à l'écran que le fichier stdio.h n'a pas été trouvé.
- **module "nom_module"** est le nom que l'on va affecter à la librairie dynamique de test. Ce nom est nécessaire au déroulement de SWIG, on peut même dire qu'il ne peut pas s'en passer.

On dispose en sortie de cette étape d'un fichier nommé par défaut "test_wrap.xml". Nous allons ensuite parcourir ce fichier xml afin d'en extraire toutes les informations nécessaires à l'écriture du fichier interface.

2 Forme du fichier XML

2.1 Squelette du fichier XML

Le fichier xml généré par SWIG à la forme suivante :

```
<top>
    <attributelist>
        <attribute name=".." value="..."/>
        <!-- Informations relatives au fichier -->
        <!-- Nom du fichier d'entree           -->
        <!-- Nom du fichier de sortie          -->
    </attributelist>
    <include>
        <!-- Declarationde typemaps, fonctions -->
        <!-- specifiques a SWIG, mais sans     -->
        <!-- interet pour nous                 -->
    </include>
    <include>
        <attributelist>
            <attribute name="name" value="nom_du_ficheir file.h"/>
        </attributelist>
        <include>
             <!-- Informations sur les fichiers  -->
             <!-- inclus dans le file.h          -->
        </include>

        <!-- Contenus du fichier file.h :       -->
        <!-- Declaration de type                -->
        <!-- Declaration de constantes          -->
        <!-- Declaration de variables globales  -->
        <!-- Declaration de fonctions           -->

    </include>
</top>
```

En réalité, toutes les balises sont de la forme suivante :

```
<nom_balise id="numero de la balise" addr="adresse memoire">
```

Afin de simplifier cette présentation, les champs id et addr sont volontairement omis. Je précise aussi que les balises fermantes

```
</nom_balise>
```

ne contiennent pas ces champs.

2.2 Declaration de constantes dans le fichier xml

Source :

```
#define mcsPROCNAME_LEN 19
```

Codage en xml :

```
<constant>
    <attributelist>
        <attribute name="sym_name" value="mcsPROCNAME_LEN"/>
        <attribute name="name" value="mcsPROCNAME_LEN"/>
        <attribute name="feature_immutable" value="1"/>
        <attribute name="value" value="19"/>
        <attribute name="storage" value="%constant"/>
        <attribute name="type" value="int"/>
        <attribute name="sym_symtab" value="4023c128"/>
        <attribute name="sym_overname" value="__SWIG_0"/>
    </attributelist >
</constant >
```

2.3 Declaration de types dans le fichier xml

2.3.1 Renommage de type simple

Source :

```
typedef float mcsFLOAT;
```

Codage en xml :

```
<cdecl>
    <attributelist>
        <attribute name="sym_name" value="mcsFLOAT"/>
        <attribute name="name" value="mcsFLOAT"/>
        <attribute name="decl" value=""/>
        <attribute name="storage" value="typedef"/>
        <attribute name="type" value="float"/>
        <attribute name="sym_symtab" value="4023c128"/>
        <attribute name="sym_overname" value="__SWIG_0"/>
    </attributelist >
</cdecl >
```

2.3.2 Déclaration de tableau

Source :

```
typedef unsigned char mcsBYTES4[4];
```

Codage en xml :

```
<cdecl>
    <attributelist>
        <attribute name="sym_name" value="mcsBYTES4"/>
        <attribute name="name" value="mcsBYTES4"/>
        <attribute name="decl" value="a(4)."/>
        <attribute name="storage" value="typedef"/>
        <attribute name="type" value="unsigned char"/>
        <attribute name="sym_symtab" value="4023c128"/>
        <attribute name="sym_overname" value="__SWIG_0"/>
    </attributelist >
</cdecl >
```

On constate que cette déclaration est sensiblement similaire au renommage de type simple. En réalité, seul le champ decl change. Il n'est plus vide mais contient la dimension du tableau. Pour les tableaux à plusieurs dimensions, par exemple [3][4], celles ci sont codées sous la forme a(3).a(4).

2.3.3 Types énumérés

Source ;

```
typedef enum
{
    FAILURE = 1,
    SUCCESS
} mcsCOMPL_STAT;
```

Code en xml :

```
<enum>
    <attributelist>
        <attribute name="sym_name" value="mcsCOMPL_STAT"/>
        <attribute name="name" value="mcsCOMPL_STAT"/>
        <attribute name="tdname" value="mcsCOMPL_STAT"/>
        <attribute name="storage" value="typedef"/>
        <attribute name="unnamed" value="$unnamed1$"/>
    </attributelist >

    <enumitem>
        <attributelist>
            <attribute name="sym_name" value="FAILURE"/>
            <attribute name="name" value="FAILURE"/>
            <attribute name="enumvalue" value="1"/>
            <attribute name="feature_immutable" value="1"/>
            <attribute name="value" value="FAILURE"/>
            <attribute name="type" value="int"/>
            <attribute name="_last" value="40241278"/>
            <attribute name="sym_symtab" value="4023c128"/>
            <attribute name="sym_overname" value="__SWIG_0"/>
         </attributelist >
    </enumitem >
    <enumitem>
         <attributelist>
            <attribute name="sym_name" value="SUCCESS"/>
            <attribute name="name" value="SUCCESS"/>
            <attribute name="enumvalue" value="FAILURE+1"/>
            <attribute name="feature_immutable" value="1"/>
            <attribute name="value" value="SUCCESS"/>
            <attribute name="type" value="int"/>
            <attribute name="sym_symtab" value="4023c128"/>
            <attribute name="sym_overname" value="__SWIG_0"/>
         </attributelist >
    </enumitem >
</enum >
```

2.3.4 Types structurés

Le cas des structures est légèrement plus complexe. On distingue en effet quatre cas.

1. Premier cas

   Source :

```
typedef struct
{
int reel;
int imaginaire;
} Complexe;
```

   Code xml :

```
<class>
    <attributelist>
        <attribute name="unnamed" value="Complexe"/>
        <attribute name="name" value="Complexe"/>
        <attribute name="sym_symtab" value="4023c128"/>
        <attribute name="symtab" value="4023e898"/>
        <attribute name="allows_typedef" value="1"/>
        <attribute name="typepass_visit" value="1"/>
        <attribute name="allocate_visit" value="1"/>
        <attribute name="kind" value="struct"/>
        <attribute name="sym_name" value="Complexe"/>
        <attribute name="allocate_default_constructor"/>
        <attribute name="allocate_default_destructor"/>
        <attribute name="tdname" value="Complexe"/>
        <attribute name="module" value="complexe"/>
        <attribute name="sym_overname" value="__SWIG_0"/>
        <attribute name="storage" value="typedef"/>
        <typescope>
            <attributelist>
                <attribute name="name" value="Complexe"/>
                <typetab>
                    <attributelist>
                    </attributelist >
                </typetab >
                <attribute name="parent" value="4023c1a8"/>
                <attribute name="qname" value="Complexe"/>
               <attribute name="symtab" value="4023e898"/>
                    </attributelist >
     </typescope >
    </attributelist >

    <cdecl>
        <attributelist>
            <attribute name="sym_name" value="reel"/>
            <attribute name="name" value="reel"/>
            <attribute name="decl" value=""/>
            <attribute name="type" value="int"/>
```

```
            <attribute name="sym_symtab" value="4023e898"/>
            <attribute name="sym_overname" value="__SWIG_0"/>
        </attributelist >
    </cdecl >
    <cdecl>
        <attributelist>
            <attribute name="sym_name" value="imaginaire"/>
            <attribute name="name" value="imaginaire"/>
            <attribute name="decl" value=""/>
            <attribute name="type" value="int"/>
            <attribute name="sym_symtab" value="4023e898"/>
            <attribute name="sym_overname" value="__SWIG_0"/>
        </attributelist >
    </cdecl >
</class >
```

2. Deuxième cas

   Source :

```
struct complexe
{
        int reel;
        int imaginaire;
};
```

Code xml :

```
<class>
    <attributelist>
        <attribute name="name" value="Complexe"/>
        <attribute name="sym_symtab" value="4023c128"/>
        <attribute name="symtab" value="4023e8c8"/>
        <attribute name="allows_typedef" value="1"/>
        <attribute name="typepass_visit" value="1"/>
        <attribute name="allocate_visit" value="1"/>
        <attribute name="kind" value="struct"/>
        <attribute name="sym_name" value="complexe"/>
        <attribute name="allocate_default_constructor"/>
        <attribute name="allocate_default_destructor"/>
        <attribute name="module" value="complexe"/>
        <attribute name="sym_overname" value="__SWIG_0"/>
        <typescope>
            <attributelist>
                <attribute name="name" value="complexe"/>
                <typetab>
                    <attributelist>
                    </attributelist >
                </typetab >
                <attribute name="parent" value="4023c1a8"/>
                <attribute name="qname" value="complexe"/>
                <attribute name="symtab" value="4023e8c8"/>
            </attributelist >
         </typescope >
    </attributelist >

    <!-- La declaration des autres champs de la structure -->
    <!-- est identique au cas 1-->

</class>
```

On constate que les modifications sont mineures. Deux balises disparaissent :

```
<attribute name="unnamed" />
<attribute name="storage"value="typedef"/>
```

Pour savoir si on a affaire à un typdef ou non, il faut donc regarder s'il existe une balise storage.

3. Troisième cas

   Source :

```
typedef struct complexe
{
        int reel;
        int imaginaire;
} Complexe;
```

   Code xml :

```
<class>
   <attributelist>

<!-- Definition de la structure comme dans le cas 2 -->
<!--  avec une balise supplementaire,               -->
<!--  plus la balise sym_name modifie               -->

       <attribute name="sym_name" value="Complexe"/>
       <attribute name="tdname" value="Complexe"/>
   </attributelist>
</class>

<cdecl id="175" addr="4023eaf8" >
    <attributelist id="176" addr="4023eaf8" >
        <attribute name="name" value="Complexe"/>
        <attribute name="decl" value=""/>
        <attribute name="storage" value="typedef"/>
        <attribute name="type" value="struct complexe"/>
    </attributelist >
</cdecl >
```

   La définition est décomposée en deux : d'un coté la définition de la structure comme dans le cas 2.On a alors un type struct complexe. Puis ce type est renommé comme dans le cas de rennomage de type simple.

4. Quatrième cas

   Source :

```
struct complexe
{
        int reel;
        int imaginaire;
};
typedef struct complexe Complexe;
```

   Cette fois ci, le code xml généré est le suivant : D'un coté, exactement le même que dans le cas 2 pour la déclaration de struct complexe.
   De l'autre, un renommage de type simple comme dans le cas 3.

Voici l'ensemble des définitions de structures que telles qu'on peut les trouver dans un programme C, et telles qu'elles sont transformées en xml. Certaines formes sont cependant moins utilisées que d'autres, mais il faut tout de même se méfier lors de l'application des feuilles de style.

Dans le cas de structure contenant une autre structure, SWIG effectue une décomposition. La structure interne se trouve définie à part. Voir à ce sujet l'exemple de gmain dans le répertoire "projet/exemple/gmain/".

2.3.5 Type union

Le type union est très similaire au type structure. En fait,une seule balise change :

```
<attribute name="kind" value="struct"/>
```

deviens :

```
<attribute name="kind" value="union"/>
```

Sinon, les unions sont, de la même manière que les structures, déclarées entre les balises

```
<class> </class>
```

2.4 Declaration de variables dans le fichier xml

Ce cas concerne les variables globales. Supposons que l'on ait une variable globale de la forme : int global. Le code xml sera :

```
<cdecl id="134" addr="4023e8d8" >
    <attributelist id="135" addr="4023e8d8" >
        <attribute name="sym_name" value="global"/>
        <attribute name="name" value="global"/>
        <attribute name="decl" value=""/>
        <attribute name="type" value="int"/>
        <attribute name="sym_symtab" value="4023c128"/>
        <attribute name="sym_overname" value="__SWIG_0"/>
    </attributelist >
</cdecl >
```

Dans le cas d'un tableau, par exemple : int global[4], Il n'y a qu'une balise qui change :

```
<cdecl>
    <attributelist>
        <attribute name="decl" value="a(4)."/>
    </attributelist>
</cdecl>
```

2.5 Declaration de fonctions dans le fichier xml

Ce dernier traite des prototypes de fonctions. un exemple simple suffira á illustrer ce cas. On a une fonction dont le prototype est : double additionne(double a, double b) ;
Le code xml est le suivant :

```
<cdecl>
    <attributelist>
        <attribute name="sym_name" value="additionne"/>
        <attribute name="name" value="additionne"/>
        <attribute name="decl" value="f(double,double)."/>
        <parmlist>
            <parm>
                <attributelist>
                    <attribute name="name" value="a"/>
                    <attribute name="type" value="double"/>
                </attributelist >
            </parm >
            <parm>
                <attributelist>
                    <attribute name="name" value="b"/>
                    <attribute name="type" value="double"/>
                </attributelist >
             </parm >
        </parmlist >
        <attribute name="type" value="double"/>
        <attribute name="sym_symtab" value="4023c128"/>
        <attribute name="sym_overname" value="__SWIG_0"/>
        </attributelist >
</cdecl >
```

Les paramètres se trouvent entre les balises

`<parmlist> <parm>`

. Le type de retour se trouve dans la balise

`<attribute name="type">`

. Dans le cas de fonctions sans paramètre, il n'y a simplement pas de balise

`<parmlist>`

.

On remarque aussi que la definition d'une fonction est proche de celle d'une variable globale. La différence se trouve dans la balise

`<attribute name="decl">`

. Si le champ value contient 'f(', on a alors affaire à une fonction. Sinon, c'est une variable globale ou un typedef. Il faut alors regarder la balise

`<attribute name="storage">`

pour savoir si c'est un typedef.

2.6 Synthèse et cas particuliers

Globalement, le fichier xml stocke les informations de la manière suivante :

```
<top>
   <include>

      <attributelist>
         Informations sur le fichier
      <attributelist>

      <constant>
         Definition des constantes
      </constant>

      <enumerate>
         definition d'enumeration
      </enumerate>

      <class>
         Definition de structure et union
      </class>

      <cdecl>
         Typdef
         Variables globales
         Prototypes de fonctions
      </cdecl>

   </include>
</top>
```

3 Ecriture du fichier interface

A partir du fichier xml crée dans la première étape, nous allons appliquer une feuille de style XSL, qui va générer le fichier interface nécessaire à SWIG. La création du fichier "test.i" s'effectue de la manière suivante :

1. Ecriture du nom du module
   Le fichier interface doit commencer par la ligne suivante :
   %module nom_module
2. Ecriture des noms des fichiers à inclure.
   On ajoute de manière générale ces trois lignes :
   %{
   #include "test.h"
   %}
3. Ecriture des entêtes de fonctions
   On écrit les prototypes de l'ensemble des fonctions qui seront disponibles et utilisables. Actuellement, le seul point délicat est le cas d'une fonction ayant un appel de fonction en paramètre.
4. Ecriture des définitions des types utilisés en entré/sortie des fonctions
   Dans ce dernier point, on ajoute les définitions des typedef qui se trouvent en paramètre des fonctions à interfacer. Dans le cas de types structurés, on ajoute une fonction pour créer facilement une structure, ainsi qu'une fonction d'affichage.

Une autre option, plus simple, consite à créer le fichier interface suivant :

```
%module nom_module
%{
include "test.h"
%}
%include "test.h"
```

Cette deuxième solution est intéressante car simple. On est certain que lors de l'étape d'écriture de ce fichier avec XSL, il n'y aura pas de problème lié à un prototype particulié, comme par exemple une fonction ayany un appel de fonction en paramètre. Cette solution a de plus l'avantage de la vitesse. En effet, pour une librairie comme cfitsio, le fichier header contient les prototypes de près de mille fonctions. La réecriture de chaque prototype dans le fichier interface avec XSL prend plusieurs minutes. Avec cette méthode, plus de soucis.

Cependant, cette solution possède plusieurs inconvénients. Elle empêche de ne selectionner que ce qui est nécessaire, ce qui a pour conséquence d'alourdir le code généré par SWIG. De plus, ce fichier simplifié n'est pas suffisant pour interfacer certaines fonctionnalités du C, comme par exemple les tableaux d'entiers. Sans code supplémentaires, une variable globale de type tableau d'entier sera déclarée en lecture seule, ce qui peut probablement se réveler gênant lors de la phase de test.

On dispose en sortie d'un fichier test.i qui va permettre à SWIG de créer la librairie dynamique de test.

4 Compilation de la librairie dynamique

Une fois le fichier test.i écrit, SWIG peut reprendre son déroulement normal. Il va créer une librairie dynamique nommé "nom_module.so" avec les trois lignes suivantes

```
> swig -tcl test.i
> gcc -c -fpic test.c test_wrap.c
> gcc -shared test.o test_wrap.o -o test.so
```

Ceci ci permet ensuite d'appeler et de tester les fonctions C via le langage Tcl. Pour Python, le processus est le même, seule la deuxième ligne change :

```
> gcc -I/usr/include/python2.3/ -c -fpic test.c test_wrap.c
```

Si au lieu d'avoir un fichier source C, on dispose d'une librairie,le processus devient :

```
> swig -tcl test.i
> gcc -c -fpic test_wrap.c # Pas de fichier test.c
> gcc -shared -l"nom de la librairie" -L"chemin d'acces" test_wrap.o -o test.so
```

5 Conclusion

Dans ce document, je me suis attaché à décrire la structure du fichier xmlqui est ensuite "parsé" à l'aide des outils xsl. En ce qui concerne le code xsl et les exemples d'execution, tout cela se trouve situé sur le serveur mariotti, dans le répertoire projet/xsl_source. Des exemples d'execution se trouvent dans le répertoire projet/exemple.

il existe aussi un script shell qui permet d'automatiser le processus et de génerer automatiquement le makefile "qui va bien". Ceci se trouve plus détaillé dans les exemples d'execution ainsi que dans mon rapport de stage.