XSPIF: User Guide

a cross standards plugin framework

—

IRCAM – Centre Georges Pompidou

Vincent Goudard and Remy Muller

September 8, 2003

Contents

1 Introduction **1**

2 System requirement - Prerequisites **2**

2.1 What you should know 2
2.2 Python 2
2.3 PyXML 0.8.2 or above 2
2.4 Compiler/IDE and SDK 3
2.4.1 Compiler/IDE 3
2.4.2 SDKs 3

3 From the model to the XML description **4**

3.1 The model 4
3.2 Document Type Definition 5
3.3 XML syntax 5

4 XSPIF Tutorial **7**

4.1 Writing a XSPIF meta-plugin 7
4.1.1 The header 7
4.1.2 < *plugin* > 7
4.1.3 < *pin* > 9
4.1.4 < *param* > 10
4.1.5 < *state* > 11
4.1.6 < *controlout* > 11
4.1.7 < *callback* > 12
4.1.8 XSPIF API 14
4.1.9 C vs C++ 14
4.2 Code generation 15
4.3 Compilation and integration 15
4.3.1 Windows 15
4.3.2 Linux 15
4.3.3 Mac OSX 16

5 Standards specific considerations **17**
5.1 VST 17
5.2 Audio Units 17
5.3 LADSPA 17
5.4 PureData, Max/Msp, jMax 18

Bibliography **19**

Abstract

XSPIF is a framework dedicated to help developing cross-plateforms and cross-standards audio plugins. It consists in describing the plugin's behaviour – i.e. its actions and its state(s) – and its interface (i.e. its controllable parameters and its audio ports) in a XML file (.xspif) and then use PYTHON a python script to generate the C or C++ source-code for each supported "standard" – i.e. for now: VST, AudioUnit,LADSPA, Max/Msp, jMax and Pure Data –.

Chapter 1

Introduction

From the XSPIF point of view, a plugin is Digital Signal Processing module aimed to be integrated inside a host application within a network, a graph or a chain of other modules. Its "interface" is fully defined by its audio input and output ports (also called pins), its controllable parameters and its control output(s) – when control data is derived from the audio stream –.
Therefore, plugins generated with XSPIF are mainly meant for doing effects, analysis or "continuous synthesis" – i.e. as the notion of events is not modelized yet, MIDI synthetizers are not handled –.
Furthermore, XSPIF relies on the availability of a generic Graphical User Interface generated automatically by the host application as a remote for the plugin.

So what should you use XSPIF for?
XSPIF lets you develop plugins for multiple standards from the same code.
XSPIF can generate plugin templates from the definition of its "interface".
XSPIF is a fast way to prototype and test new algorithms.
XSPIF can help you starting a new project by generating the skeleton of your plugin and then let you cutomize it to your needs.

In this manual, you should find all the information or the pointers you would need to quickly start doing plugins with XSPIF. There is brief description of the "technologies" involved in XSPIF – i.e. XML and PYTHON – and the links to the necessary downloads, followed by a tutorial showing you how to write your first XSPIF plugin, generate the sources and compile them.

There are 6 supported standards – VST, AudioUnits, LADSPA and Max/MSP, PureData and jMax – but others are planned to be added. If you want collaborate to XSPIF by adding a format or a new feature, please refer to the "XSPIF: developper guide" [GM03a] and if you want to have an overview of the existing plugin standards and their specificities refer to "Audio plugin architectures" [GM03b].

Chapter 2

System requirement - Prerequisites

2.1 What you should know

To fully understand how to use XSPIF, it is necessary to have a general understanding of programming techniques, especially in C and of real-time constraints. However, XSPIF has been designed in order to make it the simplest possible to develop audio plugins.
If you want to develop audio plugins, it is also useful to have basic knowledge in signal processing. There are a number of websites, where you can find information, and sample code to start with. `http://www.musicdsp.org`.

2.2 Python

First of all, XSPIF needs python 2.2 or above to be installed on your machine. Usually it is installed by default on recent linux distributions. For windows, there is an installer available at: `http://www.python.org/download/` and for Mac OSX we recommand to install python through Fink (`http://fink.sourceforge.net/`) and FinkCommmander (a front-end for apt-get, similar to synaptic on linux) until there is a real installer of PyXML for OSX. For more explanation refer to [vRc03].

2.3 PyXML 0.8.2 or above

You will also need PyXML 0.8.2 or above. It is a python package dedicated to XML handling. You can find it at `http://pyxml.sourceforge.net/`. For Mac OSX we highly recommend to install it through FinkCommander which will

automatically check all the dependencies and show if a new version is available.
For Windows, there is also an installer available in the downloads page.
For more explanation refer to [Kc03].

