Manuel d'utilisation

Jean-Yves Didier

20 août 2013

Résumé : ARCS pour Augmented Reality Component System est un système de programmation par composants destiné aux applications de réalité augmentée. Le but de ce présent document est de donner les spécifications de la nouvelle version de ce logiciel.

Table des matières

ARCS, pour *Augmented Reality Component Systems* est destiné au prototypage rapide des applications de réalité augmentée.

En quelques phrases clés, ARCS :

- est un *environnement de développement* (framework) : composé d'un ensemble de bibliothèques, d'une API documentée, de plusieurs programmes dont un moteur d'exécution d'application ;
- est *ouvert* et facilement extensible ;
- repose sur le paradigme de la *programmation orientée composants* ;
- propose un modèle d'applications *multi-processus* décrite par son *flux de données* ;
- exploite la communication entre les composants de manière *synchrone* ;
- permet de réaliser des mécanismes semblables à la *programmation orientée aspect*.

Ce document, destiné à être un manuel d'utilisation, décrit d'abord le modèle général d'application introduit par ARCS puis détaille les possibilités d'extension de ce dernier. Enfin, nous décrirons certains modules étendant ARCS.

1 A qui ARCS est-il destiné ?

ARCS est utilisable par deux catégories de personnes :

- les concepteurs d'applications de Réalité Augmentée ;
- les développeurs de composants pour les applications de RA.

2 Modèle d'une application

Les modèles évoqués ici sont surtout statiques. Il faudrait aussi évoquer les modèles comportementaux.

ARCS s'appuie sur un paradigme de programmation orientée composants.

ARCS propose un contexte d'exécution qui est constitué :

- d'un ensemble de *bibliothèques* contenant du code compilé étendant les fonctionnalités du moteur d'exécution ;
- d'un ensemble de *composants* ;
- d'un ensemble de *constantes* (des variables valuées) ;

Ce contexte est ensuite employé par l'application. Cette dernière est décrite en terme de *processus* concurrents. Un processus, au cours de son exécution, est amené à passer par différents états. La transition d'un état à un autre est pilotée par un *automate* (une machine à états finis) qui est appelé le *contrôleur*. Chaque état du processus correspond à une configuration particulière des communications entre les *composants*. Une description d'une telle configuration est appelée une *feuille*.

Nous allons à présent détailler les différents éléments constituant l'application.

2.1 Le composant

Un composant, selon la définition proposée par Szyperski :

- est un morceau de code compilé ;
- est sujet à composition avec d'autres composants ;
- possède des facultés d'introspection.

Le premier point fait débat : les récentes définitions des composants englobent aussi les scripts qui, par définition, ne sont pas compilés.

Slot d'un composant de type `A` :

```
void A::monSlotA()
{
    emit monSignal1();
    emit monSignal2();
}
```

Liste de connexions

```
a.monSignal1() --> b.monSlotB()
a.monSignal2() --> c.monSlotC()
```

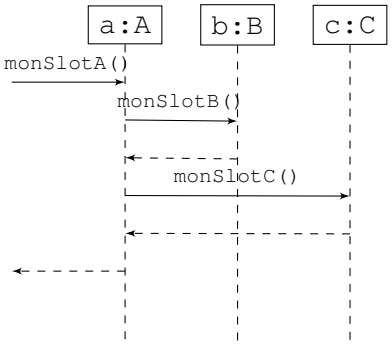


FIGURE 1 – La synchronicité

Sous ARCS, les composants sont décrits en termes d'entrées/sorties puisque ces derniers gèrent des flux de données. Les entrées des composants sont appelés des *slots* et les sorties des *signaux*. Composer deux composants revient alors à connecter un ou plusieurs signaux de l'un avec un ou plusieurs slots de l'autre. Ces communications sont considérées comme étant synchrones (voir figure 1). Ce modèle, fréquemment employé dans les bibliothèques de programmation d'interfaces graphiques permet d'effectuer des traitements effectués lorsqu'un évènement survient : frappe du clavier, clic de souris, etc. Il est généralement réalisé en employant le patron de conception *observateur* qui gère et maintient de manière sûre les connections.

Au niveau comportemental, dans ARCS, chaque composant aura la faculté de se décrire lui-même, c'est à dire d'exporter les listes des signaux et des slots qu'il gère. Cela va au delà de ce que Qt réalise parce qu'ARCS peut accepter non seulement des composants prenant comme base la classe `QObject` de Qt (ce que l'on appelle les composants natifs), mais aussi des composants dits exogènes (c'est à dire des composants que l'on peut se faire comporter comme si c'était des composants natifs). Un tel modèle de comportement est décrit dans la classe abstraite `ARCSAbstractComponent` qui est en réalité une fabrique de composants. Pour chaque composant reconnu par le moteur d'ARCS, les fonctionnalités suivantes sont implémentées (ou à implémenter) :

- l'export des listes de signaux et de slots ;
- la gestion des connections/déconnections ;
- l'initialisation des composants en utilisant leurs slots ;
- l'instanciation et la destruction des composants gérés ;
- la sérialisation/désérialisation sous la forme de chaîne de caractères et de fichiers ;
- un système de propriétés modifiables.

Un des objectifs d'ARCS est d'intégrer d'autres systèmes de composants que le sien propre. Pour cela, ARCS propose le modèle générique de composant vu ci-dessus qui est ensuite à transposer pour être utilisé avec les autres systèmes de composants (voir section 4.3 page 16). Le problème sous-jacent est celui de la réutilisabilité d'autres systèmes de composants sans pour autant avoir à redévelopper ces derniers.

2.2 Modèle général d'une feuille

Une feuille, nous l'avons dit, permet de décrire une configuration particulière des communications entre les composants. Ceci correspond à un état nominal d'un processus. A chaque état correspondent un certain nombre d'actions permettant de mettre en place, d'activer et enfin de nettoyer la configuration de communications souhaitée. Ces actions sont les suivantes :

- Réalisation des *invocations* de *préconnection* : elles permettent d'initialiser les composants avant

toute connection ;
- Mise en place des *connections* entre les composants. A l'issu de cette phase, les composants sont capables de communiquer entre eux ;
- Appel des invocations de *postconnection* qui accomplissent les actions associées à l'état nominal du processus actuel ;
- Lorsque les actions liées aux invocations de *postconnection* finissent ou débouchent sur un appel au contrôleur pour changer d'état, une phase de déconnection est réalisée ;
- Après la déconnection, une dernière phase dite de nettoyage (*cleanup*) permet d'effectuer certaines invocations pour mettre certains composants dans un état particulier.

ARCS propose un modèle d'application produites à partir de composants.

3 Installer ARCS

3.1 Obtenir ARCS

ARCS2 est disponible par le biais d'un dépôt *subversion* à l'adresse suivante :

```
https://evra.ibisc.univ-evry.fr/svn/ARCS2/
```

3.2 Pré-requis

3.2.1 Dépendances

Qt4 ou 5 [1] doit être installé.

3.2.2 Les variables d'environnement

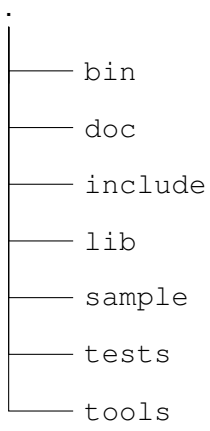
Avant de compiler ARCS, il convient de positionner la variable d'environnement `ARCSDIR` à l'endroit où se situent les sources de ARCS ainsi qu'ajouter quelques choses dans les variables d'environnement `PATH` et `LD_LIBRARY_PATH`. Il peut s'avérer pertinent de positionner de telles variables directement de le fichier `.bashrc` dont une partie se présentera sous la forme :

```
export ARCSDIR=/chemin/vers/ARCS/
export PATH=$PATH:$ARCSDIR/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ARCSDIR/lib
```

Remarque : Sous windows, il faudra positionner de la même manière des variables d'environnement. En particulier, `PATH` et `ARCSDIR` seront nécessaires.

1. http://qt-project.org

3.3 Organisation générale des sources



3.4 Compiler

Pour compiler, il ne reste plus qu'à se déplacer dans le répertoire des sources et de taper les commandes suivantes :

```
qmake
make
```

La documentation peut être générée (les pré-requis sont l'installation de doxygen et éventuellement de graphviz) en tapant :
`make docs`
La documentation de l'API sera alors disponible dans le sous-répertoire `doc/html`.

La compilation produit plusieurs bibliothèques ainsi que des exécutables assistant le développeur lors de la conception d'une application ou d'une bibliothèque de composants (voir table 1).

3.5 Vérifier le fonctionnement d'ARCS

Le sous-répertoire `sample` contient une bibliothèque permettant de tester le fonctionnement d'ARCS. Dans un premier temps, il faudra la compiler. Dans un deuxième temps, nous lancerons une ou plusieurs applications permettant de tester cette bibliothèque.

3.5.1 Compiler la bibliothèque de composants fournie en exemple

Les étapes sont les suivantes :

1. Se déplacer dans le sous-répertoire `sample` ;
2. Lancer l'utilitaire `arcslibmaker` : cela produira un fichier de projet ;
3. Générer le Makefile (sous unix) : `qmake` ;
4. Lancer la compilation de la bibliothèque : `make`.

A la fin du processus, une bibliothèque nommée `libsample.so` est alors générée.

Nom de l'exécutable	Rép.	Utilisation
`arcsengine`	`bin`	Moteur permettant de lancer une application ARCS en lui spécifiant un fichier de description XML
`arcslibmaker`	`bin`	Utilitaire permettant de créer un fichier comportant l'extension `.pro` pour une bibliothèque de composants pour ARCS.
`arcseditor`	`bin`	Editeur graphique pour créer des applications.
`arcswizard`	`bin`	Front-end graphique à `arcsengine`.
`arcsbuilder`	`bin`	A partir d'un répertoire dans lequel sont stockés les bibliothèques de composants et d'une description XML d'une application pour ARCS, lance la compilation des bibliothèques de composants nécessaires.
`arcs1to2`	`bin`	Utilitaire permettant le portage de bibliothèques de composants écrites à l'origine pour ARCS1 et portées vers ARCS2.

Nom de la bibliothèque	Rép.	Utilisation
`libarcs.so`	`lib`	Bibliothèque contenant le moteur d'ARCS.
`libarcsguiw.so`	`lib`	Bibliothèque annexe, chargée dynamiquement, si l'application nécessite d'utiliser des composants graphiques.
`arcs.dll`	`bin`	Bibliothèque contenant le moteur d'ARCS (pour windows uniquement).
`arcsguiw.dll`	`bin`	Bibliothèque annexe, chargée dynamiquement, si l'application nécessite d'utiliser des composants graphiques (pour windows uniquement).

TABLE 1 – Exécutables et bibliothèques produites par la compilation d'ARCS

3.5.2 Tester la bibliothèque de composants

Tester le fonctionnement de la bibliothèque de composants s'avère simple. Il faut se déplacer dans le répertoire `tests/xmlfiles` et lancer le moteur `arcsengine` en lui passant en paramètre le fichier XML nommé `loop.xml`.

Si tout se passe bien, le résultat devrait ressembler à ceci :

```
 $ arcsengine loop.xml
Application loop.xml loaded.
================================================
[INF!arcsapplicationcomponent.cpp:68] $ Rev: 207 $
[INF!arcsapplicationcomponent.cpp:35] needed to make "this"
[INF!Application mode] event
[Loop] Emitting iteration 0
[DInt] Received integer 0
[Loop] Emitting iteration 1
[DInt] Received integer 1
[Loop] Emitting iteration 2
[DInt] Received integer 2
[Loop] Emitting iteration 3
[DInt] Received integer 3
[Loop] Emitting iteration 4
```

```
[DInt] Received integer 4
[INF!arcsapplicationcomponent.cpp:93] main process has finished.
```

Pour plus d'information sur l'exécution, on pourra se reporter à l'annexe A.1 qui décrit les options que l'on peut fournir en ligne de commande au moteur d'exécution.

Notemment, on peut modifier le nombre d'itérations de la boucle de deux manières différentes :

```
arcsengine -d iterations=15 loop.xml
arcsengine -p profile_1.xml loop.xml
```

La première façon modifie une constante appelée *iterations* directement. La deuxième passe par un fichier de profil pour effectuer cette modification. Cela permet d'ajuster le fonctionnement du moteur directement à l'exécution sans rééditer le fichier.

3.5.3 Interprétation de l'application de test

Cette section explique ce qui s'est passé lors du lancement de l'application donnée en exemple. Si vous le souhaitez, vous pouvez donc aller plus loin directement si vous êtes familiers avec les concepts utilisés par ARCS.

Nous allons donc étudier le fichier XML utilisé pour l'application et donné dans le listing 1 page suivante. Les lignes 2 à 22 décrivent le contexte, c'est à dire les bibliothèques, composants et constantes utilisées par l'application. Les lignes 24 à 37 vont, quant à elles, décrire le fonctionnement de l'application.

A la ligne 4 est indiquée la bibliothèque de composants à charger. C'est cette bibliothèque qui contient, entre autres, les composants b de type *Loop* et d de type *DisplayInt* définis aux lignes 7 et 8. Du point de vue fonctionnel, b est une boucle qui démarre avec un certain nombre d'itérations (*setIterations(int)*), envoie un signal à chaque itération (*newIteration(int)*) et un signal en fin de boucle (*sendToken(QString)*). d se contente de recevoir une valeur entière et de l'afficher à la console (*display(int)*). Les lignes 10 à 16 définissent un composant particulier qui sera utilisé comme contrôleur pour un de nos processus. De manière simplifiée, ce composant décrit un automate à état finis avec un état initial nommé *start*, un état final nommé *end* et une transition entre les deux associée à un jeton nommé *end* (Voir figure 2). Le jeton est assigné via une fonction spéciale du composant nommée (*setToken(QString)*) et déclenche les transitions correspondantes. Enfin, la ligne 20 définit une constante nommée *iterations* qui prend une valeur entière et vaut 5 (c'est cette constante que l'on peut modifier en ajustant les paramètres du moteur).

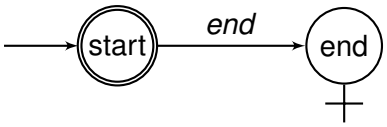


FIGURE 2 – Automate construit pour l'exemple

La partie qui décrit le fonctionnement de l'application met en évidence que cette dernière ne sera composée que d'un seul processus qui sera contrôlé par l'automate à état fini s que nous avons décrit ci-dessus (ligne 25). Ce processus est composé de deux feuilles : l'une nommée *start* et l'autre nommée *end*. Au démarrage du moteur, ce sera donc la feuille nommée *start* qui sera activée. Les actions réalisées seront l'établissement de connexions (lignes 28 à 30) entre b et d d'une part (dans le principe, chaque fois qu'une itération sera émise par b, d la recevra) et entre b et s d'autre part (cela permettra d'alimenter la machine à état s avec un jeton lorsque b aura fini sa boucle). Enfin, pour lancer la boucle, on lui donne un nombre d'itération fourni ici par le biais d'une constante [2] (lignes 32 et 33). Cela va donc déclencher la boucle avec, sauf altération, 5 itérations. Chaque itération sera émise par b, reçue par d qui va l'afficher. A la fin de la boucle, s recevra un jeton *end* de la part de de b. Cela va activer la transition entre les états *start* et *end* au niveau de l'automate et donc changer de feuille. On basculera sur la feuille *end* (ligne 35), qui étant vide, déclenchera la fin de l'application, puisque cela correspond à l'état final de l'automate.

2. Il est possible également de donner la valeur directement. L'utilisation de constantes est facultative.

```
<application mode="event">
  <context>
    <libraries>
      <library path="../../sample/sample"/>
    </libraries>
    <components>
      <component id="b" type="Loop" />
      <component id="d" type="DisplayInt" />
      <component id="s" type="StateMachine">
        <statemachine>
          <first name="start"/>
          <last name="end"/>
          <transitions>
            <transition source="start" token="end" destination="end"/>
          </transitions>
        </statemachine>
      </component>
    </components>
    <constants>
      <constant id="iterations" type="int">5</constant>
    </constants>
  </context>

  <processes>
    <process controller="s">
      <sheet id="start">
        <connections>
          <link source="b" signal="newIteration(int)" destination="d"
              slot="display(int)"/>
          <link source="b" signal="sendToken(QString)" destination="s"
              slot="setToken(QString)" />
        </connections>
        <postconnections>
          <invoke destination="b" slot="setIterations(int)"
              type="constant">iterations</invoke>
        </postconnections>
      </sheet>
      <sheet id="end"/>
    </process>
  </processes>
</application>
```

Listing 1 – Description de l'application donnée en exemple

4 Étendre le moteur d'ARCS

4.1 Créer une bibliothèque de composants pour ARCS

Dans un premier temps, nous nous intéresserons au système de composants natifs dans ARCS (les composants pris en charge directement par ARCS). Cette section a pour objectif de décrire comment on écrit des composants pour ARCS, puis comment on entrepose ces derniers dans des bibliothèques.

Pour commencer, ARCS utilisant de manière intensive Qt, la description des composant va être calquée sur la manière de décrire des `QObject` utilisant des signaux et des slots personnalisés [3]. Nous allons donner les étapes qui ont présidées à la création de la bibliothèque de composants `sample` qui est fournie avec le moteur.

4.1.1 Conception des composants

Nous allons écrire deux composants. Le premier est une boucle (*Loop*) qui prendra en paramètre un certain nombre d'itérations, puis qui, sur un top de départ va réaliser ces itérations. Chaque itérations envera un signal indiquant l'itération en cours. Enfin le composant émettra un signal final particulier indiquant au processus en cours que le travail est terminé. Le deuxième composant, beaucoup plus basique, va réceptionner une valeur entière et l'afficher à la console (*DisplayInt*). En substance, le code déclarant ces composants aura l'aspect de celui décrit dans le listing 2 page suivante. La réalisation de ces interfaces ne posera pas de difficulté particulières, comme on peut le voir dans le listing 3 page suivante. Par la suite, nous supposerons que la déclaration des composants est effectuée dans le fichier *sample.h* et l'implémentation correspondante dans *sample.cpp*.

4.1.2 Conception de la bibliothèque de composants

Une fois les fichiers sources créés, la création d'une bibliothèque de composants ne posera pas de difficulté majeure. Il s'agira tout d'abord de lancer une première fois l'utilitaire `arcslibmaker`. Si les fichiers étaient dans un répertoire nommé sample, l'utilitaire créera alors un projet pour Qt (nommé *sample.pro*) incluant les fichiers de source disponibles dans le répertoire et des cibles supplémentaires. De plus, un fichier nommé `libsample.alx` sera également créé et sera à compléter.

Il est à noter que le fichier de projet peut-être ultérieurement modifié pour inclure des bibliothèques externes nécessaires à la compilation des composants [4].

Dans un premier temps, nous allons nous intéresser au fichier `libsample.alx` qui est un fichier au format XML décrivant à l'aide de quelques balises une bibliothèque de composants pour Qt (pour avoir le détail des balises, se reporter à l'annexe B.1.3 page 26). Dans ce dernier, deux sections vont nous intéresser tout d'abord :

- la section *headers* dans laquelle on va préciser les fichiers d'en-tête contenant les déclarations de composants ;
- la section *components* dans laquelle on indiquera les composants que la bibliothèque exportera.

Notre bibliothèque comportant deux composants de type *Loop* et *DisplayInt* déclarés dans un fichier d'en-tête nommé *sample.h*, le fichier de configuration de la bibliothèque sera alors conforme à ce qui est présenté dans le listing 5 page 13.

3. Pour de plus amples informations sur le système entourant la classe *QObject* de Qt, nous invitons le lecteur à se reporter à la documentation de Qt, en particulier la partie concernant les signaux et les slots : `http://qt-project.org/doc/qt-5.1/qtcore/signalsandslots.html`

4. Pour une compréhension approfondie de la syntaxe du fichier, on peut se reporter à la documentation de l'utilitaire

```
#include <QObject>

class Loop : public QObject // héritage à QObject nécessaire
{
  Q_OBJECT // Obligatoire pour personnaliser ses signaux et ses slots
  public:
    // Constructeur à la signature imposée.
    Loop(QObject* parent=0) : QObject(parent) {}

  public slots: // début de la section de déclaration des slots
    void setIterations(int n); // donne le nombre d'itérations à faire

  signals:      // début de la section de déclaration des signaux
    void newIteration(int i);  // signal émis pour chaque nouvelle itération
    void sendToken(QString s); // signal spécial de fin d'application
};

class DisplayInt : public QObject
{
  Q_OBJECT
  public:
    DisplayInt(QObject* parent=0) : QObject(parent) {}

  public slots:
    void display(int i);       // slot affichant un nombre entier passé en entrée
};
```

Listing 2 – Déclaration des composants *Loop* et *DisplayInt* dans le fichier *sample.h*

```
#include "sample.h" // en-tête où les composants sont déclarés
#include <iostream>

void Loop::setIterations(int n)
{
    for (int i=0; i < n; i++)
    {
      std::cout << "[Loop] Emitting iteration " << i << std::endl;
      emit newIteration(i); // emit déclanche l'émission du signal correspondant
    }
    emit sendToken("end");
}

void DisplayInt::display(int i)
{
  std::cout << "[DInt] Received integer " << i << std::endl;
}
```

Listing 3 – Implémentation des composants *Loop* et *DisplayInt* dans le fichier *sample.cpp*

```
win32-msvc* {
    TEMPLATE = vclib
  } else {
    TEMPLATE = lib
}
INCLUDEPATH += $$(ARCSDIR)/include
LIBS += -L$$(ARCSDIR)/lib -larcs
CONFIG += dll
QT = core

# fichiers de source à compiler dans le projet
HEADERS += sample.h
SOURCES += sample.cpp

# règles spécifiques de génération pour ARCS
ALXFILE = libsample.alx
OTHER_FILES += libsample.alx
arcslibrary.output = alm_${QMAKE_FILE_BASE}.cpp
arcslibrary.input = ALXFILE
arcslibrary.commands = arcslibmaker ${QMAKE_FILE_NAME}
arcslibrary.variable_out = SOURCES
QMAKE_EXTRA_COMPILERS += arcslibrary
```

Listing 4 – Fichier de projet généré par `arcslibmaker`

```
<library>
  <headers>
    <header name="sample.h"/>
  </headers>
  <components>
    <component name="Loop"/>
    <component name="DisplayInt"/>
  </components>
</library>
```

Listing 5 – Fichier XML décrivant la bibliothèque d'exemple (*libsample.alx*)

Pour comprendre ce qui est réalisé, grâce aux régles mises dans le fichier *sample.pro*, ce fichier est lu par `arcslibmaker` et va générer automatiquement le code permettant de charger les composants dans un fichier nommé *alm_libsample.cpp*[5] comme on peut le voir au listing 6.

```
#include<arcs/arcslibtoolkit.h>
#include<QMetaType>
#includ.e<sample.h>

extern "C" DLL_EXPORT void arcsLibraryRegister(ARCSComponentMap* cmap,
    ARCSFamilyMap* , ARCSTypeMap* )
{
  cmap->insert("Loop",new ARCSNativeComponentTemplate<Loop>());
  cmap->insert("DisplayInt",new ARCSNativeComponentTemplate<DisplayInt>());
}

extern "C" DLL_EXPORT void arcsLibraryUnregister(ARCSComponentMap* cmap,
    ARCSFamilyMap* , ARCSTypeMap* )
{
  delete cmap->take("Loop");
  delete cmap->take("DisplayInt");
}
```

Listing 6 – Fichier généré par `arcslibmaker` à partir de la description XML de la bibliothèque

4.1.3 Compiler la bibliothèque

La bibliothèque de composant se compilera suivant le mode à présent classique. Un premier appel à `qmake` suivi d'un appel à `make` sous unix.

En résumé, la **démarche générale pour créer une bibliothèque de composants natifs** est la suivante :

1. Préparer les fichiers source des composants ;
2. Lancer `arcslibmaker` ;
3. Modifier le fichier XML produit par `arcslibmaker` pour indiquer quels sont les composants exportés par la bibliothèque ;
4. Lancer l'utilitaire `qmake` pour produire un makefile sous Unix ;
5. Compiler la bibliothèque en utilisant `make`.

4.2 Introduire un nouveau type de données

Introduire un nouveau type de données permet essentiellement de nouvelles manières d'initialiser les composants dans ARCS. En réalité, on indiquera comment sérialiser ces types et leur nom. Tout reposera sur la dérivation de la classe *ARCSTypeFactoryTemplate<...>*. La classe qui en dérive devra absolument porter un nom calqué sur le schéma suivant de manière à être traitée convenablement par le moteur d'ARCS : `ARCSTypeFactory_type` et aura une structure ressemblant à celle proposé au listing 7 page suivante.

`qmake` fourni avec Qt : `http://qt-project.org/doc/qt-5.1/qtdoc/qmake-manual.html`

5. Là encore, si nécessaire, on peut modifier ce fichier et invalider les règles le concernant dans le fichier de projet.

```
#include <arcs/arcslibtoolkit.h>

class ARCSTypeFactoryTemplate_MyNewType : public ARCSTypeFactoryTemplate<MyNewType>
{
  public:
    virtual QString getTypeName() const {
      // donne le nom du type pour le moteur d'ARCS
    }
  protected:
    virtual MyNewType parse(QString s) {
      // retourne les données construites à partir de la chaîne s
    }
    virtual QString serialize(MyNewType mnt) {
      // retourne une chaîne sérialisant les données de mnt
    }
};
```

Listing 7 – Squelette général d'une classe introduisant un nouveau type

```
#include <arcs/arcslibtoolkit.h>
#include <QSize>

class ARCSTypeFactory_QSize : public ARCSTypeFactoryTemplate<QSize>
{
  public:
    virtual QString getTypeName() const { return "size"; }
  protected:
    virtual QSize parse(QString s) ;
    virtual QString serialize(QSize s);
};
```

Listing 8 – Déclaration d'une classe introduisant le type *size*

```
#include <typesize.h>
#include <QStringList>

QSize ARCSTypeFactory_QSize::parse(QString s)
{
     QStringList sl = s.split("x");
     if (sl.count() != 2) return QSize();
     return QSize(sl[0].toInt(),sl[1].toInt());
}

QString ARCSTypeFactory_QSize::serialize(QSize s) {
     return QString::number(s.width()) + "x" + QString::number(s.height());
}
```

Listing 9 – Réalisation de la classe introduisant le type *size*

Exemple d'introduction d'un nouveau type de données

Pour illuster cela, nous allons voir comment le type nommé *size* dans le moteur d'ARCS a été implémenté. A la base, une classe de Qt stocke les tailles (couple d'entiers donnant la largeur et la hauteur d'un objet) dans une classe de type *QSize*. Logiquement, la classe à déclarer a pour nom *ARCSTypeFactory_QSize* comme on peut le voir au listing 8 page précédente où l'on suppose que la déclaration est effectuée dans un fichier nommé *typesize.h*. Le type proposéest connu par le moteur sous le nom de *size* et nous avons donc deux méthodes à implémenter : `parse()` et `serialize()`. Elles sont réalisées au listing 9 page précédente où elles font l'aller-retour entre un objet de type *QSize* et une chaîne de caractères de la forme *largeur*×*hauteur*.

Le nouveau type doit également être mis à disposition du moteur ; cela se fait en modifiant le fichier de type *.alx* qui est le point de passage obligé pour toute bibliothèque étendant le fonctionnement d'ARCS. Le listing 10 montre comment cela est réalisé en pratique. Il est à noter que le type indiqué est le véritable type de l'objet et non pas celui donné à ARCS.

```
<library>
    <headers>
        <header name="typesize.h"/>
    </headers>
    <types>
        <type name="QSize"/>
    </types>
</library>
```

Listing 10 – Fichier minimal de description de la bibliothèque contenant le type *size*

Cela permet ensuite de par exemple déclarer des constantes dans une description d'application :

```
<constant id="taille_min" type="size">320x200</constant>
```

4.3 Introduire dans ARCS un système de composants exogène

ARCS peut intégrer un système de composants exogènes. L'objectif est, dans ce cas, d'intégrer ces composants de manière à ce que ces derniers puissent interagir avec les composants natifs (ou endogènes) d'ARCS. Cela va passer par des techniques de programmation particulières et l'utilisation intensive de quelques classes de l'API d'ARCS.

4.3.1 Classes dédiées à l'intégration de familles de composants exogènes

Deux classes sont importantes pour intégrer une famille de composants exogènes : *ARCSAbstractFamily* et *ARCSAbstractComponent*. Dans le principe, l'ensemble est architecturé en employant le patron de conception « Fabrique abstraite »(*abstract factory*).

ARCSAbstractFamily est la classe dont on doit dériver pour créer une famille de composants. En réalité, il s'agit enregistrer un ensemble de fabriques (*factories*) de composants pour pouvoir les instancier. En conséquence, comme on peut le voir sur le squelette de classe proposé au listing 11 page suivante, on trouvera plusieurs opérations spécifiques à implémenter concernant la gestion des fabriques (`factoryList()`, `addFactory()` et `removeFactory()`), l'instanciation des composants (`instanciate()` et `destroy()`) et enfin les propriétés propres à la famille (`name()` et `isInternal()`).

```
#include <arcs/arcsabstractfamily.h>

class MyFamily : public ARCSAbstractFamily
{
public:
  MyFamily();
  virtual ~MyFamily() {}

  // retourne la liste de factories gérées par cette famille.
  virtual QStringList factoryList();
  // ajoute une factory de composant pour cette famille.
  virtual bool addFactory(QString type, ARCSAbstractComponent* cmp);
  // retire une factory de composants de la famille
  virtual void removeFactory(QString type);
  // instancie un composant étant donné le nom de son type
  virtual ARCSAbstractComponent* instanciate(QString type);
  // détruit un composant connaissant son pointeur.
  virtual void destroy(ARCSAbstractComponent* component);
  // retourne le nom de la famille
  virtual QString name() const;
};
```

Listing 11 – Squelette d'une implémentation d'une nouvelle famille

Pour compléter l'architecture, *ARCSAbstractComponent* est la classe doit on doit dériver pour créer les fabriques de composants proprement dites. La nouvelle classe doit décrire les composants de manière à ce que ceux ci soient manipulables par le moteur d'ARCS comme s'ils étaient des composants natifs. Le listing 12 page suivante montre le squelette d'une classe qui permettrait d'intégrer un nouveau type de fabrique de composants. La classe couvre plusieurs aspects du comportement de la fabrique :

- La sérialisation/désérialisation d'un composant à partir d'une chaîne de caractères (`toString()`, `parseString()`) ou à partir d'un fichier (`loadFile()`, `saveFile()` – bien que ce dernier comportement n'est à redéfinir que de manière facultative) ;
- La description de l'interface du composant via les méthodes `getSignals()` et `getSlots()` ;
- La création ou la récupération d'intermédiaire fonctionnant comme des composants natifs d'ARCS (c'est à dire basés sur la classe `QObject` de Qt) afin de permettre la communication entre les composants d'ARCS et la nouvelle famille de composants (`getProxySlot()` et `getProxySignal()`);
- L'établissement et la rupture d'une connection lorsque les composants sont de même type (`genuineConnect()` et `genuineDisconnect()`). Cela permet de passer par le système de connection de la famille exogène de composants et évite la génération d'intermédiaires à chaque extrémité de la connection de manière à optimiser les performances). Si cela n'est pas nécessaire (connexion ayant des *QObjects* aux deux bouts) alors les méthodes n'ont pas besoin d'être réimplémentées. Il est à noter que si la connection n'a pu être établie ou détruite, la méthode doit retourner *false* (c'est que fait l'implémentation par défaut) ;
- La gestion de l'instance réelle du composant (`getGenuineComponentInstance()`, `genuineInstanciate()` et `genuineDestroy()`).

L'intégration de la nouvelle famille de composants passe par l'inclusion de l'en-tête correspondant à la définition de la nouvelle famille dans le fichier XML décrivant la bibliothèque de composants ainsi que de l'inclusion du nom de la nouvelle famille dans une balise de type `family`. Pour finir, si l'on souhaite

```
#include <arcs/arcsabstractcomponent.h>

class MyComponent : public ARCSAbstractComponent
{
public:
  MyComponent();
  ~MyComponent();

  // sérialisation du composant en chaîne de caractères
  virtual QString toString();
  // instanciation en interne du composant à partir d'une chaîne
  virtual bool parseString(QString s);
  // récupération de la liste de signaux
  virtual QStringList getSignals();
  // récupération de la liste de slots
  virtual QStringList getSlots();
  // lecture du composant à partir d'un fichier (optionnel)
  virtual bool loadFile(QString fn);
  // écriture du composant dans un fichier (optionnel)
  virtual bool saveFile(QString fn);

protected:
  // récupération d'une liste de QObject faisant office de proxys pour les slots
  virtual void getProxySlot(
    QString slot, ObjectList &obj, QStringList &proxySlot);
  // récupération d'une liste de QObject faisant office de proxys pour les signaux
  virtual void getProxySignal(
    QString signal, ObjectList &obj, QStringList &proxySignal);
  // connection réelle entre deux composants du même type (optionnel).
  virtual bool genuineConnect(
    QString sig, ARCSAbstractComponent *dst, QString slt, bool queued=false);
  // déconnection réelle entre deux composants du même type (optionnel)
  virtual bool genuineDisconnect(
    QString sig, ARCSAbstractComponent *dst, QString slt);
  // procédure d'instanciation réelle du composant
  virtual bool genuineInstanciate();
  // procédure de destruction réelle du composant
  virtual void genuineDestroy();
  // récupération de l'instance réelle du composant
  virtual QVariant getGenuineComponentInstance();
};
```

Listing 12 – Squelette d'une implémentation d'une fabrique de composants

initialiser un composant avec un autre composant, il faut faire en sorte que Qt connaisse le nouveau type de composant par une ligne de code comme suit : `Q_DECLARE_METATYPE(MyComponent)`.

4.3.2 Exemple d'implémentation d'une famille exogène

Pour illustrer l'utilisation d'une famille exogène, nous allons montrer comment cela peut-être réalisé. A l'aide du *framework* Qt, il est possible de créer des interfaces graphiques. Ces dernières sont sauvegardée dans un fichier portant l'extension *.ui* dans lequel les données sont dans un format XML. L'API de la bibliothèque de Qt permet de charger ces fichiers pour les exploiter et générer à la volée l'interface graphique. Nous allons voir comment nous pouvons transformer ceci en un composant pleinement exploitable par Qt.

Les étapes vont donc être les suivantes :

- Dériver *ARCSAbstractFamily* ;
- Puis faire de même pour *ARCSAbstractComponent* ;
- Enfin créer la bibliothèque associée.

Création de la famille

Nous allons créer une classe *UiFamily* dérivant de *ARCSAbstractFamily*. L'objectif est donc d'enregistrer la fabrique et de lui associer un nom. *UiFamily* servira également de classe médiatrice pour créer les composants.

```
1  #include <arcs/arcsabstractfamily.h>
2
3  class UiFamily : public ARCSAbstractFamily
4  {
5    public:
6      UiFamily() {}
7      virtual ~UiFamily() {}
8
9      virtual QStringList factoryList() { return QStringList(QString("Ui")); }
10     virtual bool addFactory(QString, ARCSAbstractComponent*) { return false; }
11     virtual void removeFactory(QString type) {}
12     virtual ARCSAbstractComponent* instanciate(QString type);
13     virtual void destroy(ARCSAbstractComponent* component);
14     virtual QString name() const { return "UiFamily"; }
15 };
```

Listing 13 – Déclaration de la famille *UiFamily*

Comme on peut le voir au listing 13, la classe dérivée est peu complexe à déclarer. Un certain nombre de méthodes sont directement implémentées en ligne dans la déclaration. Le constructeur et le destructeur n'ont pas de chose particulière à mettre en place. Les méthodes `addFactory()` et `removeFactory()` permettant d'ajouter et supprimer à la volée des fabriques n'ont pas leur place. La seule fabrique à enregistrer a pour nom `Ui` et le nom de la famille est `UiFamily` (cf lignes 9 et 14). Il ne reste donc plus qu'à implémenter les méthodes `instanciate()` et `destroy()`.

Cela est réalisé au listing 14 page suivante. Les lignes 4 à 8 montrent comment l'instanciation d'un composant est réalisée : on fait une vérification sommaire du nom de fabrique proposé et, s'il est correct, on créé le composant associé. Pour la destruction, les lignes 10 à 18 vérification à nouveau le nom de la

```
1  #include "uifamily.h"
   #include "uicomponent.h"
3
   ARCSAbstractComponent* UiFamily::instanciate(QString type)
5  {
       if (type == "Ui")
7          return new UiComponent();
   }
9
   void UiFamily::destroy(ARCSAbstractComponent *component)
11 {
       if (component->getType() == "Ui" )
13     {
           UiComponent* uicmp = dynamic_cast<UiComponent*>(component);
15         if(uicmp)
               delete uicmp;
17     }
   }
```

Listing 14 – Réalisation de la famille *UiFamily*

fabrique. Si ce dernier est correct, alors on fait une conversion de pointeur de *ARCSAbstractComponent* vers *UiComponent* et, dans le cas où cela réussit, on détruit le composant associé.

Création de la fabrique de composants associée

La fabrique de composant devra dériver la classe *ARCSAbstractComponent* et devra prendre en charge diverses choses. Le listing 15 page suivante donne la déclaration du composant. Dans le principe, une interface graphique générée par les outils de Qt représente un ensemble de *widgets* nommés qui vont posséder, puisqu'ils font partie de l'API de Qt, des signaux et des slots. L'idée est donc : d'une part de pouvoir lire le fichier à l'extension *.ui* et générer le composant correspondant, d'autre part de pouvoir exposer les signaux et les slots des *widgets* internes à l'interface (nous rajouterons une contrainte : de manière à ne pas trop surcharger la liste des signaux et des slots, les *widgets* dont le nom commence par un caractère « _ » ne seront pas traités).

Comme précédemment, un certain nombre de méthode sont directement implémentées en ligne. D'autres méthodes vont bénéficier d'une implémentation plus complète qu'il convient d'examiner. Les listings 16 page 22 et 17 page 23 montrent les instructions qui les composent. Le listing 16 détaille en particulier la méthode pour charger et instancier le composant à partir de sa description textuelle. Cela permet également de construire la liste des signaux et des slots du composant. Dans le principe, une fois la chaîne de caractères disponible, on instancie le composant (lignes 11 à 17) pour pouvoir appliquer des méthodes d'introspection sur ce dernier. C'est ce qui explique que la méthode `genuineInstanciate()` se contente de retourner un pointeur vers le composant déjà existant. Les lignes 22 et 23 regardent, pour l'interface chargée, quels sont les *widgets* dont le nom ne commence pas par « _ » et lance la fonction auxiliaire `processWidget()` pour déterminer les signaux et les slots de ces *widgets*. Les signaux et les slots ainsi stockés porteront soit le nom des signaux et slots du *widget* principal de l'interface, soit auront pour nom une signature calquée sur le schéma suivant : *nomComposant.signatureSignal/slot(...)*. Les structures `signalMap` et `slotMap` sont en réalité des tables de hashage qui associe ce nom à une paire étant composé d'un pointeur vers le *widget* correspondant et une chaîne donnant la signature du signal ou du slot qui y est associé.

```
#include <arcs/arcsabstractcomponent.h>
#include <QPair>
#include <QWidget>

class UiComponent : public ARCSAbstractComponent
{
  public:
    UiComponent() { actualComponent = 0; }
    ~UiComponent() { if (actualComponent) delete actualComponent; }

    // méthodes de sérialisation / désérialisation
    virtual QString toString() { return actualData ; }
    virtual bool parseString(QString s);
    // liste des signaux et des slots du composant
    virtual QStringList getSignals() { return signalMap.keys(); }
    virtual QStringList getSlots() { return slotMap.keys(); }

  protected:
    // récupération des QObject faisant office de proxy
    virtual void getProxySlot (
      QString slot, ObjectList &obj, QStringList &proxySlot);
    virtual void getProxySignal (
      QString signal, ObjectList &obj, QStringList &proxySignal);
    // gestion de l'instance du composant
    virtual bool genuineInstanciate () { return (actualComponent) ; }
    virtual void genuineDestroy ()
      { delete actualComponent ; actualComponent = NULL; }
    virtual QVariant getGenuineComponentInstance() { return QVariant(); }

  private:
    // fonction auxiliaire de construction des listes de signaux et slots
    void processWidget(QWidget* widget,QString prefix=QString::null);
    // représentation de l'interface sous forme de chaîne de caractères
    QString actualData;
    // liste des signaux du composant
    QHash<QString,QPair<QString,QWidget*> > signalMap ;
    // liste des slots du composant
    QHash<QString,QPair<QString,QWidget*> > slotMap ;
    // pointeur vers une instance du véritable composant instancié
    QWidget* actualComponent ;
};
```

Listing 15 – Déclaration de la fabrique de composants *UiComponent*

```cpp
#include "uicomponent.h"
#include <QBuffer>
#include <QUiLoader>
#include <QBuffer>
#include <QList>
#include <QMetaMethod>

bool UiComponent::parseString(QString s)
{
  // prépare le chargement
  QUiLoader loader;
  QByteArray array= s.toUtf8();
  QBuffer buffer(&array);
  buffer.open(QIODevice::ReadOnly);
  // charge le composant dans actualComponent
  // et sauve la chaîne s dans actualData pour un usage ultérieur
  actualComponent = loader.load(&buffer);
  actualData = s ;
  // en cas d'échec, sortir en l'indiquant
  if (!actualComponent) return false;
  // préparer la liste de signaux, de slots et des QObject qui y correspondent
  QList<QWidget*> widgets =
      actualComponent->findChildren<QWidget*>(QRegExp(QString("^[^_].*")));
  processWidget(actualComponent);
  for(int i=0; i < widgets.count(); i++)
    processWidget(widgets[i],(widgets[i]->objectName().isEmpty())?
      (QString::null):(widgets[i]->objectName()+"."));
  return true;
}

void UiComponent::processWidget(QWidget *widget,QString prefix)
{
  // itérer au travers du widget passé en paramètre, et récupération
  // des signaux et des slots.
  for (int i=0; i < widget->metaObject()->methodCount(); i++)
  {
    QMetaMethod method = widget->metaObject()->method(i);
    if (method.methodType()==QMetaMethod::Signal ) {
      signalMap[prefix+method.signature()] =
          QPair<QString,QWidget*>(method.signature(),widget);
    }
    if (method.methodType()==QMetaMethod::Slot) {
      slotMap[prefix+method.signature()] =
          QPair<QString,QWidget*>(method.signature(),widget);
    }
  }
}
```

Listing 16 – Réalisation de la fabrique de composants *UiComponent* (partie 1)

```
void UiComponent::getProxySlot(QString slot, ObjectList &obj, QStringList
    &proxySlot)
{
  if (slotMap.contains(slot)) {
    proxySlot << slotMap[slot].first;
    obj << slotMap[slot].second ;
  }
}

void UiComponent::getProxySignal(QString signal, ObjectList &obj, QStringList
    &proxySignal)
{
  if (signalMap.contains(signal)) {
    proxySignal << signalMap[signal].first ;
    obj << signalMap[signal].second ;
  }
}
```

Listing 17 – Réalisation de la fabrique de composants *UiComponent* (partie 2)

Le listing 17 indique les *QObject* qui seront utilisés en tant que proxy entre les signaux et les slots exposés et les véritables signaux et slots du composant. Le principe est que la méthode doit indiquer les objets de type *QObject* qui correspondent au signal ou au slot indiqué (paramètre *signal* ou paramètre *slot*) ainsi que les véritables signaux et slots utilisés sur ces objets. Dans notre cas, à un signal ou slot correspondra un objet de type *QObject* et un signal ou slot. Le tout étant stocké dans les variables `signalMap` et `slotMap`, il suffit de passer les valeurs stokées dedans.

Créer la bibliothèque de composants correspondante Dans le fichier à l'extension *.alx* de la bibliothèque, il faudra rajouter deux lignes : une indiquant l'en-tête définissant la famille et la deuxième indiquant le nom de la famille chargée. Le listing 18 donne le contenu minimal du fichier décrivant la bibliothèque permettant d'étendre le fonctionnement du moteur et introduisant l'utilisation des fichiers *.ui* en tant que composants pour ARCS.

```
<library>
    <headers>
        <header name="uifamily.h"/>
    </headers>
    <families>
        <family name="UiFamily"/>
    </families>
</library>
```

Listing 18 – Fichier minimal de description de la bibliothèque contenant *UiFamily*

Pour compiler la bibliothèque ou pour avoir plus de détails sur cette dernière, on se reportera à la section 4.1 page 11 de ce manuel.

4.4 Créer des composants génériques

5 Ce qu'il reste à faire

Ici sont mis en vrac tous les éléments qui sont à valider pour valider ARCS2 :
- vérification sur les instanciations/destructions ;
- l'interface graphique.

A Les outils

A.1 Le moteur d'exécution *arcsengine*

A.1.1 Usage

```
arcsengine [OPTION]...  [XML_FILE]...

  -h,  --help             : print this help

Overriding application mode :

  -b,  --mode-base        : simple loop based applications.
  -e,  --mode-event       : event loop based console applications.
  -g,  --mode-gui         : event loop based GUI applications.
  -t,  --mode-thread      : threaded application.
  -te, --mode-thread-event: threaded event based application.

Defining options:
  -d, --define vars       : define constants
  -p, --profile file      : define a profile
  -o, --profile-out       : define a file where to dump profile
```

A.2 Le configurateur de bibliothèques *arcslibmaker*

A.2.1 Usage

```
arcslibmaker [--help] [file]
arcslibmaker has two modes, one for generating ARCS library wrappers,
 the second for adding ARCS options to Qt project files.

The first mode needs an xml file describing the library contents.
```

B Formats XML

ARCS s'appuie sur un certain nombre de fichiers propres encodés en XML. Ces derniers permettent de décrire des applications, des composants composites, des bibliothèques d'extension. Nous allons passer en revue l'ensemble des ces arborescences XML.

B.1 Les structures principales

B.1.1 Les applications

La figure 3 donne la hiérarchie des balises composant une application. Cette dernière est composée, à haut niveau, d'un contexte (cf annexe B.2.1) et d'une liste de processus, eux-même contenant une liste de feuilles (cf annexe B.2.2). La table 2 donne la description de ces balises.

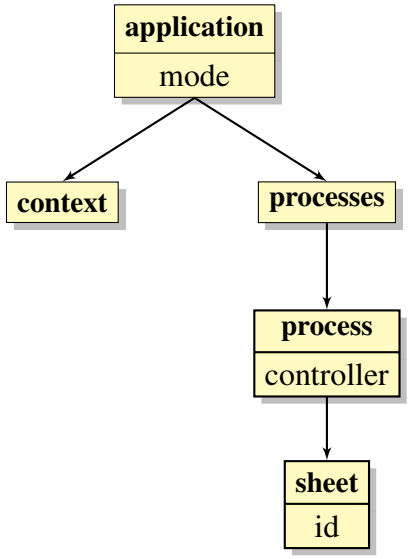


FIGURE 3 – Balises XML décrivant une application

B.1.2 Les composants composites

La figure 4 expose l'organisation des balises XML décrivant un composant composite (un composant formé par l'aggrégation de composants). Un composant composite dispose de son propre contexte (cf annexe B.2.1), d'une feuille (cf annex B.2.2) décrivant son architecture interne, ainsi que d'une interface exposant les slots et les signaux du composant composite (ses entrées-sorties). La table 3 donne la signification associée aux balises.

B.1.3 Description de bibliothèque

La figure 5 donne l'organisation des balises XML composant les fichiers décrivant les bibliothèques étendant le moteur d'ARCS. Ce fichier est lu par le configurateur de bibliothèques (cf annexe A.2) en vue de générer du code C++ utilisé pour créer les points d'entrée de la bibliothèque. Ce fichier donne essentiellement la liste des fichiers d'en-tête à utiliser, ainsi que les nouvelles familles de composants développées, les composants natifs proposés, et les nouveaux types accessibles pour le moteur. La table 4 donne la signification associée aux balises.

Balise	Description	
application	Déclare une application	
	Enfants	*context* (requis, cf B.2.1), *processes* (requis)
	Attributs *mode*	permet de lancer l'application suivant 5 modes différents : boucle de base (`base`), boucle d'évènements (`event`), thread à part (`thread`), thread avec boucle d'évènements (`threadevent`) et enfin support de l'interface graphique (`gui`). Par défaut, la valeur est `base`.
processes	Donne la liste des processus composant une application	
	Enfants	*process* (requis)
process	Déclare un processus	
	Enfants	*sheet* (requis, cf B.2.2)
	Attributs *controller*	Requis - référence à l'identifiant du composant jouant le rôle de contrôleur

TABLE 2 – Description des balises composant une application

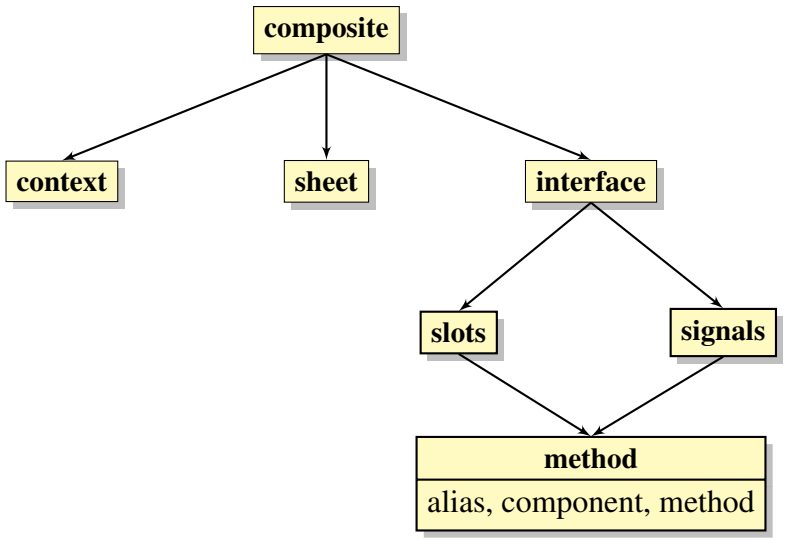


FIGURE 4 – Balises XML représentant un composant composite

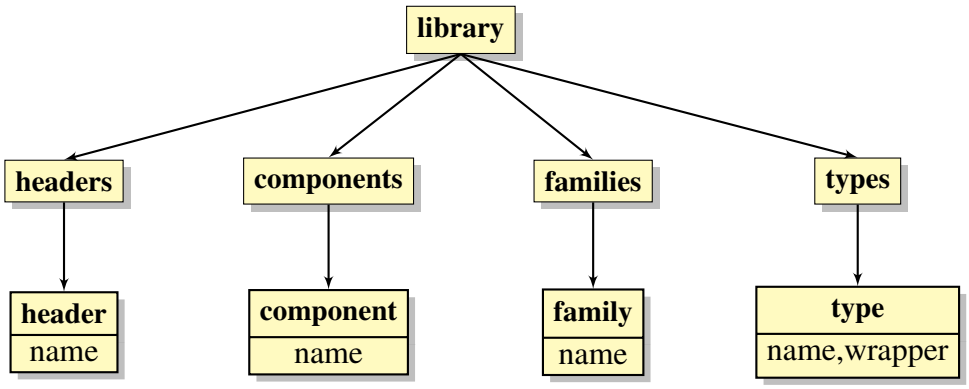


FIGURE 5 – Balises XML constituant la description d'une bibliothèque

Balise	Description	
composite	Balise de référence décrivant un composant composite	
	Enfants	*context* (requis, cf B.2.1), *sheet* (requis, cf B.2.2), *interface* (requis)
interface	Description des points d'entrée/sortie du composant	
	Enfants	*slots*, *signals*
slots	Liste des entrées	
	Enfants	*method* (requis)
signals	Liste des sorties	
	Enfants	*method* (requis)
method	Donne, pour un point d'échange donné, le composant auquel il est connecté	
	Attributs	
	alias	Requis - le nom du point d'échange
	component	Requis - une référence à un identifiant de composant auquel le point d'échange est connecté en interne
	method	Requis - le nom du point d'échange interne associé au composant connecté

TABLE 3 – Description des balises associées à un composant composite

B.1.4 Profils

Les profils permettent d'ajuster des jeux de paramètres pour les applications et peuvent être passés en paramètre du moteur d'exécution (cf annexe A.1). La figure 6 présente l'organisation des balises le constituant. Un profil peut-être considéré comme une moitié de contexte (cf annexe B.2.1). La table 5 donne la signification des balises utilisées pour décrire un profil.

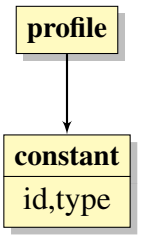


FIGURE 6 – Balises XML constituant le fichier de profil

Balise	Description	
library	Balise de référence décrivant une bibliothèque	
	Enfants	*headers* (requis), *families*, *components*, *types*
headers	Liste des fichiers d'en-tête déclarant les familles, composants et types utilisé dans la bibliothèque	
	Enfants	*header* (requis)
families	Liste des familles de composants exportables vers le moteur d'exécution	
	Enfants	*family* (requis)
components	Liste des composants exportables vers le moteur d'exécution	
	Enfants	*component* (requis)
types	Liste des types exportables vers le moteur d'exécution	
	Enfants	*type* (requis)
header	Fichier d'en-tête déclarant les familles, composants et types utilisé dans la bibliothèque	
	Attributs	
	name	Requis - chemin du fichier d'en-tête (relatif par rapport au fichier XML)
family	Exportation d'une famille en particulier	
	Attributs	
	name	Requis - nom de la famille.
	wrapper	Indique la classe faisant le point entre le moteur et le type à introduire.
component	Exportation d'un composant natif	
	Attributs	
	name	Requis - nom du composant.
type	Exportation d'un type	
	Attributs	
	name	Requis - nom du type considéré

TABLE 4 – Description des balises associées à une bibliothéque étendant le moteur.

Balise	Description	
profile	Balise de référence décrivant un profil	
	Enfants	*constant* (requis)
constant	Décrit une constante	
	Enfants	Description textuelle de la valeur de la constante (requis)
	Attributs	
	id	Requis - Identifiant de la constante
	type	Requis - Nom du type de la constante

TABLE 5 – Description des balises associées à un profil

B.2 Les structures secondaires

Les structures secondaires sont également employées dans les fichiers précédemment cités. Elles sont généralement communes à ces derniers, c'est pourquoi elles sont décrites séparément.

B.2.1 Le contexte

Un contexte maintient une liste de constantes et de composants dans lesquels l'application va puiser. Il contient également la liste des bibliothèques dans lesquelles les nouveaux composants et les nouveaux types sont réalisés. La hiérarchie de balises XML est représentée à la figure 7 et la table 6 donne la signification qui leur est associée.

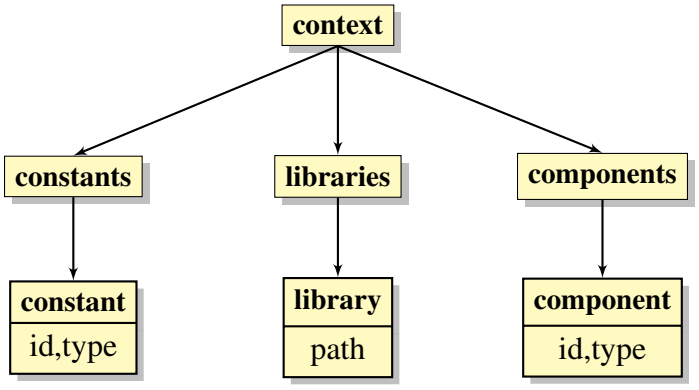


FIGURE 7 – Balises XML représentant un contexte d'exécution

B.2.2 Les feuilles

Les feuilles centralisent la liste des initialisations de préconnection, de postconnection et de nettoyage ainsi que la liste des connections réalisées entre les composants. La structure d'une feuille est représentée à la figure 8 et la table 7 donne la signification de ces dernières.

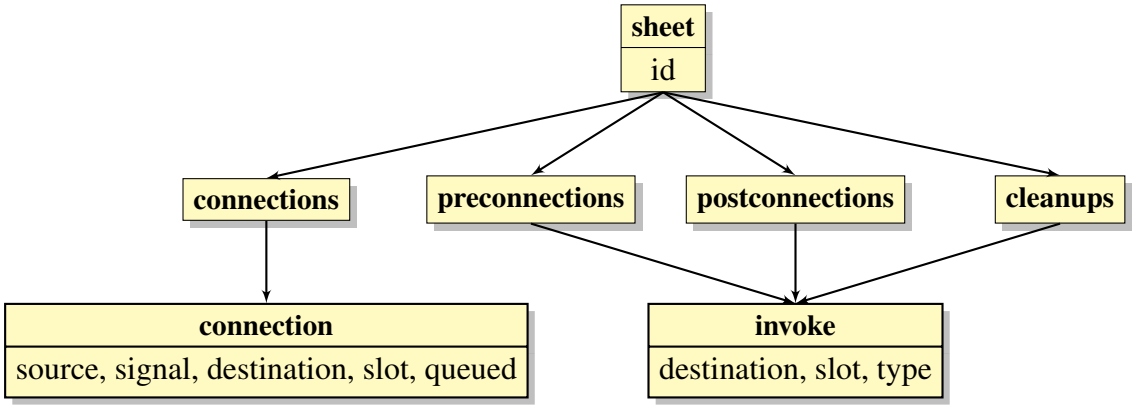


FIGURE 8 – Balises XML représentant une feuille

Balise	Description	
context	Balise générale associée à la description d'un contexte	
	Enfants	*libraries*, *constants*, *components* (requis)
libraries	Liste des bibliothèques utilisées pour instancier le contexte	
	Enfants	*library* (requis)
constants	Liste des constantes utilisées par le contexte	
	Enfants	*constant* (requis)
components	Liste des composants utilisés par le contexte	
	Enfants	*component* (requis)
library	Donne une bibliothèque que le moteur doit charger	
	Attributs	
	path	Requis - chemin relatif ou absolu de la bibliothèque à charger
constant	Décrit une constante du contexte	
	Enfants	Description textuelle de la valeur de la constante (requis)
	Attributs	
	id	Requis - identifiant de la constante
	type	Requis - type de la constante
component	Décrit un composant du contexte	
	Enfants	Description textuelle de la valeur du composant. Cette description peut-être celle d'une machien à état (cf B.2.3).
	Attributs	
	id	Requis - identifiant du composant
	type	Requis - type du composant

TABLE 6 – Description des balises associées à un contexte

Balise	**Description**	
sheet	Balise générale associée à une feuille	
	Enfants	*connections*, *preconnections*, *postconnections*, *cleanups*
	Attributs	
	id	Identifiant de la feuille (Requis dans une application).
connections	Liste des connections inter-composants	
	Enfants	*link* (requis)
preconnections	Liste des invocations de préconnection	
	Enfants	*invoke* (requis)
postconnections	Liste des invocations de postconnection	
	Enfants	*invoke* (requis)
cleanups	Liste des invocations de nettoyage	
	Enfants	*invoke* (requis)
link	Description d'une connexion inter-composants	
	Attributs	
	source	Requis - référence au composant émetteur de la communication
	destination	Requis - référence au composant récepteur de la communication
	signal	Requis - nom du signal du composant émetteur
	slot	Requis - nom du slot du composant récepteur
	queued	Booléen. Lorsque *queued* vaut *true*, alors la connection est asynchrone. La valeur par défaut est *false*
invoke	Description d'une invocation	
	Enfants	Description textuelle d'une valeur correspondant à un type reconnu par le moteur
	Attributs	
	destination	Requis - référence au composant destinataire de l'invocation
	slot	Requis - nom du slot du composant destinataire
	type	Requis - type du paramètre passé en invocation

TABLE 7 – Description des balises associées à une feuille

B.2.3 Les automates

Les automates sont les contrôleurs par défaut des processus. Ils sont décrits au moyen d'un état initial, d'un état final ainsi que d'une liste de transitions qui regroupent les états initiaux et finaux et le jeton d'activation comme représenté à la figure 9. Il est à noter que l'automate est considéré comme étant un composant comme les autres.

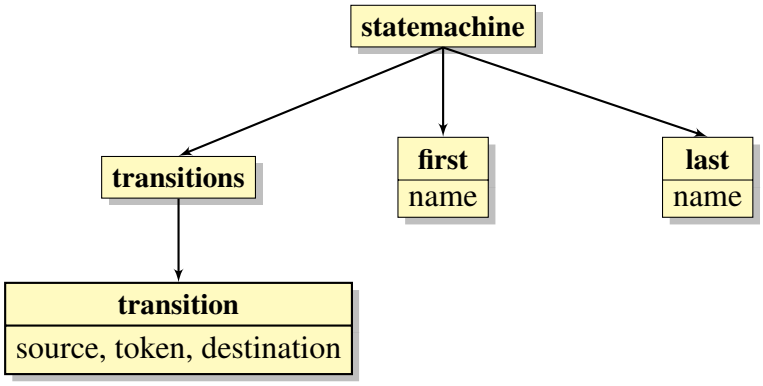


FIGURE 9 – Balises XML représentant une machine à états

Balise	**Description**	
statemachine	Balise générale décrivant un automate.	
	Enfants	*first* (requis), *last*, *transitions*
first	Donne le nom de l'état initial de l'automate	
	Attributs	
	name	Requis - référence à la feuille représentant l'état initial de l'automate
last	Donne le nom de l'état final de l'automate	
	Attributs	
	name	Requis - référence à la feuille représentant l'état final de l'automate
transitions	Liste des transitions de l'automate	
	Enfants	*transition* (requis)
transition	Description d'une transition	
	Attributs	
	source	Requis - état source de la transition, référence à un nom de feuille
	destination	Requis - état destination de la transition, référence à un nom de feuille
	token	Requis - jeton activant la transition

TABLE 8 – Description des balises associées à une machine à états

B.3 Exemples XML

B.3.1 Application

```
<application mode="event">
  <context>
    <libraries>
      <library path="../../sample/sample"/>
    </libraries>
    <components>
      <component id="b" type="Loop" />
      <component id="d" type="DisplayInt" />
      <component id="s" type="StateMachine">
        <statemachine>
          <first name="start"/>
          <last name="end"/>
          <transitions>
            <transition source="start" token="end" destination="end"/>
          </transitions>
        </statemachine>
      </component>
    </components>
    <constants>
      <constant id="iterations" type="int">5</constant>
    </constants>
  </context>

  <processes>
    <process controller="s">
      <sheet id="start">
        <connections>
          <link source="b" signal="newIteration(int)" destination="d"
              slot="display(int)"/>
          <link source="b" signal="sendToken(QString)" destination="s"
              slot="setToken(QString)" />
        </connections>
        <postconnections>
          <invoke destination="b" slot="setIterations(int)"
              type="constant">iterations</invoke>
        </postconnections>
      </sheet>
      <sheet id="end"/>
    </process>
  </processes>
</application>
```

Listing 19 – Déclaration d'une application

B.3.2 Composant composite

```
<composite>
        <context>
                <libraries>
                        <library path="../../sample/sample"/>
                </libraries>
                <components>
                        <component id="1" type="Loop"/>
                        <component id="d" type="DisplayInt"/>
                        <component id="12" type="Loop"/>
                </components>
        </context>
        <sheet>
                <connections>
                        <link source="1" destination="d"
                            signal="newIteration(int)" slot="display(int)"/>
                </connections>
        </sheet>
        <interface>
                <slots>
                    <method alias="setIterations(int)" component="1"
                        method="setIterations(int)"/>
                    <method alias="setIterations(int)" component="12"
                        method="setIterations(int)"/>
                </slots>
                <signals>
                        <method alias="sendToken(QString)" component="1"
                            method="sendToken(QString)"/>
                </signals>
        </interface>
</composite>
```

Listing 20 – Déclaration d'un composant composite

B.3.3 Profil

```
<profile>
    <constant id="iterations" type="int">10</constant>
</profile>
```

Listing 21 – Déclaration d'un profil