

Rapport Projet STL

# O'Xul : Interface Xul en O'Caml

Xiaoyi CHEN      [xiaoyi.chen@etu.upmc.fr](mailto:xiaoyi.chen@etu.upmc.fr)  
Pedro CAVADAS    [pedro.cavadas@etu.upmc.fr](mailto:pedro.cavadas@etu.upmc.fr)

encadrant: Emmanuel Chailloux  
[emmanuel.chailloux@pps.jussieu.fr](mailto:emmanuel.chailloux@pps.jussieu.fr)

Université Pierre et Marie Curie (Paris 6)

25 juin 2006

# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 Technologies Mozilla</b>	<b>5</b>
1.1 XUL	5
1.1.1 Introduction	5
1.1.2 Structure d'un fichier XUL	6
1.2 XPCOM	10
1.2.1 Introduction	10
1.2.2 Création d'un XPCOM	10
1.3 XULRunner	17
1.3.1 Introduction	17
1.3.2 Structure d'une application	17
<b>2 Intégration de XUL dans O'Caml</b>	<b>19</b>
2.1 Etat de l'art	19
2.1.1 Introduction	19
2.1.2 Choix de la bibliothèque graphique	19
2.2 Interprète Xul en O'Caml	22
2.2.1 Analyseur syntaxique pour XUL : XML-Light	22
2.2.2 Interprétation du code JavaScript : SpiderCaml	22
2.2.3 Analyseurs syntaxiques pour CSS et DTD	24
2.3 Appel de code O'Caml en JavaScript	24
2.3.1 Étude de faisabilité	24
2.3.2 Récupération de valeurs O'Caml	25
2.3.3 Conversion de valeurs O'Caml vers JavaScript	25
2.3.4 Interprétation de code O'Caml en JavaScript	28
<b>3 Interfaçage entre XPCOM et O'Caml</b>	<b>30</b>
3.1 Etude de faisabilité	30
3.1.1 Objective Caml	30
3.1.2 Appel de fonction O'Caml dans un composant XPCOM	30
3.1.3 Extension Firefox utilisant un XPCOM appelant du O'Caml	35
3.1.4 Code O'Caml appelant des méthodes C++ d'XPCOM	36
3.1.5 Pgcd graphique	42
3.1.6 nsString et nsCString	47
3.2 Application O'Xul	48
3.2.1 Interface graphique : XUL	48
3.2.2 Gestion des événements	49
3.2.3 Initialisations : XPCOM et événements	53

3.2.4	Lancement du programme . . . . .	54
3.2.5	Marche à suivre : résumé . . . . .	54
	<b>Conclusion</b>	<b>55</b>
	<b>Bibliographie</b>	<b>58</b>

# Introduction

Mozilla, le célèbre navigateur libre, est basé sur un ensemble de technologies qui, comme le dit ses développeurs, se veulent le plus portable possible. XUL et XPCOM en font partis et répondent à ces critères. XUL est un langage au format XML qui décrit les éléments constituant une fenêtre graphique tel qu'un bouton, un champ de texte, un peu comme GTK<sup>1</sup> et ses fichiers de description d'interfaces. XPCOM est le nom donné à l'ensemble des composants qui constituent le coeur des applications Mozilla. Un composant est la somme d'une interface et d'une classe d'implémentation en C++ (ou en d'autres langages pris en charge).

On cherche au travers de ce projet à tester l'intégration et l'interfaçage de ces deux technologies avec Objective Caml. Cela dans le but de donner à O'Caml un moteur de rendu qui a fait ses preuves et qui ne cesse de s'améliorer comme l'intégration bientôt d'un rendu pour la 3D, ainsi que la possibilité d'utiliser toutes les fonctionnalités que propose XPCOM à commencer par la plus connue, la navigation web. Cependant on ne voudrait pas se limiter à bénéficier de ces avantages que dans un seul sens, il faudrait aussi pouvoir par exemple à partir du navigateur Firefox exécuter et même écrire du code O'Caml.

## Cahier des charges

Le travail qui nous est demandé, est de réaliser un interprète XUL en O'Caml de sorte à ce qu'en lisant un fichier d'interface, O'Caml crée la fenêtre et tous les *widgets* qui sont décrits dans le fichier. De plus, on ne veut pas se limiter à du Javascript dans les fichiers XUL ; on cherche à intégrer des fonctions O'Caml pour avoir encore plus de possibilités. La prise en charge des fichiers CSS, DTD et peut-être les fichiers RDF n'est pas primordial mais sont compris dans les demandes.

Parallèlement, il faut étudier la faisabilité de l'interfaçage de code O'Caml avec le C++ d'XPCOM, en commençant par voir s'il est possible d'intégrer du code O'Caml dans XPCOM, plus exactement si les méthodes C++ d'XPCOM peuvent appeler des fonctions O'Caml. L'inverse est aussi inclus dans le cahier des charges, une application autonome simple serait une bonne illustration. Faire en sorte que O'Caml puisse communiquer avec les composants XPCOM et que ces mêmes composants puissent répondre est la finalité de cette deuxième partie du travail.

## Répartition du travail

Le projet se divise donc principalement en deux parties : l'interprète XUL et l'interfaçage XPCOM-O'Caml. La première partie est attribuée à Pedro Cavadas tandis que la seconde à Xiaoyi Chen. Notez que ces deux parties sont traitées séparément et que ce n'est seulement qu'à la fin, si le temps le permet, que nos travaux seront mis en commun.

---

<sup>1</sup>The GIMP Tool Kit

Le travail est régit par un compte rendu toutes les semaines à l'encadrant qui dirige, et demande ce qu'il attend au fil de la progression et des découvertes des possibilités. Les choix d'implémentation sont donc guidés par ceux de l'encadrant.

**Plan**

Concernant la planification, ce rapport est écrit de manière à présenter les technologies Mozilla au début pour avoir un aperçu et avoir une idée de comment ils fonctionnent. Après cette partie, le lecteur devrait savoir comment écrire une interface XUL, et ce qu'il doit faire pour créer un composant XPCOM. S'ensuit en deuxième partie le travail réalisé sur l'interprète XUL. La troisième et dernière partie concernera tout ce qui touche à XPCOM.

# Chapitre 1

## Technologies Mozilla

### 1.1 XUL

#### 1.1.1 Introduction

Avec l'évolution des technologies, les interfaces graphiques des applications deviennent de plus en plus complexes, que ce soit au niveau de la diversité des composants proposés, au niveau du comportement, ou au niveau visuel. Une conséquence directe de cette avancée est que plus une interface utilisateur est évoluée, plus le code correspondant devient volumineux, voire complexe. Il en résulte que le code propre à l'application est souvent moins important que le code propre à l'interface graphique, ce qui peut le rendre moins lisible et plus difficile à corriger.

XUL<sup>1</sup> — prononcez « *zoul* » — est l'acronyme pour *XML-based User interface Language*. C'est un langage multi-plateformes basé sur XML permettant de décrire l'interface graphique d'une application en dehors du code. Comme vous pouvez vous en douter, ceci apporte bien des avantages, en particulier le fait de faciliter la lecture — et donc la compréhension — du code, et par conséquent, de rendre plus aisée la maintenance d'une application.

Ce langage a été créé dans le cadre du projet Mozilla, et est aujourd'hui utilisé par toutes les applications et leurs extensions que propose la fondation Mozilla, et notamment les célèbres navigateurs Mozilla et Firefox. Cependant, il est tout à fait possible de l'utiliser pour créer des applications indépendantes.

Le langage XUL se base sur plusieurs standards pour permettre de personnaliser facilement une interface utilisateur, dont JavaScript, CSS, DTD et RDF. Ceci permet un apprentissage très rapide pour les développeurs ayant des bases en programmation web. La figure 1.1 montre comment sont reliées toutes ces technologies pour produire une interface graphique grâce au moteur de rendu *Gecko* intégré à XulRunner.

---

<sup>1</sup><http://www.mozilla.org/projects/xul/>

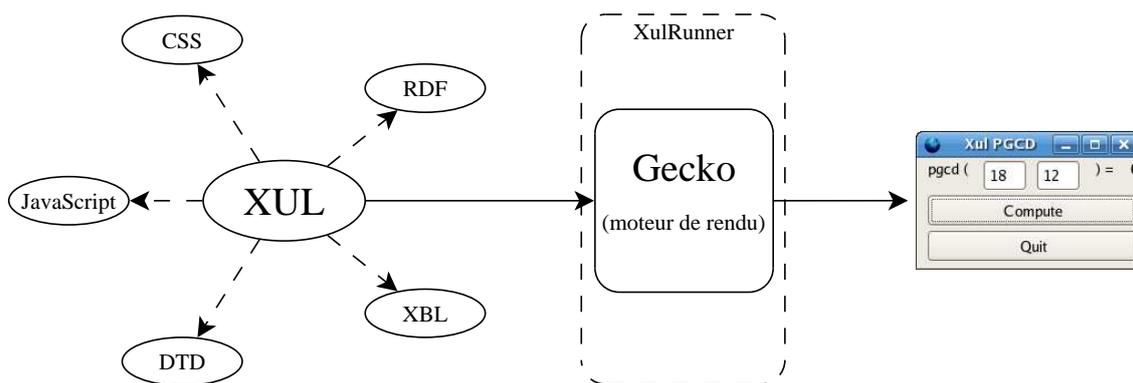


FIG. 1.1 – Exécution d'un fichier XUL

### 1.1.2 Structure d'un fichier XUL

Un fichier XUL porte l'extension `.xul`. Il peut être exécuté soit avec Firefox ou Mozilla, soit à l'aide de XulRunner (voir section 1.3). Le code source 1.2 est un exemple relativement simple d'une application permettant de calculer le PGCD de deux entiers, et la figure 1.3 montre son exécution à l'aide de XulRunner.

Comme vous pouvez le constater, la gestion des clics de souris se fait à l'aide de l'attribut `oncommand` dont la valeur est une commande JavaScript. Le calcul s'effectue ensuite à l'aide d'un script JavaScript défini à l'intérieur-même du fichier XUL, entre des balises `<script>`. Mais il est tout à fait possible d'écrire les scripts dans un fichier JavaScript annexe `.js`, puis de l'appeler dans le fichier XUL, à l'aide de la balise :

```
<script src="fichier.js" />
```

De plus, il est possible de personnaliser facilement l'apparence des widgets grâce aux feuilles de style CSS. Par exemple, il est possible de définir une couleur de fond pour les *labels*, de changer la police du texte des boutons, etc. Pour cela, il suffit de créer le fichier CSS approprié aux styles que nous voulons appliquer, et de l'importer en modifiant la deuxième ligne du code 1.2 par :

```
<?xml-stylesheet href="fichier.css" type="text/css"?>
```

Enfin, l'un des points les plus importants de XUL est la facilité à proposer une interface graphique écrite dans la langue de l'utilisateur. En effet, peu de langages offrent des outils nécessaires à la *localisation* d'une application. Or aujourd'hui, avec la distribution mondiale des logiciels, il est important, voire nécessaire, de proposer des applications écrites dans différentes langues pour le confort des utilisateurs.

XUL peut se vanter de proposer un outils simple et efficace pour la localisation d'une interface graphique, grâce aux fichiers DTD. Pour proposer une application dans différentes langues, il suffit de recréer la hiérarchie de dossiers et les fichiers nécessaires à l'exécution de notre interface à l'aide de XulRunner (voir section 1.3), puis de placer les fichiers DTD des différentes langues dans les répertoires correspondants. Par exemple, pour le Français, il faut placer le fichier DTD correspondant à la langue française dans le dossier `chrome/locale/fr-FR`.

Pour traduire une chaîne à l'aide d'un fichier DTD, il faut écrire une entité dans ce fichier, comme dans l'exemple suivant :

```
<!ENTITY chaine "Chaîne traduite">
```

FIG. 1.2 – Fichier pgcd.xul

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<window
  id="root"
  title="Xul PGCD"
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <script>
    function pgcd(n, m) {
      if (m == 0)
        return n;
      else
        return pgcd(m, n % m);
    }

    function calculer() {
      try {
        var n = Number(document.getElementById('n').value);
        var m = Number(document.getElementById('m').value);
        var res = pgcd(n, m);

        document.getElementById('result').value = res;
      }
      catch(error) {
        alert("Erreur :\n\n" + error);
      }
    }
  </script>

  <vbox>
    <hbox>
      <label value="pgcd ( " />
      <textbox id="n" width="40" />
      <textbox id="m" width="40" />
      <label value=" ) = " />
      <label id="result" value="" />
    </hbox>
    <button id="button" label="Compute" oncommand="calculer();" />
    <button id="quit" label="Quit" oncommand="self.close();" />
  </vbox>

</window>

```

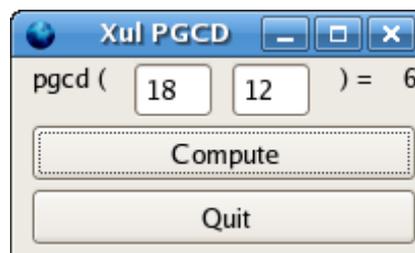


FIG. 1.3 – Exécution de pgcd.xul avec XulRunner

Puis, il suffit d’utiliser l’entité ci-dessus dans notre code XUL, par exemple dans un bouton :

```
<button label="&chaine;" />
```

Comme vous l’avez sûrement compris, grâce à l’entité précédente, toutes les occurrences de “&chaine;” dans le fichier XUL seront remplacées par le texte “**Chaîne traduite**”.

À présent, reprenons notre application 1.2 permettant de calculer le PGCD de deux entiers, et mettons en pratique tout ce que nous venons de voir. Nous allons remanier notre fichier `pgcd.xul` afin de séparer les scripts JavaScript dans un fichier `pgcd.js`, d’afficher le résultat en rouge et en caractères gras à l’aide d’une feuille de style `pgcd.css`, et de traduire le texte des boutons « Quit » et « Compute » en Français à l’aide d’un fichier `french.dtd`. Voici les fichiers correspondants :

FIG. 1.4 – Fichier `pgcd.xul`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="pgcd.css" type="text/css"?>

<!DOCTYPE window SYSTEM "chrome://pgcd/locale/french.dtd">

<window
  id="root"
  title="Xul PGCD"
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <script src="pgcd.js"/>

  <vbox>
    <hbox>
      <label value="pgcd ( "/>
      <textbox id="n" width="40"/>
      <textbox id="m" width="40"/>
      <label value=" ) = "/>
      <label id="result" value="" />
    </hbox>
    <button id="button" label="&btnCompute.label;" oncommand="calculer();"
      />
    <button id="quit" label="&btnQuit.label;" oncommand="self.close();"/>
  </vbox>

</window>
```

FIG. 1.5 – Fichier `pgcd.css`

```
@import url(chrome://global/skin/);

#result {
  font-weight: bold;
  color: red;
}
```

FIG. 1.6 – Fichier french.dtd

```
<!ENTITY btnCompute.label "Calculer">
<!ENTITY btnQuit.label "Quitter">
```

FIG. 1.7 – Fichier pgcd.js

```
function pgcd(n, m) {
  if (m == 0)
    return n;
  else
    return pgcd(m, n % m);
}

function calculer() {
  try {
    var n = Number(document.getElementById('n').value);
    var m = Number(document.getElementById('m').value);
    var res = pgcd(n, m);

    document.getElementById('result').value = res;
  }
  catch(error) {
    alert("Erreur :\n\n" + error);
  }
}
```

Notez la ligne suivante dans le fichier CSS 1.5 :

```
@import url(chrome://global/skin/);
```

Cette ligne est particulièrement importante dans la mesure où nous redéfinissons le style entier de la fenêtre graphique. Cette ligne permet d'utiliser les styles par défaut, en redéfinissant les styles suivant, écrits manuellement. Sans cette ligne, l'interface n'aurait plus aucun style, mis à part ceux que nous redéfinissons — en l'occurrence pour le *label* du résultat — et s'afficherait par conséquent avec un fond transparent.

Voici le résultat de nos fichiers 1.4, 1.5, 1.6 et 1.7 exécutés à l'aide XulRunner :

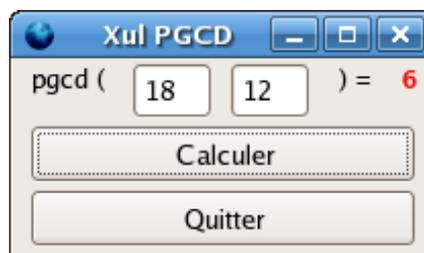


FIG. 1.8 – Exécution de pgcd.xul, pgcd.js, pgcd.css et french.dtd à l'aide de XulRunner

## 1.2 XPCOM

### 1.2.1 Introduction

XPCOM est le noyau des technologies Mozilla<sup>2</sup>. Que ce soit le moteur de rendu Gecko ou la gestion des téléchargements d'un navigateur, ce sont tous des composants XPCOM. Pour plus de facilité, je me permets un abus de langage, dorénavant j'appellerai un composant XPCOM, un XPCOM.