2.4 Compiler/IDE and SDK

2.4.1 Compiler/IDE

- **OSX:** Our reference IDE is Project Builder which uses gcc for mac and is easy to use and to customize to run XSPIF to generate new sources before rebuilding the plugin.
- **Win:** Although some people have succeed in building VST plugins with MingW Dev C++ or Borland C++ Builder. We only provide examples for Visual C++ as the vstsdk is primarily designed to be used with Visual C++.
- **Linux:** The reference compiler we have used is gcc

2.4.2 SDKs

- **VST** Available on Steinberg's web site `http://ygrabit.steinberg.de/users/ygrabit/public_html/vstsdk/OnlineDoc/vstsdk2.3/index.html`[1]
- **AU** Available on Apple's site `http://developer.apple.com/audio/`
- **LADSPA** Available on the LADSPA web site `http://www.ladspa.org`.[2]
- **PD** Available on PureData's web site `http://www.puredata.org`.
- **Max/Msp** Available on Cycling 74's web site `http://www.cycling74.com/products/dldoc.html`.
- **jMax** The jMax distribution including all the source-code is available on Ircam's freesoftware web-site `http://freesoftware.ircam.fr/`.

[1] [Gc03] and [Gra03]
[2] [Dev03]

Chapter 3

From the model to the XML description

3.1 The model

Here is a simple model behind audio plugins architecture. A plugin can be viewed as a kind of black box, which gets audio buffer(s) and control value(s) as inputs, and outputs transformed audio buffer, and control values.

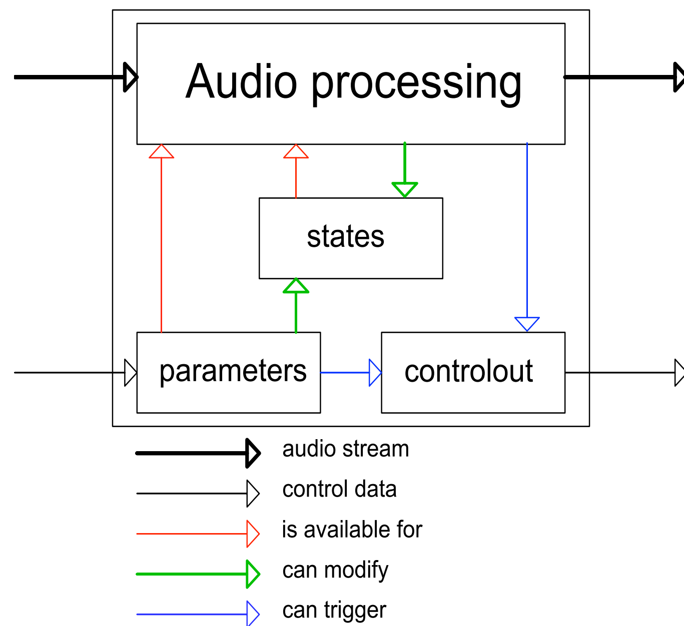

Figure 3.1: XSPIF model behind audio plugins

However, as long as our plugin remains a model, it will not do much.

In order to perform its job, this box needs to be *instanciated*. Instanciation consists in allocating the memory needed by the plugin and its variables.
Then it needs to be *activated*, i.e. perform and compute everything that is needed, so that the plugin is ready to perform some DSP processing.
If we want to stop processing audio for a while, we can *deactivate* the plugin, just like we would do with an on/off (or bypass) switch for a machine.
Now, when we no longer want to use this plugin, we have to *deinstantiate* to free the memory location used by the plugin, and by the structure it needs.
To sum up what is said above, we've introduced the following **callbacks** (i.e. the methods of the plugin that you can implement.):

- **instantiate**: the constructor.
- **deinstantiate**: the destructor.
- **activate**: plugin ON.
- **deactivate** plugin OFF (bypass).
- **process** where the DSP algorithm is implemented.

3.2 Document Type Definition

This very raw and simple model led us to a XML meta-plugin description with various elements. The Document Type Definition `xspif.dtd` is a file describing the way a meta-plugin 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 meta-plugin, what are their attributes and sub-elements and what value they can take.
The user can refer to this file to get a quick overview of the meta-plugin syntax. Otherwise refer to the next section 4 which will explain in details all the elements you can use and their respective roles.

3.3 XML syntax

XML[1][Wal98] is language to describe documents containing structured information. The information in embedded in *elements* defined by an opening and a closing *tag*:
< *myElement* > *...information...* < */myElement* >
These elements can have *child elements*, as well as *attributes* – some information specific to the element – The various elements used in XSPIF are listed below, and further described in the next chapter:

[1]XML: eXtended Markup Language

Elements	Signification
$< plugin(attributes) >< /plugin >$	Main element
$< pin(attributes) >< /pin >$	Audio input or output
$< param(attributes >< /param >$	Controllable parameter
$< state(attributes) >< /states >$	Internal state
$< controlout(attributes) >< /controlout >$	Control output
$< caption >< /callback >$	Caption of the parent element
$< comment >< /callback >$	Comment for the parent element
$< code >< /callback >$	Code for the parent element

Chapter 4

XSPIF Tutorial

4.1 Writing a XSPIF meta-plugin

A meta-plugin is a XML file beginning with a header specifying its syntax and followed by the various elements described in the last section organized in an arborescent structure whose root is $< plugin >$
This chapter will explain how to write the meta-plugin with detailed information on each specific elements, their hierarchy and examples of implementation.

4.1.1 The header

Each XSPIF meta-plugin description has to begin with the following header:

```
<?xml version="1.0"?>
<!DOCTYPE plugin SYSTEM "xspif.dtd">
```

The first line specifies which version of XML is used and the second line informs the parser where to find the the Document Type Definition (i.e. the syntaxic rules telling how a xspif file has to be written to be correct from the XML point of view).

4.1.2 $< plugin >$

This is the main tag which is the unique root of the tree.

- Attributes:

  label It will be used to name the plugin class or structure and shouldn't have spaces in it.

  plugId It should be either a four character constant into simple quotes, or an 32-bit unsigned integer identifying the plugin. It should be unique.

manufId another ID to be sure that the plugin can be uniquely identified.

maker [optionnal] a string describing the plugin maker

copyright [optionnal] a string to specify the copyright if needed. It will be used into the generated sources.

- Elements:

  < *caption* > [optionnal] A string describing the plugin

  < *comment* > [optionnal] a tag to add global comments to your code

  < *code* > [optionnal] you can declare local variables, macros, functions and additionnals includes segments on top of your file which **has to** be inside a `<![CDATA[...]]>` tag so that your code is not analyzed by the parser and can be pasted directly into your generated source file(s) "as-is".

  < *pin* > [1 or more] A pin is a port through which the audio signal flows. A pin is either input or output and can be multi-channels. see below 4.1.3

  < *param* > [0 or more] A param is a port for the control input, available to the plugin user and to the host. see below 4.1.4

  < *controlout* > [0 or more] A controlout is a port for outputting control back to the host or to other plugins when possible. see below 4.1.6

  < *state* > [0 or more] A state is keeping an internal information necessary for the behaviour of the plugin, but not available to the plugin user. see below 4.1.5

  < *callback* > [0 or more] There is a finite number of callbacks, which implement the behaviour of the plugin, namely: *instantiate, deinstantiate, activate, deactivate*, and *process*. see below 4.1.7

- Example:
  This example defines a plugin (the root element of the XML tree) whose label is `LowPass` with the plugin ID `'lowp'` and the manufacturer ID `ReMu`. There are additionnal information about the maker which is rémy muller and the copyright which is GPL. this plugin contains a caption `Lowpass` intented to be a human-readable name, some comments to describe the plugin and a piece of code which includes an additionnal header and defines a local function which will be used during by the DSP algorithm. Note that the < *plugin* > tag is not closed because it will also contain child elements.

```
<plugin  label = "LowPass" plugId="'lowp'" manufId="'ReMu'"
         maker="Remy Muller" copyright="GPL">
```

```
    <caption>Lowpass</caption>
    <comment>A simple lowpass with saturation</comment>
    <code><![CDATA[
    #include <math.h>
    /*****************************************/
    /* independent code here */
    static float saturate(float x)
    {
    if(x>0.f)
        x = 2.f*x-x*x;
    else
        x=  2.f*x + x*x;
    return x;
    }
    /*****************************************/
    ]]></code>
</plugin>
```

4.1.3 $< pin >$

Audio Ports are declared as pins, which means that a single stereo input is represented by 1 pin with 2 channels and direction = In.

- Attributes:

  label It will be used to name the audio buffers associated to the pin inside the DSP algorithm.

  dir It tells if the pin is an input or an output.

  channels It specifies the number of channels inside the pin.

- Elements:

  $<$ *caption* $>$ [optionnal] the friendly name of the pin.

  $<$ *comment* $>$ [optionnal] some comments about the pin.

- Example:
  Here we define a stereo audio pin which direction is an input, its label is input which means that the input buffers will be known as `input1[]` and `input2[]`. There is also a caption which can be used to show the pins name to the final user and some comment to document the code.

```
<pin label="input" dir="In" channels="2">
    <caption>Stereo Input</caption>
    <comment>This is a stereo input pin</comment>
</pin>
```

4.1.4 < *param* >

The parameters are the plugin variables that the user can control. They can be used either directly inside the process or converted into internal states.

- Attributes:

  label It will be used to name the parameter. Its value is stored inside the plugin class or structure and is always available when calling it directly by the label.

  min It specifies the minimum value that the parameter can take and should be less than max and default.

  max It specifies the maximum value that the parameter can take and should be strictly superior than min and default.

  default It specifies the default value that the parameter can take and should be in the range $[min; max]$

  type can be float or int.

  mapping can be lin for linear or log for logarithmic. