XSPIF: Developer Guide

a cross standards plugin framework

Vincent Goudard and Remy Muller

September 5, 2003

Abstract

XSPIF : ‘a (X)cross Standard PlugIn Framework’ is a development environment which offers the possibility of designing audio-plugins within an XML context, with a total abstraction from platform or standard. The XML files are parsed via python scripts, and one can get the C/C++ sources files in various plugin standards.

This document is aimed at providing information for people who would like to improve XSPIF abilities and performances. Before reading this document, you should first take a look at other available documentation (especially XSPIF’s User Guide [GM03a]).
Hence we could describe the Python code, but that would not be of much interest since it is pretty easy to understand, and almost exclusively consists in writing lines in a file. What we will describe here is rather the choices that have been made for the XSPIF environnement implementation for cross-platform and cross-standard purpose, as well as the specificities of each standard for the translation of the meta-plugin.

Contents

Chapter 1

Introduction

1.1 About XPSIF

XSPIF is born after a study [GM03b] of existing audio plugin standards, namely: Steinberg's VST, Apple's AudioUnit, LADSPA, Cakewalk's DXi; as well as objects in modular softwares such as MAX/MSP[1], jMax[2], PureData[3], and EyesWeb[4]. This study helped deducing a synthetic abstraction from the existing standards, to ease the development of audio plugins, with the underlying guideline: *"Write once, export many."*

The XML language was chosen to store the data needed to design an audio plugin. Its syntax is independant from any language, and platform, and it is highly flexible. The blocks of code are written in the standard ANSI C[5] language, which is also platefom independant.

1.2 Semantics

To help make things clearer, let us define some terms we will refer to in this manual, to name actors, objects and actions.

Peoples:

- A **XSPIF user** (or simply "user") is a person who aims at developping audio plugins, with the XSPIF assistance. This person should only have

[1]MAX/MSP from Cycling '74: `http://www.cycling74.com`
[2]jMax from IRCAM: `http://www.ircam.fr`
[3]Pure Data from Miller Puckette: `http://www.pure-data.org`
[4]EyesWeb from Laboratorio di Informatica Musicale: `http://www.eyesweb.org`
[5]C++ is also possible, but it narrows the scope of the exported standards

some basic knowledge of the C language, which was chosen for writing the callbacks in XSPIF.

- A **XSPIF developer** (or simply “developer”) is a person who develops XSPIF to add new features or standards to its scope. This person should know C/C++, XML, and Python.
- A **plugin user** is a person who uses audio plugins in an audio software called ‘host’. This person can know nothing at all about computer programming.

Objects:

- A **meta-plugin**: is a XML spefication which contains the “essence” of the plugin: i.e. everything needed to describe the plugin’s features and behaviour. The meta-plugins are stored in files using the `.xspif` extension to differentiate them from other XML files.

Actions:

- **Validating** is the process of examinating the meta-plugin’s XML file, and ensure that it matches the structure defined in the DTD[6]
- **Parsing** is the process of reading the XML file, and transforming it to a structured object called a DOM[7].
- **Checking** is the process of examinating the DOM elements, and checking if the description makes sense from the XSPIF point of view; e.g. that a parameter min value is less than its max value. . . etc.
- **Translating**: is the process of writing the C/C++ source files of a given standard, from the elements of the DOM tree.

1.3 Open Source License

XSPIF is Open Source[8], released under the General Public License[9]. This means that you are free to redistribute it, as long as you do not make a business with it. You are aslo free to modify and improve it: may this guide help you in this task.

[6]Document Type Definition: see 2.1.1 and the XML language syntax.
[7]Document Object Model
[8]`http://www.gnu.org/philosophy/free-sw.html`
[9]`http://www.gnu.org/licenses/gpl.html`

Chapter 2

Tools and languages used for development

2.1 XML: the plugin specification

XML: 'eXtended Markup Language' belongs to the family of markup languages as HTML for example. It is very useful to describe documents containing structured information. Structured information contains both content (words, pictures, etc.), and some indications on what role that content plays. As the major difference with other Markup Languages which comes with a predefined set of Tags, with XML you can specify you own ones dedicated to you own application [Wal98].

2.1.1 Document Type Definition : xspif.dtd

The Document Type Definition (DTD) is a file describing the way a specific XML file should be written. More precisely it defines what the possible or required elements are, where they are, how many of them one can find in the XML file, what are their attributes and sub-elements…

Here is a short XML symbols glossary:

?	allow zero or one element
*	allow zero or more elements
+	allow one or more elements
	if no symbol, allow one and only one element

On the next page is the DTD specification : `xspif.dtd`.

```
<!ELEMENT plugin (caption?,comment?,code?,pin+,param*,
                  controlout*,state*,callback+)>
<!ATTLIST plugin label          CDATA #REQUIRED
                 plugId         CDATA #REQUIRED
                 manufId        CDATA #REQUIRED
                 maker          CDATA #IMPLIED
                 copyright      CDATA #IMPLIED>

<!ELEMENT caption       (#PCDATA)>
<!ELEMENT comment       (#PCDATA)>
<!ELEMENT code          (#PCDATA)>

<!ELEMENT pin (caption?, comment?)>
<!ATTLIST pin   label           CDATA   #REQUIRED
                channels        CDATA   #REQUIRED
                dir          (In|Out)   #REQUIRED>

<!ELEMENT param (caption?,code?,comment?)>
<!ATTLIST param label           CDATA   #REQUIRED
                min             CDATA   #REQUIRED
                max             CDATA   #REQUIRED
                default         CDATA   #REQUIRED
                type      (float|int)   #REQUIRED
                mapping     (lin|log)   #REQUIRED
                unit            CDATA   #IMPLIED
                noinlet (true|false)    #IMPLIED>

<!ELEMENT controlout (caption?, comment?)>
<!ATTLIST controlout label      CDATA   #REQUIRED
                min             CDATA   #REQUIRED
                max             CDATA   #REQUIRED
                type      (float|int)   #REQUIRED
                mapping     (lin|log)   #IMPLIED
                unit            CDATA   #IMPLIED>

<!ELEMENT state EMPTY>
<!ATTLIST state label   CDATA           #REQUIRED
                type    CDATA           #REQUIRED>

<!ELEMENT callback   (code?)>
<!ATTLIST callback label (process|processEvents|
                          instantiate|deinstantiate|
                          activate|deactivate) #REQUIRED>
```

2.1.2 Required vs. optionnal features

The Document Type Definition `xspif.dtd` specifies the elements that the user can or has to use for the design of the meta-plugin. Some elements or attributes are required while some are optional, and the DTD helps specifying all this.
However, the DTD alone cannot handle all the cases for which the meta-plugin design is not complete enough or valid; e.g. if a label contains blank spaces or any bad character for a C++ class name, the XML file is still valid for the DTD, while the C source file will definitely fail to compile. This kind of verification needs special checking, which is performed after the DTD validation with the `generalCheck` function and in the standard-specific modules as it will be explained later in this document.

2.1.3 Elements and attributes

The choice of letting a certain feature be an element or a attribute is not very well defined in the XML world. In brief, elements either contain information, or have a structure of subelements, while attributes are characteristics or properties of the information object. There can be two sub-elements with the same name (if it is allowed by the DTD), but only one attribute. Then it is up to the developer to do what is more convenient.

2.2 Python: parsers and translators

Python [vRc03] is a powerful *'interpreted, interactive, object-oriented programming language'*, which has the advantage of being very easy, and intuitive. Furthermore, its license is an Open Source one, and a great number of people are contributing to its development, so that a large collection of librairies is available to the developers, as well as a strong community providing support for new users.

Last but not least when one want to design a cross-platform project, the Python implementation is portable: it runs on many brands of UNIX, on Windows, DOS, OS/2, Mac, Amiga...

2.2.1 XSPIF Python modules

The main Python script `xspif.py`, which performs the generation of the plugin's source files uses a module called `xspif`. This module contains the following sub-modules to handle its various tasks:

xspif.parsexml A sub-module with methods to parse XML, perform some verifications and manipulate DOM trees.

xspif.tools A sub-module providing common tools relative to the writing of the sources.

xspif.*standard* One sub-module per standard : xspif.vst, xspif.ladspa. . . etc. to write the C/C++ source code.

2.2.2 PyXML: XML package for Python

There are two main API for handling XML documents: SAX and DOM[1]. XSPIF uses the DOM API, which loads a whole XML tree as a unique data structure once in memory, while SAX loads it incrementally, being thus more useful for bigger XML files containing pictures for example. But as meta-plugins are text only, they're not really big, and DOM gives us a lot more flexibility because we can access the different nodes randomly on-demand.

As the default XML modules in the standard python distribution were not complete enough for our application, we used the additionnal PyXML package [Kc03]. A good documentation for PyXML can be found in the *libraries reference* [?], on the Python web site : `http:www.python.org`. The module `xml.parsers.xmlproc` is used to validate the meta-plugin file, with respect to the DTD, and the module `xml.dom.minidom` is used to parse the meta-plugin file, and convert it to a DOM data tree.

[1]SAX: "Simple API for XML"; DOM: "Document Object Model"

Chapter 3

Translation from XML to C/C++

3.1 General scenario of the parsing

To generate the C/C++ source code from the meta-plugin *filename.xspif*, in the plugin standard *standard*, the user launches the script `xspif.py` by typing:

python xspif.py *standard filename*.xspif

This will invoke:

1. **xspif.parsexml.validate** to validate the XML file with respect to the DTD and generate a DOM tree. The validation handles the XML syntax of the meta-plugin, i.e. it basically checks the name and number of elements, their place, and whether the required elements are present. However, this validation does not check the relevance of the data contained in these elements.

2. **xspif.tools.generalCheck** performs additional standard-independant checkings of the XML elements.

3. A folder is created where the meta-plugin is located, named after the meta-plugin filename, and a sub-folder named after the target standard name. The sources will be generated in this sub-folder.

4. **xspif.*standard*** The DOM tree is sent to the module corresponding to the target standard, where the C or C++ source code is written.

3.2 C vs C++ implementation

The choice has been made in XSPIF to let the user access to the parameters, states and pins directly by their label[1]. Since in C++, one can directly use the

[1]For the pins, it is the label of the pin, to which the channel index is appended. See [GM03a]

variable names from within the class scope without using explicitely `this->`, the labels written in the meta-plugin can be directly used to name the members variables.

In the C implementation, one will find the same object oriented style. The class is replaced by a structure, which is passed to all callbacks. The problem is that the variable contained in this structure cannot be named directly, and should be explicitly accessed from the structure with the `->` operator. To remedy to this problem, a local copy of all parameters and states contained in the plugin structure is done before they can be handled by their original name[2]. At the end of the callback, all these local variables are copied back to the plugin structure. This mechanism may look rather clumsy, since all the states and variables are not necessarily read or modified in the callback. However, the unused variables can easily be removed by the compiler, and hence do not burden the CPU cost.

3.3 Common implementation

3.3.1 Headers

At the beginning of any standard's plugin source-file, just after the `#includes`, should be pasted all the code contained in the sub-element `<code>` of the element `<plugin>` "as-is". The user should indeed write in this element the additionnal includes, as well as the local functions or macro he will need in his code.
By *independant function*, we mean routines not defined as callbacks in the XSPIF API, and which do not know more than what they are given as argument. Hence, they do not know the states, and parameters unless they are given them as argument.

3.3.2 XSPIF API: the macros

XSPIF_GET_SAMPLE_RATE should be accessible from any callback defined by XSPIF, and as a consequence, should be declared before them. It should return a float value corresponding to the current sample rate (in Hertz).

XSPIF_GET_VECTOR_SIZE should only be called within the process callback and return an integer value corresponding to the current buffersize.

[2]The label declared in the meta-plugin.

XSPIF_WRITE_SAMPLE can only be used in the process callback. The macro should be redefined as we will see in LADSPA and VST, for handling *replacing* and *accumulating* automatically.[3]

XSPIF_CONTROLOUT can be used both in the process callback and in the code associated to the parameters, therefore, it should be available in 2 "flavours":

3.3.3 Heading

At the very beginning of any standard's plugin source file, should be written a heading saying that the file is generated automaticlly, and containing information about the maker, the plugin name, copyright...etc. This piece of code can be taken directly from the XSPIF modules for standards already implemented.

3.3.4 Includes

Then, just after the `#includes`, should be pasted all the code from the sub-element `<code>` of the element `<plugin>`. The user should indeed write in this element the additionnal includes, as well as the independant routines he will need in his code.
By "independant routines", we mean routines not defined as callbacks in the XSPIF API, and which do not know more than what they are given as argument. Hence, they do not know the states, and parameters unless they are given them as argument.

3.4 Implementing parameters

3.4.1 Declaration

The parameters are always members of the plugin structure or class. It should be possible for the user to access the parameters value in any callback with the parameter label defined in the meta-plugin as it has been explained in 3.2

3.4.2 Mapping and boundaries

As much as possible, protection against parameters going out of range should be handled in the automatic code generation. It is often provided by the standard's

[3]These two behaviours corresponding to the *send* and *insert* we would find on a mixer console

API, but if not, this can be easily done with the following macro, as it has been done for PD.

```
   #define FIT_RANGE(value, min, max)
(((value) < min) ?  min :  ((value) > max) ?  max :  (value))
```

The mapping should be implemented, when the parameter values are not used directly. For example, parameters in VST are normalized between $[0; 1]$. Thus, the mapping from a parameter value *param* in the range [min, max] to its normalized representation[4]*val* would be:

```
   val = (param - min)/(max - min)
```

for a linear mapping, and:

```
   val = (log(param) - log(min)) / (log(max) - log(min))
```

for a logarithmic mapping.

Otherwise, in modular hosts where no plugin GUI is provided, it can be implemented as a comment to warn the plugin user, as it was done with method `print` generated by `pd.py`, the module for PureData.

3.4.3 Attached code

These piece of code are meant for parameters which need to recompute some values, and in particular: the states. Therefore the states should be directly available where this code is pasted.

3.5 Pins

3.5.1 Declaration

The pins are named after their label and their number of channels with an index starting from 1. As an example if we have 2 pins with their respective labels being input and sidechain, and with the first having 4 channels and the second only 1, we would declare input1, input2, input3, input4 and sidechain1. Each one is a pointer to a non-interleaved C-array corresponding to its channel.

[4]normalized value are used by the default GUI to adress the position of their controls. For example, with a knob, 0.5 means in the midle, while 0.0 is full left and 1.0 is full right

3.6 Implementing callbacks

A callback is written as a block of C code by the user who can assume the availability of some variables. During the translation, this block of code is not analysed, and pasted –as is–. Hence, XSPIF does NOT prevent the user from writing wrong code; yet the compiler will most likely warn him by raising an error.
As a general remark, when *translating* a callback, the developer should do all the necessary, so that all the variables the meta-plugin writer may need are available[5]. After the meta-plugin callback's code, should be written all the necessary to update any possible change of the variables the user could access.

3.6.1 instantiate

The plugin instanciation should be done automatically, with `malloc`, `new`, or any standard specific method. Since the user should do the memory allocation of the structures he needs in this callback, all the states and parameters should be available, even if not initialized.

3.6.2 deinstantiate

All the memory allocated by the user in his instantiate code, should be freed here by his deinstantiate code. So, again, all the variables the user could need should be available. The plugin class or structure de-instanciation should be done automatically, obviously **after** the structures de-instanciation, with `free`, `delete`, or any standard specific method.

3.6.3 activate / deactivate

Activate and deactivate callbacks are used to enable/disable the audio processing part of the plugin. If the plugin is deactivated, the audio stream should just flow from input to output without being modified, in a *bypass* mode.
This feature is already implemented in some standard like VST and LADSPA. For other standard, a flag can be set, that will be checked in the process callback.

[5]To know what the user should await, please refer to [GM03a]

3.6.4 process

The pointers to the input and output buffers should be available under the pin label, to which the channel index is appended, as declared in the meta-plugin.
All the XSPIF macros defined previously should be known at that point, and XSPIF_CONTROLOUT should be defined such that it does not kill the DSP thread safety (see next paragraph).

3.7 Implementing controlouts

The macro `XSPIF_CONTROLOUT` has been defined to handle control ouputs. When it is called outside the DSP thread (i.e. outside the `process` callback), the output value can be output directly, as soon as possible, the “index” argument being set to zero.
On the other hand, when `XSPIF_CONTROLOUT` is called in the process callback, it is possible that outputting the value will not be real time safe[6]. In that case, depending on the host way of dealing with outputting control within the process callback, a clock can be set, so that the controlout does not burden the real time constraints, and sample accurate synchronicity can be achieved[7] and the “groove” is preserved.

3.8 Documentation

Last but not least, it has been made a part of the meta-plugin specification, to allow the user to add informative data, such as comments, captions, and units. These data should seriously be taken in account to generate the code, for both the person who might read the generated code, and the plugin user through the default GUI. So consider not forgetting this side of the framework.

[6]This is not the case for LADSPA where it just consists in writing in a shared memory location

[7]with a constant delay however, corresponding to the time-length of an audio buffer.

Chapter 4

Extending XSPIF

XSPIF can be extended in two ways: adding new standards to it, and adding new features. These two ways are very orthogonal: adding a new standard can be viewed as a vertical approach, as one has to write down the new standard's source file(s). One the other hand, adding a new feature implies a horizontal approach, as the developer has to visit every standard to implement the new feature with respect to each standard conventions.

4.1 How to add a standard in XSPIF?

4.1.1 Modifying xspif.py

To be known from XSPIF, the module for the new standard should be imported in the main script `xspif.py` by adding the following line, where *std* stands for the new module's name:

import xspif.*std*

The new standard name should be added wherever it has to be placed in `xspif.py`. It should not present any difficulty to do so, by doing the same as what is written for other standards.

4.1.2 Creating a new module

The new module should be called after the name of the new standard, and placed in the folder `/xspif`.
The output files generated by this module will then automatically be written in a new directory, created where the meta-plugin is stored. For example, if the meta-plugin `text.xspif` is stored in the folder `/metaplugins`, the source code will be generated in the sub-folder:

/metaplugins/test/std/

However, and as much as possible, we recommandto follow the main guidelines given in chapter 3, as well as respecting the order of the callbacks implementation: indepedant code, declaration of the parameters, instantiate, deinstantiate, activate, deactivate, parameter's code, controlout routines, process. There is actually no obligation for following that order, but always following the same order helps keeping the XSPIF API clearer.
The obvious weak point of XSPIF is the lack of a more organized Python code architecture, concerning the translating part. The code is rather linear and many things are duplicated. We tried to add some routine when there was strict duplication, like declaring the states and parameters at the beginning of function for API in C language.
The function `getText(node, label)` is available from the module `parsexml.py`, to get the data of a sub-element (node or attribute) of a given node.

The problem we faced for a better python code, is the fact that hard-coded parts of the C/C++ code, and variables parts coming from the meta-plugin are deeply intricated. This made it difficult to reach a more synthetic Python code, without making it difficult to read.

4.2 How to add a feature in XSPIF?

We will give here an example: the addition of the controlout feature, used to output control values from the plugin.

4.2.1 Updating the DTD

First of all, the feature should be added in the Document Type Definition `xspif.dtd`. Special attention should be given to this, by stating as required only what is really required.

```
<!ELEMENT controlout (caption?, comment?)>
<!ATTLIST controlout label      CDATA   #REQUIRED
                min             CDATA   #REQUIRED
                max             CDATA   #REQUIRED
                type      (float|int)   #REQUIRED
                mapping     (lin|log)   #IMPLIED
                unit            CDATA   #IMPLIED>
```

This XML tag has 4 required attributes:

label: which identifies the element, and the possible variables related to this element in the source code to be written.

min and max: is necessary when output values are forced to fit within a given range (for a given standard), and thus perform any necessary mapping before outputting these output values.

type: is needed to declare the variable type in the plugin structure and callbacks, so that the user can handle this variable. Note that the value of this element is only float or int for now. Note that some plugin API store the parmeters as float internally and deliver them to you by casting them in the type you've specified.

Additionnal information tags which are not required for a correct implementation, should be declared as `#IMPLIED`. This information will be used to generate commentaries in the source code, and in the plugin GUI.

the caption: gives the full name of the controlout (meaning something more explicit than the label).

comments: can provide additionnal information about this element.

the mapping: gives the possibility to have a logarithmic mapping. the default mapping being linear.

4.2.2 Updating the checking module

Then, the feature has to be checked by the sub-module `generalCheck` of the module `tools.py`. This sub-module is called by the main script `xspif.py`, and any error that would lead to the writing of bad C/C++ source code should make this module stop and return -1, so that the script can stop immediately.
There are various things that can be checked with this module. The developer can go through the various steps, and add what is needed to check the validity of the data contained in the new element.
Please keep in mind that these verifications should still be **standard-independant** at this stage.

Validity of the label

If the new element has a label, the label should be unique to prevent from clashes in variables naming, and it should not contain any wrong character like spaces, commas. . . etc.

Unicity of the label can be easily checked with the routine `checkNotTwiceSameElement`. This function take a list as argument. It returns the first element that would appear twice, zero otherwise.

For our example, we just have to add `controlouts` in the list of elements that own a label:

```
---------------------- [begin Python code] ----------------------
# Check every label is unique
for tag in [pins, params, states, controlouts]:
    for el in tag:
        labels.append(getText(el,'label'))
labels.append(pluginLabel)

extraLabel = checkNotTwiceSameElement(labels)
if (0 != extraLabel):
    print(T+"Error: label "+extraLabel+" was used more than once")
    return -1
----------------------- [end Python code] -----------------------
```

Correctness of the label name can be checked with the routine `stringHasChar`. This function take a string and a list of wrong characters as arguments. It returns the first wrong character that would appear in the string, zero otherwise.

```
---------------------- [begin Python code] ----------------------
# Check labels do not contain bad characters
 WrongCharList = [" ","&","'",'"',"&","#","@","-","*"]
 for myString in labels:
     if ("" == myString):
         print(T+'Error: One of the labels is null!!!')
     WrongChar = stringHasChar(myString, WrongCharList)
     if (WrongChar):
         print(T+"Error: Label '"+l+"' contains bad characher '"+WrongChar+"'")
         return -1
----------------------- [end Python code] -----------------------
```

Validity of the min, max and default values

Min, max, and default values can be checked with the routine `checkMinMaxDefault`. This function take as argument an element that has `min`, `max`, `mapping` and `default` attributes. It will check that:

- `min` is stricly less that `max`.
- `min` and `max` are strictly superior than zero if the mapping is `log`.
- `default` fits within `min` and `max`.

4.2.3 Updating the translators

Then, this feature should be handled in all the translation modules. Here after is the example of the addition of the controlout's handling in various standards. First, we add the following line to get a node list of all controlout elements:

```
controlouts = domTree.getElementsByTagName('controlout')
```

VST

With VST, the only way to output control consist in using MIDI Continuous Controllers. As most of the MIDI instruments only takes into account single CC (i.e. 7-bit) we haven't implemented the use of 2 CC to extand the dynamic range to 14-bit.
Implementing controlout into VST firstly consists in declaring a MIDI out port wich is paradoxaly done by calling `wantEvents()` inside the `resume()` method and returning 1 to the following `canDo()` : `''sendVstEvents''` and `''sendVStMidiEvent''` this done with the follwing python code:

```
---------------------- [begin Python code] ----------------------
# Plugin suspend and resume

fcpp.write('//------------------------------------------------------------'+'\n'
           +'void '+pluginLabel+'::resume()'+'\n'
           +'{'+'\n')

if controlouts != []:
    fcpp.write(T+'wantEvents();'+'\n')

fcpp.write(T+activateCode+'\n'
           +'}'+'\n'
           +'\n')
----------------------- [end Python code] -----------------------
and further:

---------------------- [begin Python code] ----------------------
fcpp.write('long '+pluginLabel+'::canDo(char* text)'+'\n'
           +'\t'+'{'+'\n' )

if controlouts != []:
    fcpp.write(T+'if (!strcmp (text, "sendVstMidiEvent")) return 1;'+'\n'
               +T+'if (!strcmp (text, "sendVstEvents")) return 1;'+'\n')

fcpp.write('}'+'\n')
```

```
----------------------- [end Python code] -----------------------
```

Now that we have declared a midi Out port we have to handle the mapping of the control dynamic to the 0-127 range according to the specified mapping and then send this MIDI event to the host. The *XSPIF_CONTROLOUT()* macro is wrapped to the the controlout() method:

```
---------------------- [begin C code] ----------------------
#define XSPIF_CONTROLOUT(label, index, value)(controlOut(label,index,value))
----------------------- [end C code] -----------------------
```

and this method method is defined by the following python code which takes care of filling a VstMidiEvent structure as defined in the VSTSDK, for more detailed information on this topic refer to [?]:

```
---------------------- [begin Python code] ----------------------
if controlouts != []:
    fcpp.write(
        '//------------------------------------------------------------'+'\n'
        +'// control out'+'\n'
        +'\n'
        +'void '+pluginLabel+'::controlout(long label, long index, float value)'+'\
        +'{'+'\n'
        +T+'char out=0;'+'\n'
        +T+'char cc=0;'+'\n'
        +T+'switch(label)'+'\n'
        +T+'{'+'\n')

    i = 38 # first MIDI CC that will be used fo control out
    for controlout in controlouts:
        label = getText(controlout,'label')
        max   = label+'Max'
        min   = label+'Min'
        mapping = getText(controlout,'mapping')
        controloutCode = getText(param,'code')

        if i >127:
            print 'warning too much controlouts, CC numbers exceed 127'
            return

        fcpp.write(T*2+'case '+label+' :'+'\n'
                   +T*3+'cc = '+str(i)+';'+'\n')
        i = i+1
        if mapping == 'lin':
            fcpp.write(T*3+'out = 127*((value-'+min+')/('+max+'-'+min+'));'+'\n')
        elif mapping == 'log':
            fcpp.write(T*3+'out = 127*(log(value/'+min+')/log('+max+'/'+min+'));'+'

    fcpp.write(
        '\n'
        +T*2+'default: return;'+'\n'
        +T+'}'+'\n'
        +'\n'
        +T+'if(index>vector_size)'+'\n'
        +T*2+'index = vector_size;'+'\n'
        +'\n'

        +T+'VstMidiEvent vstEvent;'+'\n'
        +T+'VstEvents vstEvents; '+'\n'
        +T+'vstEvents.numEvents =1; '+'\n'
        +T+'vstEvents.reserved = 0; '+'\n'
        +T+'vstEvents.events[0] = (VstEvent*)(&vstEvent); '+'\n'
        +T+'vstEvents.events[1]= NULL; '+'\n'
        +'\n'
        +T+'memset(&vstEvent, 0, sizeof(vstEvent));'+'\n'
        +T+'vstEvent.type = kVstMidiType;'+'\n'
        +T+'vstEvent.byteSize = 24;'+'\n'

        +T+'vstEvent.deltaFrames = index; '+'\n'
        +T+'vstEvent.midiData[0]=0xb0; // tells it sends midi CC'+'\n'
        +T+'vstEvent.midiData[1]=cc; '+'\n'
        +T+'vstEvent.midiData[2]=out; '+'\n'
        +T+'((AudioEffectX *)this)->sendVstEventsToHost(&vstEvents);'+'\n'
        +'}'+'\n'
        )
---------------------- [end Python code] ----------------------
```

LADSPA

In LADSPA, controlouts can be implemented in a very direct way, as it just consists in writing a value to a pointer. All parameters, audio input and output, and control output are declared as "ports". For convenience, a list with all these ports already exists; we will just add the controlouts to it, and they will be declared in the plugin structure:

```
--------------------- [begin Python code] ---------------------
```

```
ports = audio_ports + param_ports + controlout_ports
                   (...)
fc.write('/* Audio and parameters ports */' + '\n')
for port in ports:
    fc.write('#define PORT_'+port[0].upper()
            +T*2+ str(ports.index(port))+'\n')
----------------------- [end Python code] -----------------------
```

Then, we will first define the XSPIF_CONTROLOUT macro, after the `#include`'s, for the parameters' attached code sections. Here the method can be called directly, hence the macros is defined like this:

```
---------------------- [begin Python code] ----------------------
if ([] != controlouts):
    fc.write('// Macro for control output' + '\n')
    fc.write('#undef XSPIF_CONTROLOUT'+ '\n'
            '#define XSPIF_CONTROLOUT(dest, index, source)'
            + '(*(dest) = (LADSPA_DATA(source)))'+ '\n')
----------------------- [end Python code] -----------------------
```

Finally, we have to declare the controlouts in the `_init` method of the LADSPA API:

```
---------------------- [begin Python code] ----------------------
control_nodes = params + controlouts
for cp_node in control_nodes:
    pl = 'PORT_'+ getText(cp_node, 'label').upper()
    fc.write(
        T*2 + '/* Parameters for ' + getText(cp_node, 'caption')+' */' + '\n'
        + T*2 + 'port_descriptors['+ pl + '] =' + '\n')
    if ('param'==cp_node.nodeName):
        fc.write(T*2+' LADSPA_PORT_INPUT | LADSPA_PORT_CONTROL;' + '\n')
    elif ('controlout'==cp_node.nodeName):
        fc.write(T*2+' LADSPA_PORT_OUTPUT | LADSPA_PORT_CONTROL;' + '\n')
    fc.write(
        T*2 + 'port_names[' + pl + '] =' + '\n'
        + T*2 + ' strdup("' +  getText(cp_node, 'caption') + '");' + '\n'
        + T*2 + 'port_range_hints[' + pl + '].HintDescriptor =' + '\n'
        + T*2)
----------------------- [end Python code] -----------------------
```

PureData

In PD, the controlouts will appear as outlets, and values are output via an API specific method called `outlet_float`[1].
We will first define the XSPIF_CONTROLOUT macro, after the `#include`'s, for the parameters' attached code sections. Here the method can be called directly, hence the macros is defined like this:

```
---------------------- [begin Python code] ----------------------
// Macro for control outputs
#undef XSPIF_CONTROLOUT
#define XSPIF_CONTROLOUT(dest, index, value)(outlet_float(dest, value))
----------------------- [end Python code] -----------------------
```

Then we need to declare the controlouts in the object's structure. Each controlout will own:

- A variable storing the value to be sent to the outlet.
- An object of type `t_outlet` as defined in the PD API.
- A clock than can be set in the *process* callback to postpone the output.

```
---------------------- [begin Python code] ----------------------
fc.write('typedef struct _' + pluginLabelTilde + ' {' + '\n'
               (...)
fc.write(T + '// Pointers to the outlets:\n')
for c_node in controlouts:
    c_label = getText(c_node, 'label')
    c_type  = getText(c_node, 'type')
    fc.write(T + c_type + ' '+ c_label+'Value;'+ '\n')
    fc.write(T + 't_outlet *' + c_label + ';' + '\n')
    fc.write(T + 't_clock *p_'+c_label+'Clock;'+ '\n')
               (...)
----------------------- [end Python code] -----------------------
```

Then we need a function that will do the output, when called by the clock of a given controlout:

```
---------------------- [begin Python code] ----------------------
if ([]!=controlouts):
    fc.write('\n' + '\n' + separator)
    fc.write('// Functions for controlouts called by clocks'+'\n')
```

[1]Only float values can be output.

```
    for c_node in controlouts:
        c_label = getText(c_node, 'label')
        fc.write('static void '+pluginLabelTilde+'_'+c_label+'('
                 +t_pluginLabel+' *x){'+'\n')
        fc.write(T+'outlet_float(x->'+c_label+',
                 x->'+c_label+'Value);'+'\n')
    fc.write('}' + '\n')
----------------------- [end Python code] -----------------------
```

The following function is the one that will be called by the macro XSPIF_CONTROLOUT, when called from the process callback. Note that the conversion from the sample index to the clock delay (in ms) is done here.

```
---------------------- [begin Python code] ----------------------
fc.write('// Function for controlouts : setting the clock'  + '\n')
fc.write('static void '+pluginLabelTilde+'_controlouts('
        +t_pluginLabel
        +' *x, t_outlet *dest, t_float index, t_float value){'+'\n')
for c_node in controlouts:
    c_label = getText(c_node, 'label')
    fc.write(T + 'if (dest == x->'+c_label+'){'+'\n')
    fc.write(T*2+'clock_delay(x->p_'+c_label+'Clock,
            index*1000/XSPIF_GET_SAMPLE_RATE());'+'\n')
    fc.write(T*2+'x->'+c_label+'Value = value;'+'\n')
    fc.write(T+'}' + '\n')
fc.write('}' + '\n')
----------------------- [end Python code] -----------------------
```

Then, just before writing the process callback, we need to re-define our macro, so that it set the relevant clock, instead of calling `float\_outlet`:

```
---------------------- [begin Python code] ----------------------
fc.write('// Macro for control outputs in the perform method : use clock' + '\n')
fc.write('#undef XSPIF_CONTROLOUT'+ '\n'
         '#define XSPIF_CONTROLOUT(dest, index, value)('
         + pluginLabelTilde+'_controlouts(x, dest,
           index*1000/XSPIF_GET_SAMPLE_RATE(), value)) '+ '\n')
----------------------- [end Python code] -----------------------
```

In the instantiate callback, we need to create the outlets and the clocks corresponding to the controlout outlets:

```
---------------------- [begin Python code] ----------------------
fc.write('void *' + pluginLabelTilde + '_new('
         + 't_symbol *s, int argc, t_atom *argv)' + '\n'
         + '{' +  '\n'
               (...)
    for c_node in controlouts:
        c_label = getText(c_node, 'label')
        fc.write(T + '//  controlout ' + c_label + '\n')
        fc.write(T+'x->'+c_label+' = outlet_new(&x->x_obj,  &s_float);'+'\n')
        fc.write(T+'x->p_'+c_label+'Clock =
                    clock_new(x, (t_method)'+pluginLabelTilde+'_'+c_label+');'+'\n'
               (...)
----------------------- [end Python code] -----------------------
```

In the deinstantiate callback, we need to destroy the outlets and the clocks:

```
---------------------- [begin Python code] ----------------------
if (deinstantiate  or [] != controlouts):
    fc.write('\n' + '\n' + separator)
    fc.write('// Plugin cleanup method' + '\n')
    fc.write('void ' + pluginLabelTilde + '_free('+t_pluginLabel+' *x){' + '\n')
               (...)
    for c_node in controlouts:
        c_label = getText(c_node, 'label')
        fc.write(T+'clock_free(x->p_'+c_label+'Clock);'+'\n')
               (...)
----------------------- [end Python code] -----------------------
```

Chapter 5

Perspectives and TODOs

5.1 Current limitations and known bugs

C and C++

The current version of XSPIF imposes the C language, to be compliant with plugins usually written in C, namely LADSPA, PureData, and jMax[1]. It is possible to write LADSPA and PureData sources in C++ but we didn't explore that field. Maybe that would be nice to be able to write the meta-plugin code parts, in both C and C++, with an optionnal flag warning the XSPIF translator for that.

5.2 MIDI instrument

Though it is possible to develop simple synthesizers with XSPIF, MIDI suppoort to develop instruments is not yet implemented. This is partly due to the fact that MIDI is not currently supported by LADSPA.

5.3 Custom GUI support

Since the Graphical User Interface (GUI) is a critical part for cross plateform and standard development, it would be nice to be able to specify the GUI with an XML specification. Such attempt has already been raised for LADSPA by Paul Davis[2], but is still at an early stage and meant for LADSPA only. In our case, we could imagine adding a $< gfx >$ tag which could specify the type of control widget for each parameter, its size and its position. If custom GUI support is added, it will help to take in account plugins standard which do not have default GUI like MAS or DX.

[1]Though not done in the current version, this is one of the target standards

[2]see [Dev03]

5.4 More with XML

5.4.1 Document generation

Though incomplete and at a rather early stage, XSPIF has been modelized with perpectives of evolutions, by making the plugin design part totally independant of the standards conventions, and using flexible and cross-platform language like XML and Python. The benefit is that you can use the information of the meta-plugin to generate additional documents concerning the plugin, such as HTML documentation. For example, we can think of a general plugin's GUI, which could use this information.

5.5 User front end

Though faster and easier than writing C or C++ code, writing XML is not that funny. A graphical front end to implement the various elements of the meta-plugin, launch the source code generation, and compilation would be more friendly.

5.5.1 XSPIF meta-plugin repositories

As XSPIF meta-plugin files are written in XML, it is very easy to share them over the internet. We could imagine building a cooperative database where these files could be centralized, so that people could inspire from pre-existing works instead of re-inventing the wheel.

Bibliography

[Dev03] Linux Audio Developers. Linux audio developers simple plugin api. `http://www.ladspa.org/`, 2003.

[GM03a] Vincent Goudard and Remy Müller. *XSPIF user guide*, 2003.

[GM03b] Vincent Goudard and Rémy Müller. *Real-time audio plugin architectures*, 2003. `http://www.ircam.fr/equipes/temps-reel/xspif/`.

[Kc03] A.M. Kuchling and col. Pyxml. `http://pyxml.sourceforge.net/`, 2003.

[vRc03] Guido van Rossum and col. Python. `http://www.python.org/`, 2003.

[Wal98] Norman Walsh. *A Technical Introduction to XML*, October 1998. `http://www.xml.com/pub/a/98/10/guide0.html`.