XPCOM est l'abréviation de Cross Platform Component Object Module. Cette technologie consiste à découper l'application en petites pièces et à séparer l'interface des composants de l'implémentation. Cette séparation permet l'interopérabilité entre les systèmes puisque les interfaces sont identiques. En effet les composants sont accessibles via les fichiers d'extensions *.xpt*, et on ne s'occupe plus de savoir si les fichiers d'implémentations sont écrits pour tel ou tel système. L'accès aux composants se fait via un langage multi-plateforme, en l'occurrence Javascript. En tant que langage de script, il n'a pas besoin de différencier par exemple un système Windows d'un système basé sur Unix.

Les applications n'ont pas forcément besoin de tous les composants, c'est pourquoi lors du lancement d'une application seuls les composants nécessaires sont chargés ; le chargement dynamique économise les ressources mais il entraîne le redémarrage des applications à chaque modification pour que celle-ci soit effective, comme par exemple l'installation de nouvelles extensions dans Firefox.

Un XPCOM peut être écrit dans différents langages, parmi ceux-ci il y a le C, C++, Javascript, Perl, Python, Java, etc. .Le langage supporté en natif est le C++. Les autres langages utilisent un système d'interfaçage avec les interfaces (au sens interface de classes) des composants. Ceci est une autre force de XPCOM : une application ne se soucie pas du langage dans lequel est implémenté le composant auquel il accède, elle sait juste que l'interface propose une telle fonction ou méthode. Bien sûr ceci provient du principe de la programmation orientée objet, mais appliqué à une multitude de langages.

### 1.2.2 Création d'un XPCOM

Dans cette section je vais détailler les étapes de la création d'un composant XPCOM en détaillant toutes les subtilités que j'ai trouvé. Un XPCOM peut être écrit dans plusieurs langages, bien que le C n'est pas un langage à objet, les autres sont ou ont une couche objet ; dans notre cas j'ai choisi de détailler la création en C++ car c'est le plus intéressant pour nous, et c'est ce sur quoi je vais me baser pour l'interfaçage avec O'Caml. On va utiliser le même exemple sur le pgcd : un XPCOM qui calcul le pgcd et retourne le résultat.

La création d'un composant XPCOM peut se faire de deux manières. Soit on incorpore le code de notre XPCOM dans le code source de Firefox ou XULRunner, ainsi il sera automatiquement compilé et enregistré dans la liste des composants quand on compile toute l'application. Avantage de cette méthode : le Makefile est facile à écrire et on ne se complique pas la tâche avec les problèmes de dépendances. Inconvénient : c'est trop lent, on doit attendre que le compilateur parcourt toute l'arborescence du code source de Firefox (ou XULRunner). Soit, et c'est la méthode qu'on applique, on crée notre composant séparément et on fait attention à compiler avec les bonnes en-têtes et lier avec les bonnes bibliothèques. Beaucoup d'erreurs et de temps perdu proviennent de cela, mais le gain en temps est non négligeable une fois corrigées.

---

<sup>2</sup>[www.mozilla.org](http://www.mozilla.org)

## Schéma de conception

Jetons un coup d'oeil sur le schéma global :

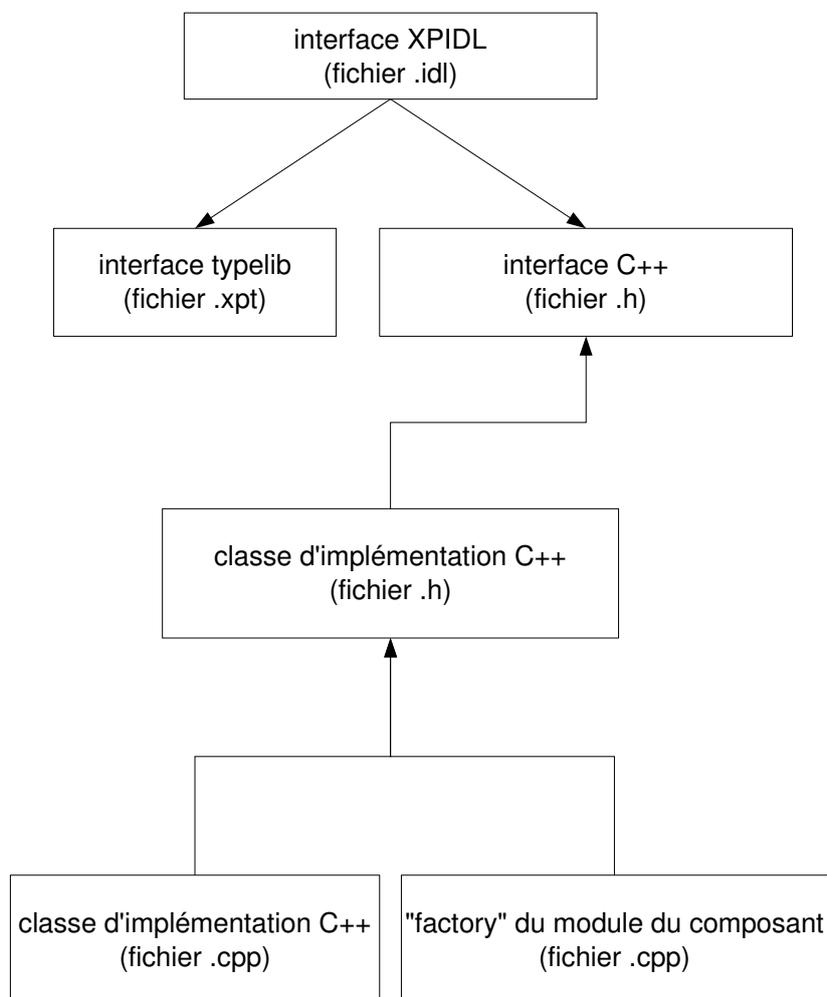


FIG. 1.9 – Schéma global d'un composant XPIDL

Pour reprendre le même principe de séparation des développeurs Mozilla : on a d'une part l'écriture de l'interface, qui est souvent dans le répertoire *public*, et d'autre part l'implémentation qui est dans *src*. Intéressons-nous d'abord sur l'écriture de l'interface.

### Les fichiers d'interfaces

Les fichiers d'interfaces sont des fichiers IDL (Interface Definition Language), dont la syntaxe est très similaire à CORBA IDL. Pour suivre leur nomenclature, Mozilla décide

de nommer leur version XPIDL. Pour ceux qui ne connaissent pas CORBA IDL, on peut dire que la syntaxe XPIDL est similaire à la syntaxe objet de C++ à peu de chose près.

Un fichier d'interface IDL contient les attributs, le prototype des méthodes et les interfaces qu'elle hérite. Les composants qui implémentent cette interface doit donc fournir les services (méthodes) qu'elle décrit.

Voici notre exemple du `pgcd`, il ne contient qu'une méthode :

```
#include "nsISupports.idl"

[scriptable, uuid(83caf949-e207-47f6-9788-3d51e56a3074)]
interface nsIPgcd : nsISupports
{
    long pgcd(in long n, in long m);
};
```

FIG. 1.10 – `nsIPgcd.idl`

Détaillons un peu :

- presque toutes, sinon toutes, les interfaces héritent de `nsISupports` donc il faut l'inclure; si la méthode prend un paramètre de type (au hasard) `nsIDOMElement`, il faut le préciser en insérant

```
interface nsIDOMElement;
```

juste avant

```
interface nsIPgcd : nsISupports
```

sinon le compilateur `xpidl` ne connaîtra pas le type du paramètre.

- pour rendre le composant lisible et chargeable par Javascript, on le précise par le mot-clé `scriptable`; on peut rendre une méthode d'un composant `scriptable` non accessible par Javascript en écrivant `[noscript]` devant.
- une interface doit posséder un id unique, qu'on génère avec `uuidgen` (sous Linux)
- les attributs et les méthodes doivent respecter le type de XPIDL; le type `long` employé dans l'exemple est l'équivalent du `int` de C.
- ne pas oublier le point-virgule final.

Après l'écriture du fichier `idl`, il faut traduire le code IDL en code C++. Il faut aussi créer le fichier `typelib` d'extension `.xpt` qui contient les informations compilées de l'interface. Un outil bien pratique permet de compiler tout cela automatiquement, la commande `xpidl`. Pour la génération du fichier `nsIPgcd.xpt` on utilise cette commande

```
xpidl -m typelib -I $MOZILLA_SDK/idl nsIPgcd.idl
```

Pour le fichier `nsIPgcd.h` celle-ci

```
xpidl -m header -I $MOZILLA_SDK/idl nsIPgcd.idl
```

A la fin de cette étape, on se retrouve avec trois fichiers : `nsIPgcd.idl`, `nsIPgcd.xpt` et `nsIPgcd.h`. Lorsqu'on travaille avec XPCOM et qu'on ne s'intéresse pas à savoir comment

sont écrits les composants, on ne manipule que ces fichiers. Dans notre cas, ce n'est pas suffisant, on est celui qui doit fournir le service et non celui qui requiert, il reste l'implémentation à écrire, en commençant par l'en-tête C++.

### L'implémentation en C++

L'implémentation demande à écrire trois fichiers. Il est possible d'écrire le tout dans un seul fichier, si le code est très petit comme notre exemple. Cependant dans le cas d'un composant faisant plus d'une centaine de lignes il est vivement recommandé de séparer en trois fichiers distincts pour des raisons d'efficacité et de lisibilité. Pour prendre de bonnes habitudes, on va appliquer cette séparation dans notre exemple.

#### Fichier nsPgcd.h

A partir du fichier .h de l'interface généré par la commande xpidl, il est possible de récupérer le squelette de notre en-tête d'implémentation. A celui-ci il faut lui ajouter deux instructions *define*. Un pour la CLASS ID, un autre pour le CONTRACT ID. Le CONTRACTID est une chaîne "lisible" par l'humain, qui peut être ce qu'on veut mais différente de celles déjà existantes. La CLASS ID est ce qui identifie le composant, l'id doit être unique et différent de celui de l'interface.

```
#include "nsIPgcd.h"

// 0239a95e-93d1-4cef-b5f0-59dd4f0265af
#define NS_PGCD_CID \
{ 0x0239a95e, 0x93d1, 0x4cef, \
  { 0xb5, 0xf0, 0x59, 0xdd, 0x4f, 0x02, 0x65, 0xaf } }

#define NS_PGCD_CONTRACTID "@oxul.org/pgcd;1"

class nsPgcd : public nsIPgcd {
public:
  NS_DECL_ISUPPORTS
  NS_DECL_NSIPGCD

  nsPgcd();
private:
  virtual ~nsPgcd();
};
```

FIG. 1.11 – nsPgcd.h

On voit dans le code de la figure 1.11 la présence de deux macros : la première, NS\_DECL\_ISUPPORTS est commune à toutes les classes, elle définit les trois méthodes que doit posséder n'importe quelle classe XPCOM ; la deuxième, NS\_DECL\_NSIPGCD provient du fichier nsIPgcd.h, elle comprend le prototype de toutes les méthodes de l'interface nsIPgcd.

#### nsPgcd.cpp

Voici le code d'implémentation, qui fait tout le traitement.

La macro NS\_IMPL\_ISUPPORTS1(nsPgcd,nsIPgcd) ajoute à la classe nsPgcd l'implémentation des trois méthodes (obligatoires) de l'interface nsISupports, qui sont AddRef, Release et QueryInterface, en fonction des autres interfaces qu'elle implémente (ici, nsIPgcd).

```

#include "nsPgcd.h"

NS_IMPL_ISUPPORTS1(nsPgcd, nsIPgcd)
/* c++ constructor */
nsPgcd::nsPgcd()
{
    NS_INIT_ISUPPORTS();
}

/* c++ destructor */
nsPgcd::~nsPgcd(){}

/* calcul du pgcd */
PRInt32 pgcd(PRInt32 n, PRInt32 m){
    if(m==0)
        return n;
    else
        return pgcd(m,n-m);
}

/* long pgcd (in long n, in long m); */
NS_IMETHODIMP nsPgcd::Pgcd(PRInt32 n, PRInt32 m, PRInt32 *_retval)
{
    *_retval = pgcd(n,m);
    return NS_OK;
}

```

FIG. 1.12 – nsPgcd.cpp

Le chiffre 1 désigne le nombre d'interfaces qu'implémente une classe : s'il y en a plus d'un, on écrit le chiffre et les interfaces correspondant.

```
NS_IMPL_ISUPPORTSN(classe,interface1,...,interfaceN)
```

Pour le reste je pense que le code parle de lui-même, ce n'est que de la programmation C/C++. Néanmoins, la programmation XPCOM demande à éviter l'utilisation des exceptions (à cause des problèmes de portabilité paraît-il). Pour pallier à ce manque on utilise un autre moyen. On voit que la méthode *pgcd* retourne une valeur de succès (NS\_OK) à la fin : on fait les tests sur cette valeur pour savoir si l'opération s'est bien déroulée.

La lecture des méthodes XPCOM se fait de la manière suivante : elle retourne une valeur d'échec ou de succès ; elle a un nombre défini de paramètres ; si l'opération retourne une valeur on l'affecte au pointeur donné en paramètre. En somme, une méthode d'une classe XPCOM ne fait jamais un *return* sur un résultat de traitement.

Nous voilà avec le code de traitement écrit, il reste une chose à faire, c'est écrire le code d'enregistrement du module du composant.

### nsPgcdModule.cpp

Il est indispensable d'écrire le code d'enregistrement car sinon, notre composant ne sera pas chargé (ni déchargé à la fin). Un module peut n'avoir qu'un seul composant tout comme il peut contenir plusieurs. Par exemple le module *xul* (fichier *libxul.so* sous Linux) contient à lui tout seul tous les composants qui font le traitement des fichiers XUL et plus

encore. Dans l'exemple de la figure 1.13, dans le cadre de l'étude on fait simple, on crée un module pour notre unique composant.

```
#include <nsIGenericFactory.h>
#include "nsPgcd.h"

NS_GENERIC_FACTORY_CONSTRUCTOR(nsPgcd)

static NS_METHOD nsPgcdRegistrationProc(nsIComponentManager *aCompMgr,
                                        nsIFile *aPath,
                                        const char *registryLocation,
                                        const char *componentType,
                                        const nsModuleComponentInfo *info)
{
    return NS_OK;
}

static NS_METHOD nsPgcdUnregistrationProc(nsIComponentManager *aCompMgr,
                                          nsIFile *aPath,
                                          const char *registryLocation,
                                          const nsModuleComponentInfo *info)
{
    return NS_OK;
}

static nsModuleComponentInfo components[] =
{
    { "PGCD",          // a message to display when component is loaded
      NS_PGCD_CID,    // CLASS ID of type UUID
      NS_PGCD_CONTRACTID, // our human readable ID
      nsPgcdConstructor,
      nsPgcdRegistrationProc, /* NULL if you don't need one */
      nsPgcdUnregistrationProc /* NULL if you don't need one */
    }
};
NS_IMPL_NSGETMODULE(nsPgcdModule, components)
```

FIG. 1.13 – nsPgcdModule.cpp

Le code d'un module ne change pas énormément, on peut reprendre cet exemple en renommant juste les noms. Mais si on veut ajouter des informations supplémentaires, on peut le faire en ajoutant à la suite de *components* qui est un tableau. Ce tableau doit comporter au moins les quatre premiers éléments qui sont obligatoires.

Il ne reste plus qu'à compiler notre composant.

### Compilation

Avant tout il faut savoir que les nouvelles versions de Firefox ne sont pas compatibles avec les anciennes, l'API changeant souvent. Pour ne pas avoir de problèmes d'éditions de liens et compatibilité, il vaut mieux compiler les sources de Firefox et développer dessus. De plus, on récupère ainsi tous les headers des interfaces. Mozilla fournit le SDK mais celui-ci ne comprend que l'API gelé (frozen). Ce qu'ils entendent par gelé, ce sont toutes les interfaces qui ne seront plus modifiées peu importe les futures versions. Or, il y a beaucoup de fonctionnalités intéressantes dans l'API non gelé. C'est pourquoi si on travaille avec les interfaces *unfrozen*, il faut vérifier la compatibilité du code à chaque sortie de nouvelle version.

Pour notre exemple, l’API gelé suffit, on n’utilise que nsISupports. La figure 1.14 est un Makefile basique, utilisable par notre composant.

```

CC = gcc

CPPFLAGS += -fno-rtti -fno-exceptions -shared

# Change this to point at your SDK directory.
SDK = $(HOME)/mozilla/dist/sdk

# GCC only define which allows us to not have to #include mozilla-config
# in every .cpp file. If your not using GCC remove this line and add
# #include "mozilla-config.h" to each of your .cpp files.
CONFIG_INCLUDE = -include mozilla-config.h

DEFINES = -DXPCOM_GLUE

INCLUDES = -I $(SDK)/include -I $(SDK)/include/nspr

LDFLAGS = -L $(SDK)/lib -lxpcomglue -lnspr4 -lplds4 -lplc4

ICOM = nsIPgcd

COM = nsPgcd

FILES = $(COM).cpp $(COM)Module.cpp

TARGET = $(COM).so

build: public
    $(CC) -Wall -Os $(CONFIG_INCLUDE) $(DEFINES) \
    $(INCLUDES) -o $(TARGET) $(LDFLAGS) $(CPPFLAGS) $(FILES)
    chmod +x $(TARGET)
    strip $(TARGET)

public:
    $(SDK)/bin/xpidl -m header -I $(SDK)/idl $(ICOM).idl
    $(SDK)/bin/xpidl -m typelib -I $(SDK)/idl $(ICOM).idl

clean:
    rm $(TARGET)

```

FIG. 1.14 – Makefile

Quelques éclaircissements sur la figure 1.14 :

- `-DXPCOM_GLUE` : on précise au compilateur qu’on utilise la *glue*. Elle fournit un ensemble de fonctions et de macros très pratiques pour manipuler les classes, sans faire appel au pointeurs C (ils utilisent à la place les “smart pointers” comme ils les appellent).
- `-lxpcomglue` : les bibliothèques partagées `libxpcom` et `libxpcomglue` diffèrent au niveau des dépendances. La glue permet d’utiliser les classes et fonctions d’XPCOM sans avoir de dépendances sur les parties non gelé. C’est à dire que l’application ne cherchera pas à trouver la bibliothèque `xpcom_core` à l’exécution, ce sera géré par la glue. Mozilla préconise l’utilisation de XPCOMGlue pour les applications qui embarquent le SDK. Si on lie avec `libxpcom` on doit avoir l’adresse de la bibliothèque

dans la variable d'environnement PATH, sans oublier d'ajouter dans LDFlags les bibliothèques xul et js, ce qui donne :

```
LDFlags = -L $(SDK)/lib -lpcom -lxul -lmozjs -lnspr4 -lplds4 -lplc4
```

A la suite de cela, il reste à copier les deux fichiers nécessaires pour utiliser le composant fraîchement créé : le fichier d'interface typelib et la bibliothèque dynamique (.so sous Linux et .dll sous Windows) dans le répertoire *components* de l'application.

### Instanciation par Javascript

Cette partie ne concerne pas la création du composant en elle-même, mais il est intéressant de savoir comment, une fois notre composant créé, il est utilisé dans Firefox ou XULRunner.

L'utilisation et l'instanciation de notre objet se fait par Javascript. Dans un fichier XUL, seules les commandes Javascript sont prises en charge. Il y a plusieurs manières de créer un objet XPCOM en Javascript. Une d'entre elles est celle-ci :

```
var obj = Components.classes["@oxul.org/pgcd;1"].createInstance(Components.interfaces.nsIPgcd);
```

Une autre est celle-là :

```
var Obj = new Components.Constructor("@oxul.org/pgcd;1", "nsIPgcd");
var o = new Obj();
```

Ensuite il suffit d'appeler les méthodes de l'objet par la syntaxe pointé.

Maintenant que l'on a vu comment créer un composant simple, on a le nécessaire pour s'attaquer à l'interfaçage entre XPCOM et O'Caml. Mais avant cela, voyons ce qu'est ce XULRunner qu'on mentionne souvent, et comment on peut lancer une application avec Xulrunner.

## 1.3 XULRunner

### 1.3.1 Introduction

La première comparaison que l'on peut faire avec XULRunner, c'est que XULRunner est un Firefox sans l'interface graphique. Il lit les fichiers XUL, et est composé de composants XPCOM. XULRunner est un lanceur d'applications utilisant les technologies Mozilla. Les versions de XULRunner suivent l'évolution de la numérotation du moteur Gecko. La version utilisée pour notre travail est la *1.8.0.1* et est compatible avec Firefox 1.5. A terme, dans la version 1.9 de Gecko, et donc de XULRunner, Firefox se basera sur XULRunner pour être lancé.

### 1.3.2 Structure d'une application

Toutes les applications XULRunner ont la même arborescence générale. Pour qu'elles puissent être lancés, elles doivent impérativement avoir au moins les fichiers et répertoires indispensables. Il est possible d'ajouter des sous-répertoires en plus pour mieux organiser

l'application, mais il ne faut pas modifier les noms de répertoires par défaut comme *chrome* ou *content* entre autres.

```

monApplication
- application.ini
- chrome/
  - content/
    - monApplication.xul
  - locale/
    - fr-FR/
      - monApplication.dtd
  - icons/
- chrome.manifest
- components/
  - monIComposant.xpt
  - monComposant.so /* pour Linux */
  - monComposant.dll /* pour Windows */
- defaults/
  - preferences/
    - monApplication.js

```

FIG. 1.15 – Arborescence d'une application XULRunner

Une application XULRunner est lancé avec la commande *xulrunner* de la manière suivante :

```
xulrunner application.ini
```

Le fichier *application.ini* contient les informations de l'application. Il ressemble à ceci.

```

[App]
; Nom de l'auteur, de l'éditeur...
Vendor=oxul

; nom de l'application
Name=oxul

; version de l'application
Version=0.1

; un numero de version interne, il contient en général un timestamp
BuildID=20060305

; le copyright
Copyright=Copyright (c) 2006 Oxul

; uuid de l'application. exemple :
ID={8a5269f3-b5fc-4b40-a2d2-d7f6c70a4011}

[Gecko]
; version minimum de Gecko prise en charge
MinVersion=1.8
; version maximum
MaxVersion=1.9

```

Maintenant que l'introduction des technologies Mozilla est faite, passons à la conception de l'interprète, puis à l'interfaçage.

## Chapitre 2

# Intégration de XUL dans O'Caml

### 2.1 Etat de l'art

#### 2.1.1 Introduction

XUL étant une technologie toute nouvelle, il n'existe quasiment aucun projet identique au notre. Nous avons trouvé seulement deux projets similaires : *jXUL* et *Luxor*, les deux étant des intégrations de XUL dans Java.

*jXUL* [1] est un interprète XUL pour Java compatible à 100% avec le standard Mozilla XUL, créant une interface à l'aide de Java Swing. Malheureusement, ce projet n'est que très peu avancé, et sa page sur SourceForge indique qu'il n'a pas été mis à jour depuis le 14 octobre 2001. Or, depuis cette date, XUL et toutes les technologies Mozilla (XPCOM, Gecko, *etc.*) ont évolués considérablement, et par conséquent, le projet *jXUL* est bien trop obsolète pour nous apporter une quelconque aide.

En revanche, *Luxor* [2] est un projet très actif, intégrant un serveur web, et permettant la gestion de *plugins*. Il se découpe en plusieurs modules : *Luxor* qui constitue le noyau, *Luxilla* qui se veut l'équivalent de XulRunner mais en Java, et *Petra* qui est le système de gestion de plugins. Il ne respecte pas à 100% le standard Mozilla XUL dans la mesure où *Luxor* ne supporte pas le JavaScript, lui préférant le Java et le Python en tant que langage de script.

Le choix de l'intégration de XUL dans O'Caml s'est porté sur l'écriture d'un interprète XUL permettant de créer à la volée une fenêtre graphique à l'aide de code O'Caml, comme si elle avait été écrite dans ce langage. Cependant, comme nous l'avons vu en introduction, XUL utilise à profit certains standards tels que JavaScript, CSS et DTD. Ainsi, l'écriture de cet interprète ne se limitait pas à analyser syntaxiquement un fichier XUL, mais impliquait également la prise en charge de tous ces standards.

#### 2.1.2 Choix de la bibliothèque graphique

L'un des points-clé dans l'écriture de cet interprète était le choix de la bibliothèque graphique à utiliser pour la création des fenêtres. Deux choix s'offrirent à nous : *LablTK* et *LablGTK2*.

La bibliothèque *LablTK* s'appuie sur le toolkit TCL/TK. Elle a le gros avantage d'être fournie par défaut avec le compilateur O'Caml. Aucune installation n'est donc nécessaire, tout est déjà intégré au compilateur, et son utilisation est, par conséquent, immédiate.

La bibliothèque *LablGTK2*, en revanche, n'est pas fournie par défaut, et son utilisation nécessite une installation préalable. Cette bibliothèque s'appuie sur le toolkit GTK2.

Repenons notre exemple 1.2 permettant de calculer le PGCD de deux entiers, et adaptons-le en code O'Caml écrit pour LablTK et pour LablGTK2. Le code-source 2.1 montre une implémentation possible avec LablTK, et la figure 2.2 nous montre son exécution. De même, le code-source 2.3 présente un exemple utilisant LablGTK2, et la figure 2.4 nous montre son exécution.

FIG. 2.1 – Fichier tk.ml

```

open Tk;;

let rec pgcd n1 m1 =
  match m with
  | 0 -> n
  | _ -> pgcd m (n mod m);;

let calculer entry_n entry_m label_r root () =
  try
    let n = int_of_string (Entry.get entry_n) in
    let m = int_of_string (Entry.get entry_m) in
    let res = pgcd n m in
    Label.configure label_r ~text:(string_of_int res)
  with
  _ -> failwith "Impossible de calculer le PGCD !";;

let _ =
  (* création des éléments graphiques *)
  let root = Tk.openTk () in
  let top_frame = Frame.create root in
  let gcd_begin = Label.create top_frame ~text:"pgcd ( " in
  let n = Entry.create top_frame ~width:4 ~relief:'Sunken in
  let m = Entry.create top_frame ~width:4 ~relief:'Sunken in
  let gcd_end = Label.create top_frame ~text:") = " in
  let result = Label.create top_frame in
  let button = Button.create root ~text:"Compute"
    ~command:(calculer n m result root) in
  let quit = Button.create root ~text:"Quit" ~command:(Tk.closeTk) in

  (* placement des éléments dans la fenêtre *)
  pack [top_frame] ~side:'Top ~expand:true ~fill:'Both ;
  pack [gcd_begin] ~side:'Left ~expand:true ~anchor:'Center ;
  pack [n;m] ~side:'Left ~expand:true ~fill:'X ;
  pack [gcd_end;result] ~side:'Left ~anchor:'Center ;
  pack [quit;button] ~side:'Bottom ~expand:true ~fill:'X ;
  mainLoop ();;

```



FIG. 2.2 – Exécution de tk.ml

FIG. 2.3 – Fichier gtk2.ml

```

let rec pgcd n2 m2 =
  match m with
  | 0 -> n
  | _ -> pgcd m (n mod m);;

let calculer entry_n entry_m label_r () =
  try
    let n = int_of_string (entry_n#text) in
    let m = int_of_string (entry_m#text) in
    let res = pgcd n m in
    label_r#set_text (string_of_int res)
  with
  _ -> failwith "Impossible de calculer le PGCD !";;

let _ =
  (* création des éléments graphiques *)
  let root = GWindow.window () in
  let main_frame = GPack.box 'VERTICAL ~packing:root#add () in
  let top_frame = GPack.hbox ~packing:main_frame#add () in
  let gcd_begin = GMisc.label ~text:"pdcg ( " () in
  let n = GEdit.entry ~width:40 () in
  let m = GEdit.entry ~width:40 () in
  let gcd_end = GMisc.label ~text:" ) = " () in
  let result = GMisc.label () in
  let button = GButton.button ~label:"Compute" ~packing:main_frame#add ()
  in
  let quit = GButton.button ~label:"Quit" ~packing:main_frame#add () in

  (* callbacks *)
  ignore (root#connect#destroy ~callback:GMain.Main.quit);
  ignore (button#connect#clicked ~callback:(calculer n m result));
  ignore (quit#connect#clicked ~callback:GMain.Main.quit);

  (* placement des éléments dans la fenêtre *)
  top_frame#pack ~expand:true ~fill:false gcd_begin#coerce;
  top_frame#pack ~expand:true ~fill:false n#coerce;
  top_frame#pack ~expand:true ~fill:false m#coerce;
  top_frame#pack ~expand:true ~fill:false gcd_end#coerce;
  top_frame#pack ~expand:true ~fill:false result#coerce;
  root#show ();

  GMain.Main.main ();;

```

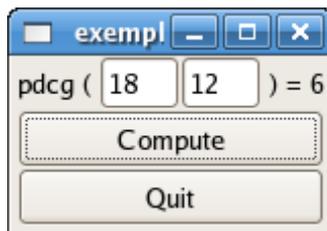


FIG. 2.4 – Exécution de gtk2.ml

Comme vous pouvez le constater, d'un point de vue esthétique, l'interface de GTK2 est bien plus attrayante que celle de TK qui paraît plus *archaïque*. Ceci est dû au fait que GTK2 supporte la gestion de thèmes. D'un point de vue programmation, on remarque que GTK2 est plus évolué que TK avec un système d'enregistrement de callbacks séparé, des noms de méthodes plus intuitifs (comme la méthode `set_text` permettant de changer le texte d'un label, contrairement à la méthode `Label.configure` de LablTK), et d'un système de positionnement des composants graphiques plus évolué analogue au système de positionnement de la classe `Box` de Java.

Finalement, notre choix s'est orienté vers LablGTK2, dans la mesure où XUL offre des widgets que TK ne propose pas, contrairement à GTK2, comme par exemple les barres d'outils flottantes. Par ailleurs, XulRunner utilise lui-même GTK2, ce qui finalement confirme que le choix de LablGTK2 est le plus judicieux.

## 2.2 Interprète Xul en O'Caml

### 2.2.1 Analyseur syntaxique pour XUL : XML-Light

Comme nous l'avons vu en introduction, XUL est un langage dont la syntaxe est basé sur XML. Par conséquent, pour analyser syntaxiquement un fichier XUL, un simple parseur XML suffisait. Notre choix s'est porté sur XML-Light[3] qui a l'avantage d'être très léger et simple d'utilisation.

Par exemple, pour parser un fichier XUL, il suffit de l'ouvrir avec la fonction `parse_file` afin de créer l'arbre correspondant en mémoire, comme ceci :

```
let x = Xml.parse_file "fichier.xul" in
```

Puis, pour parcourir chaque noeud de l'arbre XUL, il suffit d'utiliser la fonction `iter` qui permet d'appliquer une fonction à chaque noeud de même niveau dans l'arbre. Pour atteindre les sous-noeuds, un appel récursif à `iter` suffit.

Ensuite, pour chaque noeud, il faut récupérer son nom et ses attributs afin de décider de l'action à effectuer. Par exemple, si le noeud correspond à un bouton, il faut créer dynamiquement ce bouton en le paramétrant à l'aide des valeurs des attributs du noeud correspondant. Pour récupérer le nom d'un noeud, il faut utiliser la fonction `tag` qui prend un noeud en argument et renvoie son nom sous forme de chaîne de caractères. Et pour récupérer ses attributs, il faut utiliser la fonction `attrib` qui prend en argument le noeud et une chaîne correspondant au nom de l'attribut à récupérer, et en renvoie la valeur sous forme de chaîne.

Tout ceci permet de facilement analyser un fichier XUL sans aucune restriction.

### 2.2.2 Interprétation du code JavaScript : SpiderCaml

Pour gérer le code JavaScript, il nous fallait un interpréteur JavaScript pour O'Caml. Nous avons trouvé la librairie SpiderCaml[4] qui réutilise l'interprète SpiderMonkey initialement développé pour le navigateur Netscape, et aujourd'hui utilisé dans les technologies Mozilla.

La plus grosse difficulté est de définir tout le DOM (*Document Object Model*) propre à XUL. En effet, lorsqu'un fichier XUL est chargé, tous les noeuds sont stockés en mémoire sous forme d'arbre, chaque noeud de l'arbre correspondant à une balise du document XUL. Cet arbre s'appelle le DOM. Il existe alors tout un jeu d'objets et de méthodes permettant d'accéder et modifier cet arbre. Nous avons déjà rencontré le DOM dans notre exemple

1.2 de calcul de PGCD, avec les commandes JavaScripts `self.close()` et `document.getElementById(...)`.

Pour utiliser l'interprète SpiderCaml, il faut d'abord créer un objet principal qui correspondra à la racine de l'arbre du DOM. Cet objet est une instance de la classe `SpiderCaml.jsobj`. Pour se faire, il suffit d'utiliser la fonction `new_global_obj` comme ceci :

```
let glb = new_global_obj () in
```

Ensuite, pour évaluer une commande JavaScript, il suffit d'utiliser la méthode `eval` de cet objet principal précédemment créé. Voici comment évaluer le script `"var a = 15;"` en affichant son évaluation :

```
print_endline ((glb # eval "var a = 15;") # to_string)
```

Une autre difficulté est de définir une fonction. SpiderCaml permet de substituer une fonction apparaissant dans un script JavaScript par du code O'Caml. Par exemple, pour définir une fonction `print` qui permet d'afficher un message passé en argument, voici comment procéder :

```
let glb = new_global_obj () in
glb # set "print"
  (glb # lambda (fun _ args ->
                    print_endline (args.(0) # to_string);
                    glb # null))
```

Enfin, la dernière difficulté consiste à définir les objets et leurs méthodes. SpiderCaml permet d'effectuer ceci de manière relativement simple. Par exemple, pour définir l'objet `self` et sa méthode `close`, voici une manière de faire :

```
let glb = new_global_obj () in
let obj_self = glb # new_object () in
glb # set "self" obj_self;
obj_self # set "close"
  (obj_self # lambda (fun _ _ ->
                    GMain.Main.quit ();
                    js_obj_self # null))
```

L'objet créé est également du type `SpiderCaml.jsobj`.

Ceci devrait nous permettre de définir tous les éléments par défaut du DOM à l'exécution.

En clair, il est utile de noter que toutes les instructions JavaScript natives à SpiderCaml sont effectuées en C par la librairie SpiderMonkey. Seules les fonctions JavaScript définies manuellement à l'aide de la méthode `lambda` sont exécutées en O'Caml.

Finalement, la figure 2.5 nous montre comment un code XUL est interprété grâce à la relation entre XUL, le code JavaScript et le code O'Caml à l'aide des bibliothèques XML-Light et SpiderCaml.

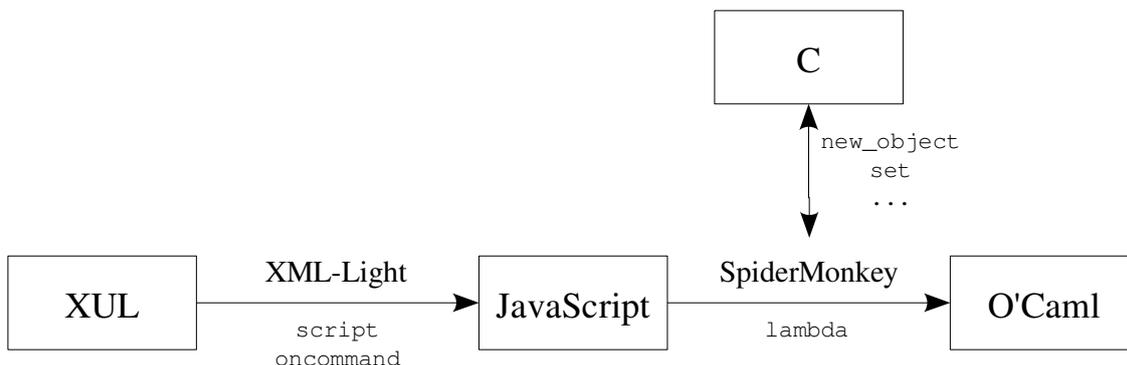


FIG. 2.5 – Interprétation du code XUL et du code JavaScript en O'Caml

### 2.2.3 Analyseurs syntaxiques pour CSS et DTD

Nous avons vu que XUL permet de tirer profit des feuilles de style CSS pour modifier l'aspect graphique des éléments d'une interface, ainsi que des fichiers DTD pour localiser le texte d'une application. Il était donc nécessaire d'avoir des analyseurs syntaxiques pour ces deux technologies.

Pour le langage DTD, la librairie XML-Light offre tous les outils d'analyse syntaxique pour l'écriture de notre interprète. En revanche, pour le langage CSS, nous n'avons trouvé aucun parseur pour O'Caml. Il était donc nécessaire de l'écrire nous-même.

Malheureusement, nous n'avons pas pu gérer ces deux technologies dans notre interprète par manque de temps. Un parseur/lexeur CSS a commencé à être écrit, mais n'a pas pu être achevé.

## 2.3 Appel de code O'Caml en JavaScript

### 2.3.1 Étude de faisabilité

Le but principal de notre interprète était de permettre d'effectuer des calculs en O'Caml. Comme nous l'avons vu à la section 2.2.2, SpiderCaml n'est qu'une « surcouche » de SpiderMonkey, et par conséquent, toute fonction ou objet défini dans un code JavaScript interprété par SpiderCaml est traité en C au niveau interne. Par conséquent, il était nécessaire de définir une technique permettant d'effectuer un calcul 100% en Caml.

Nous nous sommes orienté vers l'écriture d'un nouvel objet JavaScript appelé « OCaml » offrant plusieurs méthodes d'évaluation de code O'Caml telles que `eval`, `evalInt`, ... L'évaluation d'une phrase O'Caml est permise grâce à l'embarquement d'un `oplevel` au sein même de notre interprète à l'aide du module `Toploop`, et s'effectue à l'aide de la fonction `exec` suivante :

```

open Toploop ;;
...
let exec s =
  let ast = !parse_toplevel_phrase (Lexing.from_string s) in
  ignore (execute_phrase true (Format.str_formatter) ast);
  Format.flush_str_formatter ()
;;

```

Cette fonction prend une chaîne de caractère représentant le code O’Caml à évaluer, et retourne une chaîne représentant la sortie du toplevel. Par exemple, le code :

```
exec "1 + 2 ;;"
```

retourne la chaîne de caractères “- : int = 3”.

### 2.3.2 Récupération de valeurs O’Caml

L’évaluation d’une phrase O’Caml n’est pas suffisant. Car notre but est d’être capable de récupérer la valeur d’un calcul effectué dans le toplevel pour pouvoir ensuite créer l’objet `SpiderCaml.jsobj` correspondant, afin que cette valeur puisse être réutilisée en JavaScript par `SpiderCaml`.

Le module `Toploop` offre la fonction `getvalue` permettant de récupérer une valeur définie dans l’environnement global du toplevel. Cependant, la valeur renvoyée par cette fonction est du type `Obj.t` et n’est pas manipulable directement.

Objective Caml fournit un module appelé `Obj` permettant d’effectuer des instructions de bas niveau sur des valeurs, à la manière des fonctions C assurant l’interfaçage entre C et O’Caml. Ce module permet notamment des *cast* à la C grâce à sa fonction `magic` permettant de *caster* un type  $\alpha$  en un type  $\beta$ . Ainsi, il est tout à fait possible de « casser » le typage, par exemple en castant un `int` en une `string`. Par conséquent, l’utilisation de ce module est à prendre avec beaucoup de précautions.

Ce module a l’avantage d’offrir un type générique `Obj.t` analogue au pointeur `void*` en C, et qui est l’équivalent du type `value` offert par la librairie O’Caml pour le C pour l’interopérabilité C/Caml. De plus, ce module propose diverses fonctions permettant de manipuler ce type. Par exemple, la fonction `repr` permet d’obtenir le type `Obj.t` représentant une valeur  $\alpha$ , et réciproquement, la fonction `obj` permet d’obtenir un type  $\alpha$  à partir du type `Obj.t`.

Ainsi, grâce au module `Obj`, il est possible de récupérer une valeur définie dans le toplevel, en voici un exemple :

```
ignore (exec "let a = 5;;");
print_int (Obj.obj (getvalue "a") : int);
print_newline ();
```

Ce code évalue une valeur `a` dans le toplevel, puis la récupère sous forme d’une valeur de type `Obj.t` à l’aide de `getvalue`, puis la transforme en `int` à l’aide de la fonction `obj`, et l’affiche. La sortie de ce code est donc “5”.

Cependant, le code ci-dessus suppose de connaître à l’avance le type de l’expression calculée. Mais comment savoir si le type `Obj.t` récupéré est un entier, une chaîne, une liste, ... ? Il est nécessaire de faire une analyse de type en parcourant le format interne de la valeur, à la manière de la fonction `affiche_bloc` du DA-OCAML [5]. Ici, le module `Obj` permet d’effectuer les mêmes opérations qu’en C. Par exemple, la fonction `Obj.is_int` en O’Caml est identique à la fonction `Is_long` en C.

### 2.3.3 Conversion de valeurs O’Caml vers JavaScript

Une fois l’analyse de type effectuée et le type récupéré, la classe `SpiderCaml.jsobj` offre des méthodes de création d’objets JavaScript, telles que `string`, `int`, `array`, ...

Voici la fonction `jsobj_of_ocamlvalue` qui prend en argument un objet `SpiderCaml.jsobj` pour avoir accès aux méthodes de création d’objets JavaScript, et une valeur de type `Obj.t`, et qui renvoie un objet `JavaScript.jsobj` correspondant à la valeur :

```

let jsobj_of_ocamlvalue js_obj value =
  (* 'liste' sert ici pour retenir les pointeurs copiés dans les
   * tableaux de valeurs, pour éviter les récursions infinies dans le
   * cas de valeurs cycliques *)
  let liste = ref [] in
  let rec aux js v =
    if (Obj.is_int v) then
      js # int (Obj.obj v)
    else
      begin
        let taille = Obj.size v
        and tag = Obj.tag v in
        if (tag == Obj.closure_tag) then
          begin
            failwith "Fermeture : type non supporté"
          end
        else if (tag == Obj.string_tag) then
          js # string (Obj.obj v)
        else if (tag == Obj.double_tag) then
          js # float (Obj.obj v)
        else if (tag == Obj.double_array_tag) then
          begin
            (* On contruit un tableau d'objets JS vides. Ces objets
             * deviendront ensuite des flottants (cf. ci-après). *)
            let float_array = Array.make taille (js # null) in
            for i = 0 to (taille - 1) do
              Array.set float_array i
                (js # float (Obj.obj (Obj.field v i)))
            done;
            js # array float_array
          end
        else if (tag == Obj.abstract_tag) then
          failwith "Type abstrait : non supporté"
        else
          begin
            if (tag >= Obj.no_scan_tag) then
              failwith "Type inconnu"
            else
              begin
                let value_array = Array.make taille (js # null) in
                begin
                  for i = 0 to (taille - 1) do
                    let field = Obj.field v i in
                    (* si le i-ième champ est une valeur immédiate,
                     * alors il ne faut pas l'ajouter à la liste de
                     * pointeurs. *)
                    if (Obj.is_int field) then
                      Array.set value_array i (aux js field)
                    else
                      begin
                        let adresse_bloc : int = Obj.obj field in
                        try
                          ignore
                            (List.find
                              (fun p ->
                                let b = ((fst p) == adresse_bloc) in
                                if b then
                                  Array.set value_array i (snd p);
                                  b)
                              !liste);
                        end
                      end
                end
              end
            end
          end
        end
      end
  end
  aux js_obj value

```

```

                with
                    Not_found ->
                        let js_o = aux js field in
                            Array.set value_array i js_o;
                            liste := (adresse_bloc, js_o) :: !liste
                        end
                    done
                end;
                js # array value_array
            end
        end
    end
in
    aux js_obj value
;;

```

Pour la manipulation de tableaux de valeurs, il peut être utile de sauvegarder l’adresse d’un bloc ajouté dans le tableau dans une liste de couples (`adresse`, `jsobj`), et de vérifier la présence d’une adresse dans cette liste avant d’effectuer un appel récursif. Car dans le cas de valeurs avec cycle, le cycle produirait une récursion infinie, recréant infiniment les mêmes objets en mémoire, et produisant un erreur de type “*stack overflow*”. En revanche, il ne faut pas ajouter de valeurs immédiates à cette liste, car si la fonction `Obj.obj` appliqué à un bloc renvoie son adresse, appliqué à une valeur immédiate, elle renvoie la valeur de celle-ci. Ainsi, dans le cas de listes — ou tout autre type ayant une représentation interne sous forme de tableau de valeurs — il suffirait que cette liste contienne deux entiers ou booléens de même valeur pour que la construction du tableau de valeur s’arrête à la seconde occurrence de cette valeur immédiate.

### Restrictions de l’analyse de type

L’analyse de type ne permet pas toujours de récupérer le vrai type d’une valeur. En effet, certains types ont une représentation interne similaire à d’autres. C’est le cas notamment du type `bool` qui, en mémoire, est une valeur immédiate, tout comme le type `int`. Ainsi, que l’on soit en présence d’un type `bool` ou `int`, la fonction `Obj.is_int` renvoie toujours `true`, et il est impossible de savoir s’il s’agit d’un entier ou d’un booléen. De même pour les types `list`, `array` et enregistrements qui ont une représentation interne de tableaux de valeurs (une valeur pouvant être un tableau), et sont donc considérés comme des `array` par l’analyse interne du type.

Ainsi, l’analyse de type renvoie une valeur telle qu’elle est représentée en mémoire, et non pas selon son vrai type Caml. Par exemple, dans le cas d’une liste `l` telle que :

```

let l1 = [1; 2; 3];;
let l2 = [4; 5; 6];;
let l  = [l1; l2; l1];;

```

sa représentation en mémoire est (en supposant que `[a;b]` est un tableau contenant deux éléments `a` et `b`) :

```

[[1; [2; [3; NULL]]]; [[4; [5; [6; NULL]]]; [[1; [2; [3; NULL]]]; NULL]]

```

Ainsi, un appel à `jsobj_of_ocamlvalue` sur la liste `l` ci-dessus renvoie un objet JavaScript égal au tableau ci-dessus, les valeurs `NULL` étant égales à 0. Il s’agit donc d’un format difficilement réutilisable ensuite en JavaScript.

### 2.3.4 Interprétation de code O’Caml en JavaScript

Finalement, nous avons décidé de la convention suivante pour l’appel de code O’Caml en JavaScript :

- `OCaml.eval("code_ocaml")` : évalue la *phrase* O’Caml `"code_ocaml"`, et renvoie la chaîne de sortie du toplevel
- `OCaml.eval("code_ocaml", "type_attendu")` : évalue l’*expression* O’Caml `"code_ocaml"` de type `type_attendu`, et renvoie l’objet JavaScript correspondant à la représentation interne de l’expression calculée
- `OCaml.evalFile("fichier.ml")` : évalue le contenu du fichier `fichier.ml`, et renvoie la chaîne de sortie du toplevel

Pour la seconde méthode `eval`, la phrase à évaluer doit être une expression unique, sans définition globale avec `let`, car pour récupérer la valeur de cette expression, nous devons l’enregistrer dans l’environnement global du toplevel, pour récupérer la valeur à l’aide de `Toploop.getvalue` (cf. section 2.3.2, page 25). Pour se faire, nous concaténons la chaîne `"let ___expr___ : type_attendu ="` à l’expression. Par exemple, un appel à :

```
OCaml.eval("1 + 2 ;;", "int");
```

produit la phrase :

```
let ___expr___ : int = 1 + 2 ;;
```

Ainsi, tout autre code dans l’expression à évaluer peut produire une chaîne syntaxiquement incorrecte, et provoquer une exception du toplevel.

De plus, il est important de préciser le type de retour sous forme de chaîne, afin de le concaténer à l’expression d’évaluation définissant la valeur `___expr___`. En effet, ceci permet de forcer le typage au sein même du toplevel, et d’éviter tout cast erroné. Car comme nous l’avons vu à la section 2.3.2, le module `Obj` permet de facilement casser le typage de Caml. Ainsi, sans cette concaténation, il serait tout à fait possible d’écrire :

```
OCaml.eval("1 + 2 ;;", "string");
```

ceci produisant la phrase :

```
let ___expr___ = 1 + 2 ;;
```

L’entier retourné (ici égal à 3) sera alors casté en une chaîne, sans obtenir la moindre erreur, et produisant un résultat tout à fait inattendu.

Avec la concaténation, le code JavaScript ci-dessus produit la phrase :

```
let ___expr___ : string = 1 + 2 ;;
```

qui est erronée puisque l’on attend un type `string` alors que l’expression située derrière est de type `int`. Une exception est alors levée par le toplevel.

#### Mise en pratique

Finalement, réécrivons notre exemple 1.2 de calcul de PGCD avec cette fois-ci, la fonction `pgcd` définie en O’Caml. Le code se compose de trois fichiers : `pgcd_ocaml.xul`, `pgcd_ocaml.js` et `pgcd_ocaml.ml`.

FIG. 2.6 – Fichier pgcd\_ocaml.xul

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<window
  id="root"
  title="Xul PGCD"
  xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

  <script src="pgcd_ocaml.js"/>

  <vbox>
    <hbox>
      <label value="pgcd ( " />
      <textbox id="n"/>
      <textbox id="m"/>
      <label value=" ) = " />
      <label id="result" value="" />
    </hbox>
    <button id="button" label="Compute" onclick="calculer();"/>
    <button id="quit" label="Quit" onclick="self.close();"/>
  </vbox>

</window>

```

FIG. 2.7 – Fichier pgcd\_ocaml.js

```

function calculer() {
  try {
    var n = document.getElementById('n').value;
    var m = document.getElementById('m').value;

    OCaml.evalFile("pgcd_ocaml.ml");
    var res = OCaml.eval("pgcd " + n + " " + m + ";;", "int");

    document.getElementById('result').value = res;
  }
  catch(error) {
    alert("Erreur :\n\n" + error);
  }
}

```

FIG. 2.8 – Fichier pgcd\_ocaml.ml

```

let rec pgcd n m =
  match m with
  | 0 -> n
  | _ -> pgcd m (n mod m)
;;

```

ATTENTION, il est important de noter que le code O'Caml est interprété par un toplevel. Ceci implique une syntaxe de code nécessaire à une bonne exécution par un toplevel, à savoir finir chaque définition de valeurs par ”;;”, etc.

## Chapitre 3

# Interfaçage entre XPCOM et O'Caml

Le but de cette partie est d'embarquer du code O'Caml dans Mozilla Firefox ou toute autre application utilisant la technologie XPCOM, et inversement.

### 3.1 Etude de faisabilité

#### 3.1.1 Objective Caml

Dans cette partie du projet, je me base sur le côté interfaçage entre le C et O'Caml[?]. Le chapitre 12 du livre[5] détaille les deux sens d'appel de fonctions. D'une part, comment du code C appelle du code O'Caml, d'autre part, comment le code O'Caml appelle du code C.

Procédons par étape et commençons par le plus facile des deux, l'appel de fonction O'Caml à partir d'un XPCOM.

#### 3.1.2 Appel de fonction O'Caml dans un composant XPCOM

Intégrer du code Caml dans un XPCOM consiste à utiliser trois voire quatre langages de programmation différents. Comme il est dit dans l'introduction de XPCOM, le langage natif des composants est le C++. Pour appeler les fonctions O'Caml à partir des méthodes des classes C++, il va falloir utiliser des fonctions *passerelles*. Le C++ est une surcouche objet de C et il est possible d'écrire du code utilisable par les deux langages. On sait aussi qu'O'Caml est interopérable avec le C. Je vais expliquer le principe en prenant l'exemple d'une fonction simple O'Caml que j'intègre à un XPCOM, puis pour tester la communication des appels réflexes entre C++ et O'Caml j'essaye d'embarquer le toplevel O'Caml[6]. Les tests se font avec Xulrunner pour lancer l'application.

#### Un simple appel à une fonction O'Caml

Bruler les étapes n'est jamais bon, dans ce but il faut commencer par savoir si du code C++ de XPCOM peut faire un appel à une opération O'Caml. On va prendre l'exemple du pgcd pour rester sur la même voie. Cette étape permettra de trouver les commandes de base d'une compilation entre C++ et O'Caml. Procédons par étape.

**Première étape** : faire en sorte que du code C puisse appeler le code O'Caml. Pour se faire, il faut voir le problème dans l'autre sens : faire en sorte que le code O'Caml

permette au code C de l'appeler. A l'aide du module *Callback* et de sa fonction *register*, il est possible d'enregistrer une fonction O'Caml et de l'associer à un nom. Lorsqu'une fonction C appellera le nom associé avec la fonction *callback*, il exécutera le code O'Caml.

Soit notre fonction *pgcd* en O'Caml :

```
let rec pgcd n m =
  match n,m with
  | n,0 -> n
  | n,m -> pgcd n (n mod m);;

Callback.register 'pgcd' pgcd;
```

La compilation du fichier O'Caml doit se faire avec l'option *-output-obj*, ceci a pour but de créer un fichier objet C que l'on donnera en argument à la compilation du fichier C ou C++.

```
ocamlc -output-obj unix.cma pgcd.ml -o pgcdcaml.o
```

Notre fonction *pgcd* peut être maintenant utilisée en C.

Il y a deux manières d'écrire la suite : soit on sépare l'écriture du C et C++, soit on met tout dans le fichier C++. Si on met tout dans un seul fichier il faudra déclarer la fonction de callback en *extern "C"*. J'ai opté pour la séparation, comme cela on ne touche pas trop au code du composant (juste l'ajout d'un include et appel des fonctions).

**Deuxième étape** : écriture du code C/C++. Comme on sépare les fichiers, il faut écrire un en-tête qui soit partagé par C et C++. Il ressemble à cela :

```
#ifdef __cplusplus
extern "C" {
#endif

  int pgcd(int n, int m);

#ifdef __cplusplus
}
#endif
```

Le composant qu'on va écrire pourra utiliser la fonction sans se soucier de comment C l'écrit. Pour la création du XPCOM, on se réfère à la section 1.2.2. Il faut juste ajouter au début

```
#include "pgcd.h"
```

Quant au code C, c'est simple, la fonction *pgcd* est la suivante :

```
int pgcd(int n,int m){
  value vn,vm,res;
  vn = Val_int(n);
  vm = Val_int(m);
  res = callback2(*caml_named_value('pgcd'),vn,vm);
  return Int_val(res);
}
```

N'oublions pas que pour cet exemple, à la compilation du XPCOM, il faut lier avec la bibliothèque mathématique.

Le schéma 3.1 donne une vue globale de la répartition des fichiers, la syntaxe C/O'Caml n'est pas respectée faute de place. La deuxième partie de la figure représente un appel partant de XPCOM, qui est le langage déclencheur.

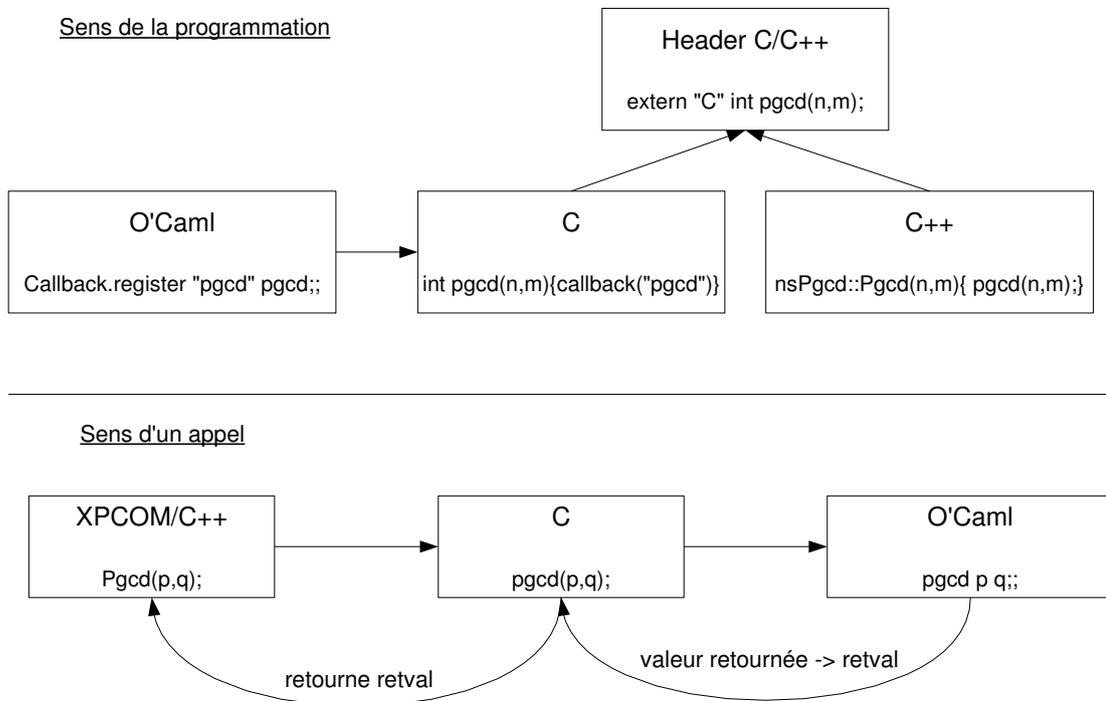


FIG. 3.1 – Schéma des appels

Une fois que le module du composant ainsi que son interface sont compilés, on doit le tester. On crée pour cela une application tout ce qu'il y a de plus simple, que l'on lance avec XULRunner. Si le calcul renvoie bien le résultat escompté, alors le composant fonctionne. Plus exactement, si le composant est enregistré, il ne devrait pas poser de problèmes. En effet, s'il est mal codé, XULRunner (ou Firefox) ne l'enregistrera pas. Quand on voudra faire un appel à la méthode de la classe, Javascript renverra une erreur du type :

```
TypeError : Components.classes["@oxul.org/pgcd;1"] has no properties
```

Maintenant qu'on sait qu'il est possible pour XPCOM de faire appel à du code O'Caml, il faut se convaincre avec un code plus pertinent. Pour cela, on décide d'embarquer le toplevel O'Caml, histoire de pouvoir coder en O'Caml dans Firefox ou XULRunner.

### Un exemple plus compliqué : toplevel O'Caml embarqué

Le toplevel O'Caml est une boucle qui tourne indéfiniment. Elle prend une phrase O'Caml et retourne le résultat de l'évaluation en enrichissant l'environnement s'il le faut.

Embarquer le toplevel consiste à ajouter la fonction qui évalue une phrase O'Caml. Le reste de la procédure est similaire à l'exemple du pgcd. La fonction de la figure 3.2 est

celle qui va évaluer la chaîne de caractères donnée en paramètre. Elle retourne le résultat de l'évaluation sur le buffer des chaînes de caractères.

```
open Toploop;;
open Callback;;

let exec s =
  let ast = !parse_toplevel_phrase (Lexing.from_string s) in
  ignore(execute_phrase true (Format.str_formatter) ast);
  Format.flush_str_formatter();;

Callback.register "exec" exec;;
```

FIG. 3.2 – Fonction d'évaluation de phrase O'Caml

La compilation des fichiers *.ml* et *.cpp* demande un minimum de connaissances en compilation O'Caml. En effet, il ne faut pas oublier de donner les options *-output-obj toplevel-lib.cma unix.cma* au compilateur *ocamlc*. La commande est donc :

```
ocamlc -output-obj toplevellib.cma unix.cma -o camltoplevel.o camltoplevel.ml
```

De même, pour la compilation C/C++ il faut savoir lier avec les bibliothèques *camlrn*, *curses*, *unix* et *mathématique*.

```
-L $(CAML) -lunix -lcamlrun -lcurses -lm
```

où *\$(CAML)* est le répertoire où est installé Objective Caml.

Le toplevel O'Caml doit être initialisé pour pouvoir être utilisé, ceci est faisable par appel à la fonction *caml\_startup*; il n'a besoin de démarrer qu'une seule fois tout le temps que dure l'application. Mettre un *flag* à l'aide du mot-clé *static* fait l'affaire. Après initialisation il faut charger le module *Pervasives* pour avoir un fonctionnement minimal correct, sinon on ne peut pas faire grand-chose. Comme ce module est généralement présent par défaut, on écrit le code de chargement du module dans le constructeur du composant XPCOM.

```
static int flag = 0;
if(!flag){
  flag = 1;
  char *name = "Eval";
  char *argv[] = {name, NULL};
  caml_startup(argv);
  exec("open Pervasives;;");
}
```

FIG. 3.3 – Initialisation d'O'Caml et chargement du module Pervasives

On représente les callbacks pour le toplevel par le schéma 3.7.

Pour utiliser notre composant, il faut écrire une fonction Javascript qui prend une chaîne de caractères en paramètre et retourne une chaîne de caractères à afficher.

La figure 3.5 montre une exécution dans XULRunner.

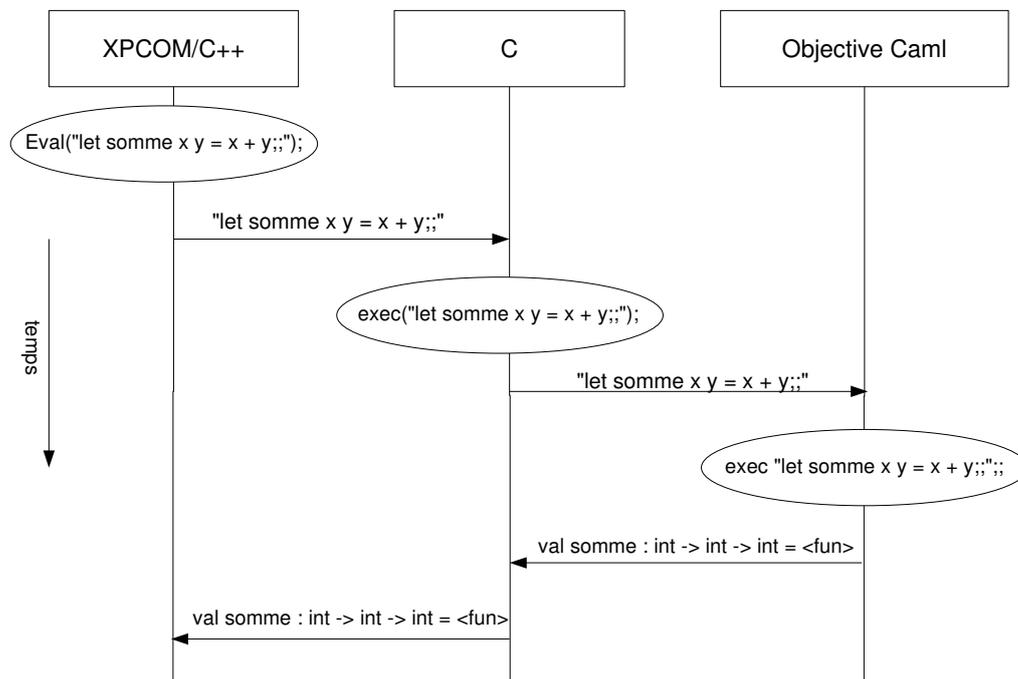


FIG. 3.4 – Callbacks du toplevel

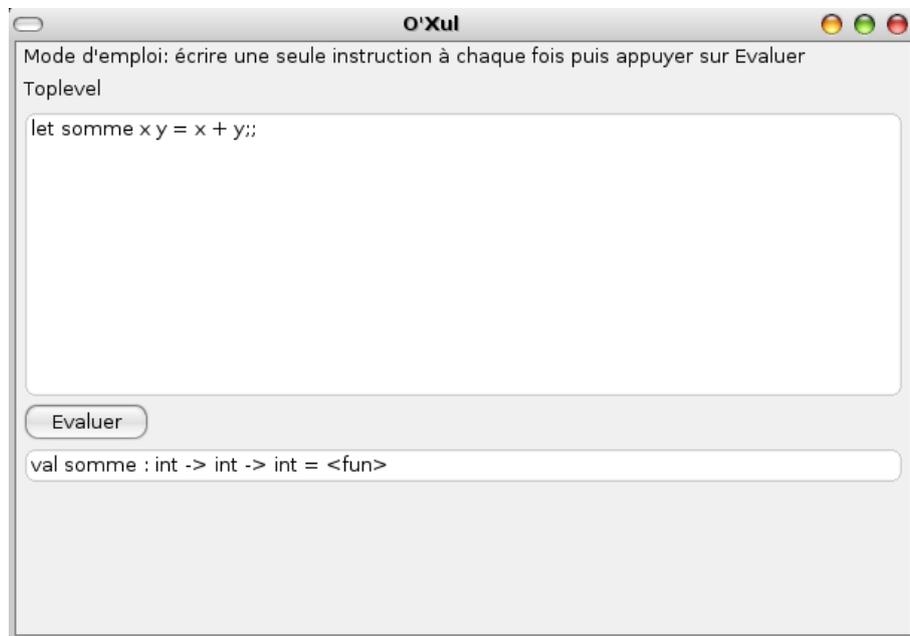


FIG. 3.5 – Une instruction O'Caml et son résultat évalué sous XULRunner

Voyons maintenant comment se fait l'appel à l'évaluation de notre instruction O'Caml par le toplevel embarqué dans une application, plus exactement une extension, Firefox.

### 3.1.3 Extension Firefox utilisant un XPCOM appelant du O'Caml

Avant de voir comment se fait les appels de fonctions entre les différents langages, on va commencer par voir l'organisation d'une extension Firefox.

#### Structure d'une extension

Une extension Firefox est, comme une application XULRunner, composée au moins des fichiers indispensables avec les noms par défaut. Voir l'arborescence de la figure 3.6 sera plus explicite.

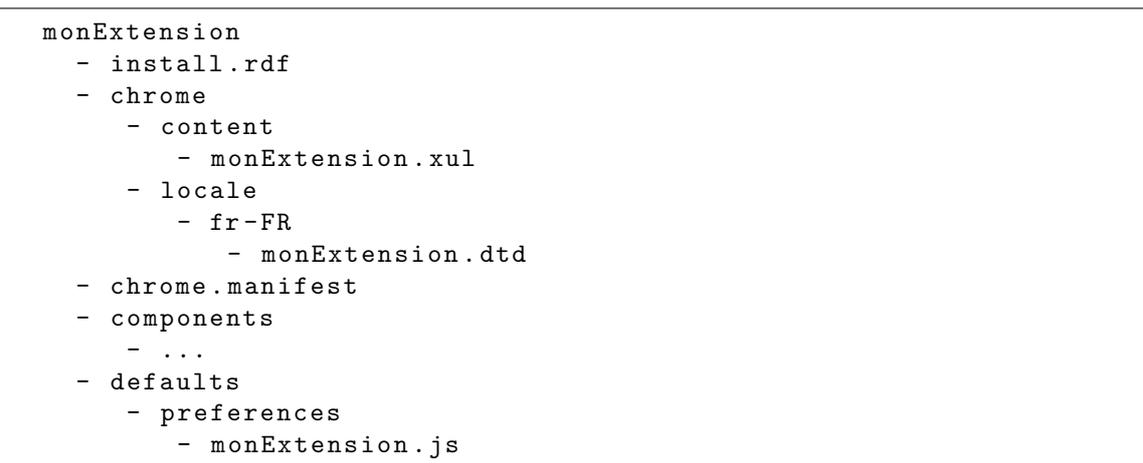


FIG. 3.6 – Arborescence d'une extension Firefox

Tout ce qui est contenu sous le répertoire chrome peut, mais n'est pas obligatoire, être archiver dans un fichier *.jar*. Si c'est le cas, il faut le préciser dans le fichier chrome.manifest. Celui-ci est de la forme suivante :

```
content oxul chrome/content/oxul
```

où *content* désigne le répertoire content, *oxul* le nom de l'application (qui doit contenir un fichier oxul.xul), et enfin l'adresse à partir de l'emplacement du fichier manifest où se trouve justement le fichier oxul.xul. Dans le cas d'un fichier jar, ce sera de cette forme :

```
content oxul jar :chrome/oxul.jar!/content/oxul
```

On vient de construire l'arborescence, pour que Firefox reconnaisse notre paquetage en tant qu'extension il faut faire une archive avec l'extension *.xpi*.

#### Utilisation de l'extension

L'utilisation de l'extension ne peut se faire qu'une fois qu'il y a le fichier d'interface XUL et le script Javascript. Dans le fichier XUL, on doit récupérer l'événement qui déclenche l'évaluation, comme par exemple un élément bouton. Lorsque l'utilisateur cliquera dessus, Javascript entrera en action et instanciera le composant XPCOM et lui demandera d'exécuter son code. Dans notre exemple du toplevel embarqué, on passe ainsi du langage XUL à du Javascript qui appelle du C++, celui-ci passe par du C qui appelle du O'Caml pour évaluer la phrase. Le code retourné par O'Caml passe par le chemin inverse.

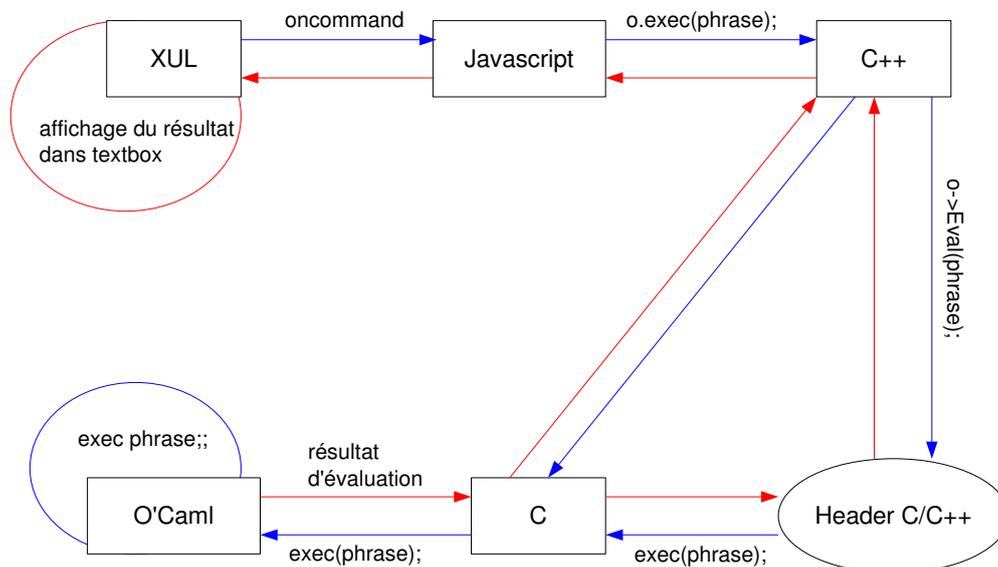


FIG. 3.7 – Les différents appels entre les langages

**flèche bleu (extérieur) :** le parcourt des appels lorsque l'utilisateur envoie du code O'Caml à traiter

**flèche rouge (intérieur) :** le parcourt du code traité

On remarque la flèche oblique qui va du C au C++ : en effet, si on écrit le code C directement dans le C++ alors on a plus besoin de l'en-tête, mais on passe quand même par du C.

La figure 3.8 montre ce que donne notre toplevel dans Firefox. On voit qu'en enlevant la barre de menu c'est exactement la même chose que sous XULRunner.

A ce stade, on sait qu'il est possible d'intégrer du code O'Caml dans les composants XPCOM. Etudions la faisabilité de l'opération inverse.

### 3.1.4 Code O'Caml appelant des méthodes C++ d'XPCOM

Une application XUL peut utiliser du code O'Caml à travers XPCOM, voyons si l'inverse est possible. Cela permettra d'avoir des applications O'Caml qui utilise le moteur de rendu de Mozilla par exemple.

#### Petite application autonome

Ecrire du code C ou C++ pour qu'il s'interface avec O'Caml revient au même à peu de chose près. En effet, on pourrait dire qu'il "*suffit*" d'ajouter le mot-clé *extern* "C" aux fonctions, et de préciser au compilateur O'Caml que le code est compilé avec un

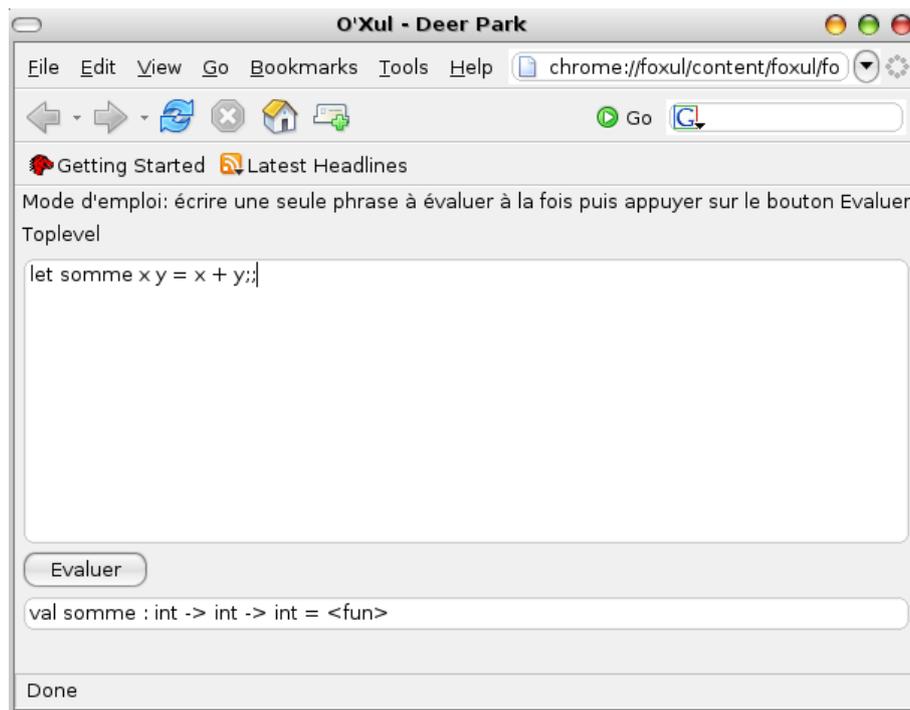


FIG. 3.8 – Une instruction O'Caml et son résultat dans une extension Firefox

compilateur C++ tel que `g++`. Dans le cadre de l'écriture de ma petite application, c'est ce que je fait.

Si on ne sait pas comment se code en C++ XPCOM, le fichier `nsTestSample.cpp` qui se trouve dans les sources de Mozilla est un bon point de départ. C'est un exemple que les développeurs ont mis dans le code pour montrer l'autonomie de XPCOM. Il utilise donc les composants XPCOM sans avoir besoin de XULRunner ou Firefox. En se basant sur ce code et en modifiant le composant auquel je veux accéder, en l'occurrence celui du `pgcd`, j'écris une fonction C/C++ qui prend deux *value*, qui sont des entiers *int*, en paramètre et appelle la méthode C++. Le résultat du calcul est renvoyé à O'Caml. On remarque les *step* qui sont commentaires dans le code, ils représentent les différentes étapes de l'emploi d'un XPCOM. Plus de détails à la sous-section Modularisation.

A ce stade, on compile avec `g++` en lui donnant les bons répertoires où se trouvent les en-têtes. Le `makefile` figure 1.14 est réutilisable en modifiant le nécessaire. Le plus important est de préciser à O'Caml que le fichier object C/C++ qu'il utilise est compilé avec `g++` par

```
ocamlc -custom -cc g++ gcd.o pgcd.ml ...
```

où `gcd.o` est le fichier en question.

Dans cette application, O'Caml ne fait pas grand-chose et on initialise et arrête XPCOM à chaque fois que l'on lance l'application. Cela est tout à fait correct pour une petite application, mais qu'en est-il des applications qui font plusieurs appels de méthodes, et de fonctions O'Caml? Il faut séparer donc les différentes étapes, et faire en sorte que tout soit contrôlé du côté d'O'Caml.

```

extern 'C' value getPgcd(value pN,value pM){
{
  CAMLparam2(pN,pM);
  int n,m;
  n = Int_val(pN);
  m = Int_val(pM);
  nsresult rv;

  /* Step 1 */
  XPCOMGlueStartup(nnull);

  // Initialize XPCOM
  nsCOMPtr<nsIServiceManager> servMan;
  rv = NS_InitXPCOM2(getter_AddRefs(servMan), nnull, nnull);
  if (NS_FAILED(rv))
  {
    failwith("ERROR: XPCOM initialization error");
  }
  // register all components in our default component directory
  nsCOMPtr<nsIComponentRegistrar> registrar = do_QueryInterface(servMan);
  NS_ASSERTION(registrar, "Null nsIComponentRegistrar");
  registrar->AutoRegister(nnull);

  nsCOMPtr<nsIComponentManager> manager = do_QueryInterface(registrar);
  NS_ASSERTION(registrar, "Null nsIComponentManager");

  /* Step 2 */
  // Create an instance of our component
  nsCOMPtr<nsIPgcd> gcd;
  rv = manager->CreateInstanceByContractID(NS_PGCD_CONTRACTID,
                                          nnull,
                                          NS_GET_IID(nsIPgcd),
                                          getter_AddRefs(gcd));

  if (NS_FAILED(rv))
  {
    failwith("Cannot create instance of component\n");
  }

  /* Step 3 */
  PRInt32 res;
  rv = gcd->Pgcd(n,m,&res);
  if(NS_FAILED(rv))
    failwith('Can't get the pgcd from given numbers\n');

  /* Step 4 */
  // All nsCOMPtr's must be deleted prior to calling shutdown XPCOM
  // as we should not hold references passed XPCOM Shutdown.
  servMan = 0;
  registrar = 0;
  manager = 0;
  gcd = 0;

  /* Step 5 */
  // Shutdown XPCOM
  NS_ShutdownXPCOM(nnull);

  XPCOMGlueShutdown();

  CAMLreturn(Val_int(res));
}

```

FIG. 3.9 – Standalone XPCOM (XPCOM autonome)

## Modularisation

Dans la section précédente, tout le code est regroupé en une seule fonction et il n'est pas possible d'intervenir au milieu du traitement de la fonction. Or il faudrait que du côté du développeur O'Caml, il puisse faire par exemple plusieurs calcul du pged avant d'arrêter XPCOM, dans ce but on va modulariser les parties du code précédent (fig. 3.9).

« La modularité est le principe de toute technologie COM. »

L'utilisation d'XPCOM peut se découper en plusieurs étapes (schéma figure 3.10) :

- L'initialisation de la prise en charge des composants XPCOM
- La création du composant
- L'appel des méthodes du composant
- Suppression du composant
- Arrêt d'XPCOM

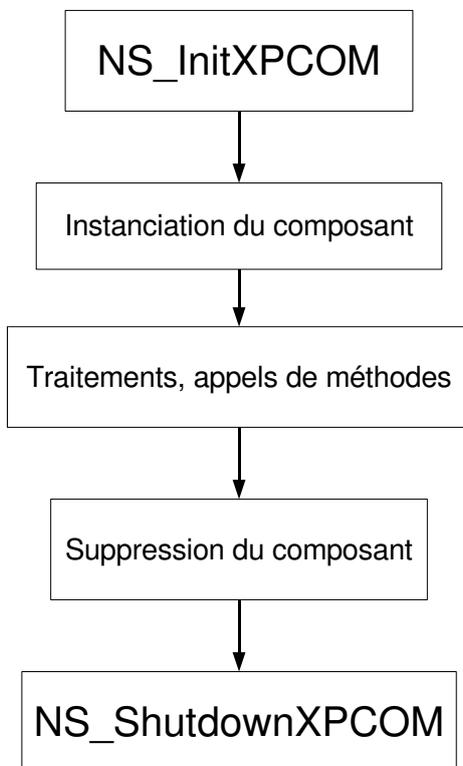


FIG. 3.10 – Les différents étapes d'utilisation d'XPCOM

A ces étapes on peut voir la correspondance avec le code précédent.

Le découpage que j'ai choisi se fait de la manière suivante :

- à chaque étape on associe une fonction C et une déclaration *external* O'Caml. En séparant l'initialisation et l'arrêt d'XPCOM des phases que je qualifierais de traitement, on se retrouve avec une plus grande facilité de manipulation des XPCOM. D'une part, l'initialisation est faite qu'une seule fois, d'autre part, on peut instancier différents XPCOM et les utiliser jusqu'à l'arrêt d'XPCOM qui se doit d'être fait par le programmeur. On devra utiliser des variables globales, sans oublier de déclarer chaque fonction en *extern "C"* si on utilise un compilateur C++.

En appliquant ce procédé, les seules parties qui changent sont l'instanciation du composant, l'écriture des méthodes de traitement et la mise à zéro de tous les pointeurs de classes. O'Caml pourra alors utiliser l'ensemble des XPCOM écrits en C++ (le reste étant en Javascript). Vu la quantité de classes et interfaces présentes dans XPCOM (le code source fait plus de quatre millions de lignes!), il faut écrire un générateur de code pour ces parties. L'idée serait de prendre un fichier XPIDL et de produire le code C et O'Caml correspondant[7].

En prenant le code figure 3.9, le découpage donne les bouts de code suivants :

```
#include 'nsXPCOMGlue.h'
#include 'nsXPCOM.h'
#include 'nsCOMPtr.h'
/* et bien d'autres encore */

nsCOMPtr<nsIServiceManager> servMan;
nsCOMPtr<nsIComponentRegistrar> registrar;
nsCOMPtr<nsIComponentManager> manager;
```

FIG. 3.11 – Etape 0 : includes et variables globales

Fig. 3.11 : parce qu'elles sont utilisées un peu n'importe où, on doit les garder en globales.

```
extern "C" value caml_initXPCOM(value v)
{
    CAMLparam1(v);
    nsresult rv;

    XPCOMGlueStartup(nsnull);

    rv = NS_InitXPCOM2(getter_AddRefs(servMan), nsnull, nsnull);
    if (NS_FAILED(rv))
    {
        failwith("ERROR: XPCOM initialization error");
    }

    registrar = do_QueryInterface(servMan);
    NS_ASSERTION(registrar, "Null nsIComponentRegistrar");
    registrar->AutoRegister(nsnull);

    manager = do_QueryInterface(registrar);
    NS_ASSERTION(manager, "Null nsIComponentManager");

    CAMLreturn(v);
}
```

FIG. 3.12 – Etape 1 : initialisation d'XPCOM

Fig. 3.12 : lorsque O'Caml utilisera cette méthode, XPCOM est démarré et les composants accessibles sont enregistrés.

Fig. 3.13 : cette fonction doit être modifiée pour chaque classe que l'on veut, ce qui n'est pas bon du tout. La variable gcd doit être globale pour que la fonction de l'étape suivante

```

nsCOMPtr<nsIPgcd> gcd;

extern ‘‘C’’ value createXPCOM(value v)
{
  CAMLparam1(v);
  nsresult rv;

  rv = manager->CreateInstanceByContractID(NS_PGCD_CONTRACTID,
                                           nsnull,
                                           NS_GET_IID(nsIPgcd),
                                           getter_AddRefs(gcd));

  if (NS_FAILED(rv))
  {
    failwith("Cannot create instance of component\n");
  }
  CAMLreturn(v);
}

```

FIG. 3.13 – Etape 2 : instanciation d’une classe donnée

puisse l’utiliser. De plus il y a ce qu’on appelle les services et les instances. On obtient un service par le *ServiceManager* (*servMan*) et par la méthode *GetServiceByContractID*. La différence entre service et instance se situe au niveau de la portée. Une instance est un objet amenée à être modifiée, tandis qu’un service modifie les données, et il en existe généralement qu’un seul.

```

extern ‘‘C’’ value callPgcd(value pn, value pm)
{
  CAMLparam2(pn, pm);
  nsresult rv;
  int n = Int_val(pn);
  int m = Int_val(pm);

  PRInt32 res;
  rv = gcd->Pgcd(n,m,&res);
  if (NS_FAILED(rv))
  {
    failwith("Can't get the pgcd from given numbers\n");
  }
  CAMLreturn(Val_int(res));
}

```

FIG. 3.14 – Etape 3 : traitement

Fig. 3.14 : cette fonction est écrite que pour un appel à la méthode *pgcd* du composant. On commence à voir les problèmes qu’il va y avoir, notamment l’écriture de code dans deux langages différents (C++/O’Caml) pour une opération toute simple.

Fig. 3.15 : on remarque qu’on pourrait aussi séparer les variables globales communes à toutes les applications et celle propre à notre application.

Fig. 3.16 : à partir de là, il n’est plus possible d’appeler les autres fonctions sans faire au préalable l’appel à *caml\_initXPCOM*.

```
extern 'C' value removeXPCOM(value v)
{
  CAMLparam1(v);

  servMan = 0;
  registrar = 0;
  manager = 0;
  gcd = 0;

  CAMLreturn(v);
}
```

FIG. 3.15 – Etape 4 : suppression des pointeurs

```
extern 'C' value caml_shutdownXPCOM(value v)
{
  CAMLparam1(v);

  NS_ShutdownXPCOM(nsnull);
  XPCOMGlueShutdown();

  CAMLreturn(v);
}
```

FIG. 3.16 – Etape 4 : arrêt d'XPCOM

Finalement on voit les problèmes que se posent l'intégration d'XPCOM dans O'Caml. Néanmoins cela fonctionne malgré la mauvaise efficacité du choix de l'implémentation. On verra à la fin quels autres choix possibles on pourrait faire pour améliorer les choses.

Pour compléter le travail, voyons les comportements de notre application dans une application graphique. Pour cela, il va falloir ouvrir une fenêtre ou accéder à une fenêtre.

### 3.1.5 Pgcd graphique

Avec les sections précédentes on a la base du calcul. De plus les fonctions sont divisées pour être plus modulaires, ce qui facilite les développements. Il faut maintenant passer à l'étude des fonctions de traitement graphiques. Deux possibilités se sont offertes pour définir les fenêtres graphiques : soit on utilise les fichiers XUL, soit on décrit nous même le contenu de la fenêtre à partir des classes XPCOM. On choisit la plus simple, le fichier XUL.

Les classes de l'API XPCOM proposent un ensemble de fonctions pour manipuler le DOM (Document Object Model) et gérer les applications autonomes. On jette un coup d'oeil dans les sources sur le lancement de XULRunner (qui n'est rien d'autre qu'un *main* avec initialisation des profils Mozilla) on voit les classes qui sont utilisées.

On reprend donc les fonctions définies auparavant qui sont réutilisables et on ajoute la fonction d'ouverture de la fenêtre.

Malheureusement ce n'est pas aussi simple, il faut au préalable définir plusieurs services, et initialiser GTK par un `gtk_init` (sous Linux, sous Windows pour ce que j'en sais il n'y a rien à faire à part encapsuler dans le `main` Windows, mais il faudrait étudier plus en

```

nsCOMPtr<nsIDOMWindow> domwindow;

extern "C" value openWindow(value url,value options)
{
  CAMLparam2(url,options);
  nsresult rv;
  char *aUrl = String_val(url);
  char *aOptions = String_val(options);

  rv = watcher->OpenWindow(nsnull,aUrl,
                          nsnull,aOptions,nsnull,
                          getter_AddRefs(domwindow));
  if(NS_FAILED(rv))
    failwith("Can't open main thread window\n");

  rv = watcher->SetActiveWindow(domwindow);
  if(NS_FAILED(rv))
    failwith("Cannot set Active Window\n");

  CAMLreturn(Val_unit);
}

```

FIG. 3.17 – Ouverture d'une fenêtre

détail).

Il y a trois services qui doivent être requis :

- nsIAppStartup : cette interface n'a pas beaucoup de méthodes, la seule qui nous intéresse est la méthode *Run* qui une fois lancée, ne s'arrête que si on quitte l'application.
- nsIWindowCreator : sans cette interface, impossible de créer la fenêtre. Elle est appelée quand une nouvelle fenêtre va être créée.
- nsIWindowWatcher : c'est en utilisant cette interface qu'on fait appel à sa méthode *OpenWindow*. Il faut la lier avec le service précédent, pour que la création de la fenêtre puisse s'effectuer (voir figure 3.18).

Ces trois services sont déclarées en variables globales car elles seront utilisées dans d'autres fonctions. Les deux *define* sont utilisés juste après.

```

#define NS_WINDOWWATCHER_CONTRACTID \
  "@mozilla.org/embedcomp/window-watcher;1"
#define NS_APPSTARTUP_CONTRACTID \
  "@mozilla.org/toolkit/app-startup;1"

nsCOMPtr<nsIAppStartup> app;
nsCOMPtr<nsIWindowWatcher> watcher;
nsCOMPtr<nsIWindowCreator> creator;

```

Leur initialisation est faite par la fonction suivante : (figure 3.18).

Le résultat de l'exécutable une fois lancé est celui de la figure 1.3 puisque le fichier XUL utilisé est le même. Ce fichier contient du Javascript et notre application interprète le code Javascript. Pour être plus exacte, Javascript (JS) peut être interprété si l'url donné en paramètre utilise le protocole *chrome*. C'est à dire que si le fichier se trouve dans le répertoire *chrome* de XULRunner, ou celui de notre application si c'est défini, alors la commande *self.close()* par exemple sera bien exécutée et terminera l'application. Au contraire, si le fichier donné en paramètre est du type *file ://... Javascript* n'est pas

```

extern "C" value createXPCOM(value v)
{
  CAMLparam1(v);
  nsresult rv;

  rv = servMan->GetServiceByContractID(NS_APPSTARTUP_CONTRACTID,
                                       NS_GET_IID(nsIAppStartup),
                                       getter_AddRefs(app));

  if(NS_FAILED(rv))
    failwith("Cannot create appshell\n");

  creator = do_GetService(NS_APPSTARTUP_CONTRACTID);
  if (!creator) return NS_ERROR_UNEXPECTED;

  watcher = do_GetService(NS_WINDOWWATCHER_CONTRACTID, &rv);

  if (NS_FAILED(rv)){
    failwith("ERROR: Cannot create instance of component.\n");
  }

  rv = watcher->SetWindowCreator(creator);
  if(NS_FAILED(rv))
    failwith("ERROR: Cannot set creator\n");

  CAMLreturn(v);
}

```

FIG. 3.18 – Obtention des services de fenêtrage et d'application

connu et rien ne se passe en cliquant sur le bouton Quit. Qu'à cela ne tienne, le but est de ne pas utiliser Javascript dans cette partie.

L'application pged doit se procéder ainsi : l'utilisateur entre deux nombres dans les champs de texte et lorsqu'il clique sur le bouton *compute* alors le calcul se fait et le résultat s'affiche.

Le problème qui se pose alors est de récupérer les valeurs des champs de texte (textbox, pour faire plus court).

### Manipuler le DOM

Un fichier XUL est un fichier dérivé du XML, donc peut être défini par le DOM. Pour obtenir la valeur du textbox, il faut penser en DOM. Pour cela il faut récupérer le *Document*, l'*Element* et le reste. L'interface nsIDOMWindow a une méthode *GetDocument* qui renvoie un nsIDOMDocument.

La fonction suivante (fig. 3.19) récupère le *document* à partir du *window*, puis l'*element* du *document*, ... Elle prend un id et le nom de l'attribut et renvoie la valeur suivant ces paramètres. L'interface qu'on utilise, nsIDOMElement, offre une méthode *GetAttribute* qui prend le nom de l'attribut et renvoie la valeur associée. Dans le cas du textbox, si on applique la méthode *GetAttribute* sur *value*, la valeur obtenue est la valeur par défaut, de plus si *value* n'est pas défini dans le fichier XUL alors la méthode échoue. Pour contourner ce problème, il faut caster nsIDOMElement en nsIDOMXULTextboxElement, faire un QueryInterface (QI) dans le jargon XPCOM. Ainsi dès qu'il y a changement de la valeur du champ de texte l'attribut est mis à jour.

```

extern "C" value getTextboxAttribute(value pId,value pAttribute){
    CAMLparam2(pId,pAttribute);
    nsresult rv;
    char * id = String_val(pId);
    char * attr = String_val(pAttribute);

    nsCOMPtr<nsIDOMDocument> document;
    rv = domwindow->GetDocument(getter_AddRefs(document));
    if(NS_FAILED(rv))
        failwith("Cannot get DOM document\n");

    if(document){
        nsCOMPtr<nsIDOMELEMENT> element;
        rv = document->GetElementById(getnsString(id),
                                      getter_AddRefs(element));

        if(NS_FAILED(rv))
            failwith("Cannot get DOM element\n");

        if(element){
            nsCOMPtr<nsIDOMXULTextXMLElement> text (do_QueryInterface(element));

            if(text){
                if(strcmp(attr,"value")==0){
                    nsEmbedString attrval;
                    rv = text->GetValue(attrval);
                    if(NS_FAILED(rv))
                        failwith("Cannot get attribute value\n");
                    CAMLreturn(copy_string(getnsCString((PRUnichar*)attrval.get()).get()));
                } else if(strcmp(attr,"size")==0){
                    PRInt32 size;
                    rv = text->GetSize(&size);
                    if(NS_FAILED(rv))
                        failwith("Cannot get attribute size\n");
                    char s[32];
                    sprintf(s,"%d",size);
                    CAMLreturn(copy_string(s));
                }
            } else {
                failwith("Can't get textbox\n");
            }
        }
    }
    //le textbox n'a rien ou n'a pas pu etre recuperé
    CAMLreturn(copy_string(""));
}

```

FIG. 3.19 – Récupérer un attribut d'un élément Textbox

Deux remarques sur cette fonction : elle ne traite que deux attributs, le type `nsIDOMXULTextXMLElement` ayant une méthode pour chaque attribut ; les fonctions `getnsString` et `getnsCString` (voir section `nsString` et `nsCString`).

### Gérer les événements DOM

Le code tel qu'il est là ne fonctionne pas et le script `run-mozilla.sh` s'arrête lamentablement avec un message d'erreur qui ne veut rien dire. Lorsque l'on lance l'application la fenêtre apparaît mais aucune opération n'est possible. Quelques tests montre que le DOM qu'on récupère est vide. La raison est qu'on effectue des traitements avant d'avoir appelé la

méthode `Run` de `nsIAppStartup`, et après `Run` on ne peut plus rien faire. Il faut donc attendre que le fichier soit chargé. Il existe un événement DOM qui indique quand une page est chargée.

Lorsque l'on parle de DOM, il faut penser événements et écoute d'événements. En ajoutant l'événement `load` sur la fenêtre on peut récupérer la main et faire les opérations requises. Il faut avoir au préalable écrit une classe d'écoute héritant de `nsIDOMEventListener` qui offre la seule méthode `HandleEvent`. La fonction d'ajout de `load` (ou tout autre event) est écrit comme cela :

```
extern "C" value addWindowEventListener(value event){
    CAMLparam1(event);
    nsresult rv;
    char * aEvent = String_val(event);

    listener *mylistener = new listener();

    nsCOMPtr<nsIDOMEventTarget> target;
    target = do_QueryInterface(domwindow);
    if(target){
        rv = target->AddEventListener(getnsString(aEvent),
                                     mylistener, PR_FALSE);
        if(NS_FAILED(rv))
            failwith("Cannot add event listener\n");
    }
    CAMLreturn(Val_unit);
}
```

FIG. 3.20 – Ajout d'événement à la fenêtre

La gestion des événements a besoin d'être écrit de manière à ce que du côté d'O'Caml on donne le nom de l'événement et ce soit ajouté automatiquement. Pour cela j'ai choisi de définir une structure qui mémorise l'id et l'événement en question en C.

```
typedef struct caml_event {
    int size;
    char *id;
    char *event;
} caml_event;

caml_event *caml_events;
int caml_event_counter;
```

O'Caml n'a plus qu'à utiliser la fonction définie en external

```
external addEventListener : string -> string -> bool = "caml_addEventListener"
```

Comme il n'y a pas qu'un seul événement, on crée un tableau de structures. Ce tableau sera initialisé chaque fois que O'Caml utilisera la fonction suivante avec en paramètre le nombre d'événements

```
external initEventQueue : int -> unit = "caml_initEventQueue"
```

Au final on se retrouve avec :

- un DOM accessible
- un DOM modifiable avec récupération des données mises à jour

Pour clore l'application du pgcd : on a du côté O'Caml des fonctions qui manipulent XP-COM. Il revient au développeur O'Caml d'écrire la fonction pgcd (ou celle qu'il veut) en O'Caml, ainsi que la fonction qui récupère le champ de texte (en appelant `getTextboxAttribute`) et fait le calcul, puis il doit l'enregistrer par un `Callback.register`. Il lui reste plus qu'à initialiser XPCOM avec les différentes fonctions définies avant et lancer l'application.

Pour mieux illustrer une application plus importante et voir la marche à suivre je vous renvoie à la section suivante (3.2).

### 3.1.6 nsString et nsCString

Je me réserve une sous-section pour parler des chaînes de caractères dans Mozilla, car l'utilisation est plutôt étrange et perturbant.

L'API sur les string offre un ensemble de macros qui permet de construire une chaîne de caractères C en objet `nsAString`, qui est la classe abstraite racine des chaînes de caractères. `nsString` dérive de `nsAString`. Toutes les interfaces du DOM par exemple prend des chaînes de type `nsAString` en paramètre. Cette classe définit les chaînes en unicode codées sur deux octets. Les macros préconisées qui prend une chaîne C et qui retourne une `nsAString` sont les suivantes

```
NS_LITERAL_STRING("textbox")
NS_NAMED_LITERAL_STRING(var,"textbox")
```

Hélas, le résultat retourné n'est pas celui escompté. Lorsque l'on fait un `GetElementsByTagName` sur un `textbox` en donnant la variable `var` de la macro, la liste de noeuds retournée est vide.

Pour contourner le problème, on utilise deux autres macros : `NS_UTF16ToCString` et `NS_CStringToUTF16`. Elles travaillent sur les classes `nsString` et `nsCString`. `nsCString` est une classe de chaînes de caractères codée sur un octet. Le C désignant le langage de même nom. On écrit deux fonctions qui va faciliter la manipulation des chaînes : `getnsString` et `getnsCString`.

```
nsCString getnsCString(nsAString ustr)
{
    nsEmbedCString str;
    NS_UTF16ToCString(ustr, NS_CSTRING_ENCODING_UTF8, str);
    return str;
}
```

On donne en paramètre un objet d'une chaîne de caractères au format unicode, et il est converti en chaîne de caractères C. Pour récupérer la chaîne de caractères sur l'objet `nsCString` on applique la méthode `get()`. Le type retourné alors est `const char *`.

```
nsString getnsString(char *s)
{
    nsEmbedCString str(s);
    nsEmbedString ustr;
    NS_CStringToUTF16(str, NS_CSTRING_ENCODING_UTF8, ustr);
    return ustr;
}
```

Le paramètre de la fonction `getnsString` est de type `char *` car il s'interface mieux avec O'Caml et qu'il est plus utilisé. La fonction précédente prend un objet car les méthodes des interfaces du DOM retournent toujours un `nsAString`.

Avec ces deux fonctions il est possible désormais de faire un appel comme celui-ci :

```
rv = element->GetElementsByTagName(getnsString("textbox"),
                                   getter_AddRefs(nodelist));
```

s'il y a des éléments `textbox`, il ne devrait plus renvoyer une liste de noeuds vide.

Ceci termine notre étude, qui finalement est concluante sur la faisabilité de l'interfaçage. Cependant, vu l'ampleur des possibilités offertes par XPCOM et O'Caml, il serait intéressant d'approfondir les recherches.

## 3.2 Application O'Xul

On essaye ici de créer une application autonome en O'Caml et utilisant un fichier d'interface XUL et XPCOM. L'application est une calculatrice. On va voir toutes les étapes à suivre dans l'ordre pour un programmeur O'Xul.

### 3.2.1 Interface graphique : XUL

La première des choses à faire est d'écrire le fichier d'interface XUL. En vérifiant bien qu'un id unique est attribué à chaque élément qu'on veut manipuler. Notre calculatrice est représentée par un `textbox` pour afficher les valeurs et une grille pour les boutons des chiffres et opérateurs. Le rendu de l'affichage est le suivant (figure 3.21) :

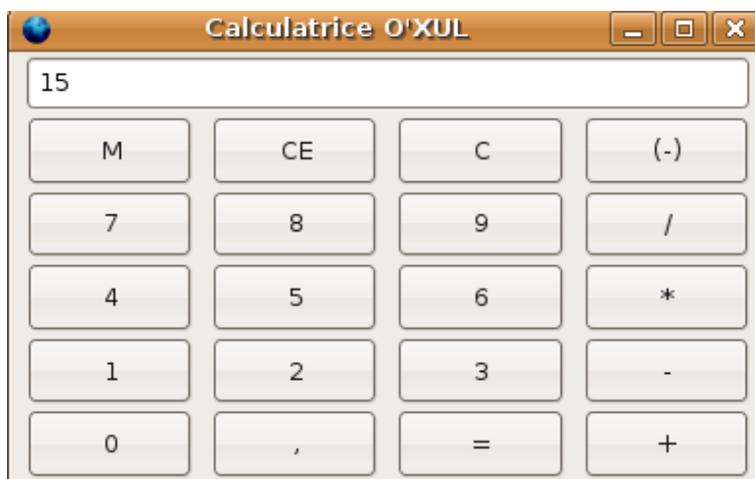


FIG. 3.21 – Affichage de la calculatrice

Il faut voir la correspondance des boutons avec l'attribut `label` de l'élément `button`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>

<window
  id="calc"
  title="Calculatrice O'XUL"
```

```

xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

<hbox>
  <textbox id="text" flex="1" align="right"/>
</hbox>
<grid flex="1">
  <columns>
    <column flex="1"/>
    <column flex="1"/>
    <column flex="1"/>
    <column flex="1"/>
  </columns>
  <rows>
    <row flex="1">
      <button id="b10" label="M" value="M" flex="1"/>
      <button id="b11" label="CE" value="CE" flex="1"/>
      <button id="b12" label="C" value="C" flex="1"/>
      <button id="b13" label="(-)" value="(-)" flex="1"/>
    </row>
    <row flex="1">
      <button id="b20" label="7" value="7" flex="1"/>
      <button id="b21" label="8" value="8" flex="1"/>
      <button id="b22" label="9" value="9" flex="1"/>
      <button id="b23" label="/" value="/" flex="1"/>
    </row>
    <row flex="1">
      <button id="b30" label="4" value="4" flex="1"/>
      <button id="b31" label="5" value="5" flex="1"/>
      <button id="b32" label="6" value="6" flex="1"/>
      <button id="b33" label="*" value="*" flex="1"/>
    </row>
    <row flex="1">
      <button id="b40" label="1" value="1" flex="1"/>
      <button id="b41" label="2" value="2" flex="1"/>
      <button id="b42" label="3" value="3" flex="1"/>
      <button id="b43" label="-" value="-" flex="1"/>
    </row>
    <row flex="1">
      <button id="b50" label="0" value="0" flex="1"/>
      <button id="b51" label="," flex="1"/>
      <button id="b52" label="=" value="=" flex="1"/>
      <button id="b53" label="+" value="+" flex="1"/>
    </row>
  </rows>
</grid>
</window>

```

Une fois l'interface écrite on connaît les id que l'on veut. On passe ensuite au traitement des événements.

### 3.2.2 Gestion des événements

On reprend les fichiers créés avant sur les événements et la gestion des XPCOM. On fournit une méthode de plus : `getButtonAttribute`, pour la manipulation des attributs des boutons. Du côté de C, celui-ci associe à chaque id et événement une fonction stockée en chaîne de caractères. Cette fonction sera appelée par un callback lorsqu'un événement est déclenché. Pour le moment, les fonctions O'Caml doivent être écrites en ne prenant qu'un paramètre *unit*. Comme en C, on utilise un tableau de *camLevent*, on fournit une fonction qui prend une liste d'id et d'actions et les ajoute dans le tableau. Les deux listes doivent



```

        setTextboxAttribute "text" "value" !fst_operand;
        operateur := "");
else
  (if !fst_operand = "" then
    (let t= getTextboxAttribute "text" "value" in
      if t = "" then
        fst_operand := string_of_float 0.
      else
        fst_operand := t;
        operateur := (getButtonAttribute id "value");
        setTextboxAttribute "text" "value" "");
    else
      (let t = getTextboxAttribute "text" "value"
        in
          fst_operand := string_of_float(resolv_op_f !operateur
            (float_of_string(!fst_operand))
            (float_of_string(t)));
          setTextboxAttribute "text" "value" !fst_operand;
          operateur := ""));;

(* fonction pour les chiffres *)
let operande id =
  let p = getTextboxAttribute "text" "value" in
    setTextboxAttribute "text" "value" (p^(getButtonAttribute id "value"))
;;

```

Il faut ensuite écrire une fonction par bouton. Chaque nom de fonction correspond au label du bouton de même nom. Le code étant long, je n'ai pas mis les fonctions qui se répètent comme celles des chiffres et opérateurs.

```

(* Somme -> bouton M *)
let b10 () =
  if !fst_operand = "" then
    (fst_operand := (getTextboxAttribute "text" "value");
     setTextboxAttribute "text" "value" "");
  else
    (if not(!flotte) then
      (fst_operand :=
        string_of_int((int_of_string(!fst_operand)) +
          (int_of_string(getTextboxAttribute "text" "value")
            ));
       setTextboxAttribute "text" "value" "");
     else
      (fst_operand :=
        string_of_float((float_of_string(!fst_operand)) +.
          (float_of_string(getTextboxAttribute "text" "value"
            ")));
        setTextboxAttribute "text" "value" "value" ""));
;;
(* bouton CE *)
let b11 () = setTextboxAttribute "text" "value" "";;

(* bouton C *)
let b12 () =
  setTextboxAttribute "text" "value" "";
  fst_operand := "";
  operateur := "";
  flotte := false;;

(* bouton (-) *)
let b13 () =

```

```

if !fst_operand = "" then
  let p = getTextboxAttribute "text" "value" in
  if not(!flotte) then
    let r = string_of_int(-(int_of_string p)) in
    setTextboxAttribute "text" "value" r
  else
    let r = string_of_float(-.(float_of_string p)) in
    setTextboxAttribute "text" "value" r
else
  (let r = (getTextboxAttribute "text" "value") in
  if not(!flotte) then
    fst_operand := string_of_int(-(int_of_string(r)))
  else
    fst_operand := string_of_float(-.(float_of_string(r)));
  setTextboxAttribute "text" "value" !fst_operand);

(* bouton chiffre 7,8,9 et operateur / *)
let b20 () = operande "b20";;
let b21 () = operande "b21";;
let b22 () = operande "b22";;
let b23 () = opfun "b23";;

(* ... les autres chiffres et operateurs ... *)

(* bouton , *)
let b51 () = flotte := true;
  let t = getTextboxAttribute "text" "value" in
  setTextboxAttribute "text" "value" (t^".");;

(* bouton = *)
let b52 () =
  let t = getTextboxAttribute "text" "value" in
  if not(!flotte) then
    if t = "" then
      let r = string_of_int(resolv_op !operateur
                            (int_of_string(!fst_operand))
                            (0))
      in
      setTextboxAttribute "text" "value" r
    else
      let r = string_of_int(resolv_op !operateur
                            (int_of_string(!fst_operand))
                            (int_of_string(t)))
      in
      setTextboxAttribute "text" "value" r;
  else
    (if t = "" then
      let r = string_of_float(resolv_op_f !operateur
                              (float_of_string(!fst_operand))
                              0.)
      in
      setTextboxAttribute "text" "value" r
    else
      let r = string_of_float(resolv_op_f !operateur
                              (float_of_string(!fst_operand))
                              (float_of_string(t)))
      in
      setTextboxAttribute "text" "value" r);
  operateur := "";
  fst_operand := "";;

```

Enfin, une fois ces fonctions écrites, il faut les enregistrer avec *Callback.register*. On crée aussi la liste des id avec leur actions (fonctions) associées.

```

Callback.register "b10" b10;;
Callback.register "b11" b11;;
Callback.register "b12" b12;;
Callback.register "b13" b13;;
(* ... les autres Callback.register ... *)

let liste_id =
  let l = ref [] in
  for i=1 to 5 do
    for j=0 to 3 do
      l := !l @ [((b"^(string_of_int(i))^^(string_of_int(j))), "command")]
    done;
  done;
  !l;;

let liste_actions =
  let l = ref [] in
  for i=1 to 5 do
    for j=0 to 3 do
      l := !l @ ["b"^(string_of_int(i))^^(string_of_int(j))]
    done;
  done;
  !l;;

```

A ce stade l'application est presque terminée. Il reste la dernière étape : le lancement du programme.

### 3.2.3 Initialisations : XPCOM et événements

Ceci est faisable par cette fonction :

```

let run() =
  try
    caml_initGTK 0 (Sys.argv);
    caml_initXPCOM();
    createXPCOM();
    openWindow ("file://" ^ Sys.argv.(1))
                "file,centerscreen";
    if isOpened () then
      (setActions liste_id liste_actions;
       addWindowEventListener "load";
       run();
       eraseXPCOM();
       caml_shutdownXPCOM())
    else
      caml_shutdownXPCOM()
  with Failure(s) ->
    print_string s;
    caml_shutdownXPCOM()
;;
run();;

```

### 3.2.4 Lancement du programme

Les composants XPCOM dépendent des bibliothèques partagées, on doit donc lancer notre exécutable avec le script de Mozilla.

```
run-mozilla.sh calc.exe
```

### 3.2.5 Marche à suivre : résumé

Pour finir avec cette application on résume ce que doit faire le programmeur :

- Définition de l'interface graphique
- Ecriture des fonctions de traitements en O'Caml et enregistrement
- Initialiser XPCOM et ajouter les événements pour les *écouter*
- Lancement du programme

# Conclusion

## Interprète

En conclusion, du point de vue de l’interprète, il reste à gérer les technologies CSS et DTD. Il serait particulièrement intéressant de gérer toute la hiérarchie de dossiers et de fichiers présentée dans la section 1.3 pour permettre à l’interprète d’exécuter correctement une application XUL faite pour XulRunner. Ceci permettrait aux programmes XUL d’être complètement interchangeables entre XulRunner, Mozilla/Firefox et notre interprète O’Xul.

De plus, il est important de corriger les problèmes de conversion de valeurs O’Caml vers JavaScript. Une toute autre approche peut être effectuée à l’aide de *Ocaml-templates*[8]. Ce système a l’avantage de permettre de générer automatiquement le code nécessaire en fonction du type attendu. Par exemple, nous pourrions utiliser *Ocaml-templates* au sein même du toplevel, en générant une chaîne à évaluer à l’aide de la chaîne “`type_attendu`”. Cela permettrait de générer des fonctions de création de l’objet `SpiderCaml.jsobj` du type adéquat directement dans le toplevel. Ensuite, il suffirait de créer une valeur dans le toplevel, à la manière de “`let ___expr___`”, qui serait l’objet `SpiderCaml.jsobj` créé à l’aide des fonctions générées par *Ocaml-templates*. L’avantage, c’est qu’il n’y aurait plus aucun problème de type `bool/int`, etc. puisque nous manipulerions directement des valeurs de type Caml dans le toplevel — et non pas des valeurs de type `Obj.t` — et que la valeur de type `Obj.t` qui serait récupérée par la suite dans l’interprète (par la valeur `___expr___`) serait *toujours* de type `SpiderCaml.jsobj`. En clair, cela simplifierait grandement les étapes puisque nous n’aurions plus à faire une analyse du type interne pour retrouver le type Caml caché derrière une valeur de type `Obj.t`.

De même, une des parties les plus importantes de cet interprète sera de définir entièrement le DOM de XUL à l’aide de *SpiderCaml* (voir section 2.2.2). Ceci constituant une grande charge de travail, il serait utile de pouvoir automatiser une partie des tâches de création, également à l’aide de *Ocaml-templates*, par exemple.

Enfin, une idée intéressante du projet serait d’étendre XUL en lui permettant d’utiliser du code O’Caml à la place du code JavaScript, par exemple en embarquant le toplevel d’O’Caml pour évaluer les lignes de code. Cependant, il faudrait permettre à notre interprète de savoir s’il est en présence de code O’Caml ou de code JavaScript. Il s’avère que XUL est déjà prêt pour cette extension grâce à l’attribut `type` de la balise `script` qui permet de préciser le langage utilisé pour les scripts. Par exemple, pour le JavaScript, la valeur de cet attribut est “`application/x-javascript`”. Cependant, même si cet attribut existe, actuellement XUL ne permet pas d’interpréter d’autres langages que le JavaScript. Ainsi, les applications XUL utilisant des scripts O’Caml ne seraient pas utilisables en dehors de notre interprète.

## XPCOM

Dans le cadre de l'étude de faisabilité, les choix d'implémentation ont permis de connaître la faisabilité des opérations d'interfaçage. XPCOM peut intégrer du code O'Caml dans ses composants. O'Caml peut utiliser tout l'API d'XPCOM à condition d'écrire du code en C/C++. L'exemple de la calculatrice montre ce fait. Plus généralement le problème qui se pose est l'encapsulation des méthodes C++ par des fonctions O'Caml. On a le choix entre soit générer du code O'Caml pour chaque interface et méthodes de l'interface, soit des fonctions généralisées à la manière de Javascript qui arrive à appeler les méthodes sans avoir les problèmes de type, de cast des *smart pointeurs* (les nsCOMPtr),... Le premier choix est plus simple mais cela engendrerait une bibliothèque énorme de fichiers O'Caml et C. Le deuxième est plus difficile à implanter mais plus efficace.

Cette étude s'est basée sur la couche basse des technologies Mozilla, à savoir XPCOM et C++. L'architecture XPCOM offre la possibilité de placer l'interfaçage à plusieurs niveaux. Un d'entre eux se situe au dessus d'XPConnect, le module *chargeur et lieur* pour Javascript.

Pour que l'intégration de O'Caml dans XPCOM et vice-versa soit complet, on doit écrire un traducteur à la manière des autres langages qui s'interfaçent avec XPCOM, qui permet à XPCOM de charger un composant écrit entièrement en O'Caml et à O'Caml d'utiliser XPCOM plus facilement. Ceci dépasse le cadre du projet mais enrichirait énormément la bibliothèque d'O'Caml.

Ce qui n'a pas été fait, faute de temps et non pas de faisabilité, est la mise en commun des deux parties. Fournir la manipulation du DOM en XPCOM/C++ à l'interprète XUL, demande à savoir si le pointeur sur la fenêtre récupérée est nul ou pas. D'autre part, GTK gère déjà les événements et ils ne se propageront peut être pas sur le DOM d'XPCOM.

Je conclus cette partie du projet par dire qu'il est difficile de travailler sans documentation. Mozilla ne fournit aucune documentation précise de son API (qui de plus change souvent). Le site Mozilla Developer Center<sup>1</sup> regroupe un ensemble de pages qui décrivent certains points, mais c'est un Wiki qui n'est pas complet.

---

<sup>1</sup><http://developer.mozilla.org/en/docs/XPCOM>

# Bibliographie

- [1] Vaughn Bullard, Kevin T. Smith, and Michael C. Daconta. Jxul. <http://jxul.sourceforge.net>.
- [2] Luxor. <http://luxor-xul.sourceforge.net>.
- [3] Nicolas Cannasse. Xml-light. <http://tech.motion-twin.com/doc/xml-light/>.
- [4] Alain Frisch. Spidercaml. <http://yquem.inria.fr/~frisch/SpiderCaml/doc/>.
- [5] Manoury P., Chailloux E., and Pagano B. *Développements d'applications avec Objective Caml*. O'Reilly, 2000. Chapitre 12, Interopérabilité avec C.
- [6] Capel C., Chailloux E., and Eber J.M. *Applications du toplevel embarqué d'Objective Caml*, 2004. Journées Francophones des Langages Applicatifs.
- [7] Chailloux E. and Henry G. *O'Jacaré, une interface objet entre Objective Caml et Java*, 2004. RSTI - L'objet.
- [8] François Maurel. Ocaml-templates. <http://www.pps.jussieu.fr/~maurel/programmation/>.
- [9] Deakin N. *Tutoriel Xul*. Tutoriel traduit par Xulfr.org.
- [10] Xavier Leroy. *The Objective Caml System*. <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>.
- [11] Gareta H. *Le langage et la bibliothèque C++ Norme ISO*. Ellipses, 2000.
- [12] Mozilla developer center. Build documentation. [http://developer.mozilla.org/en/docs/Build\\_Documentation](http://developer.mozilla.org/en/docs/Build_Documentation), Documentation sur la compilation des sources Mozilla.
- [13] Mozilla developer center. Creating xpcocomponents. [http://developer.mozilla.org/en/docs/Creating\\_XPCOM\\_Components](http://developer.mozilla.org/en/docs/Creating_XPCOM_Components), Tutoriel XPCOM et création d'un composant.
- [14] Xpcocom reference. <http://www.xulplanet.com/references/xpcocomref>, Site qui regroupe plus ou moins l'API des technologies Mozilla.
- [15] Wiki xulfr.org. <http://xulfr.org/wiki>, Site français sur XUL moins fourni que Mozilla Developer Center.
- [16] Mozillazine knowledge base. <http://kb.mozillazine.org/NsIWindowMediator>, Héberge une communauté de développeurs extérieurs à Mozilla.
- [17] Creating a c++ xpcocom component. <http://www.iosart.com/firefox/xpcocom>, Tutoriel pas à pas pour créer un XPCOM simple.
- [18] Alexis Seigneurin. Xpcocom. <http://igm.univ-mlv.fr/~dr/XPOSE2004/xpcocom>, Vue global de création d'un composant XPCOM.
- [19] Gtk+ reference manual. <http://developer.gnome.org/doc/API/2.0/gtk/index.html>.

- [20] SooHyung Oh. Gtk+ 2.0 tutorial using ocaml. <http://plus.kaist.ac.kr/~shoh/ocaml/labgtk2/labgtk2-tutorial/>.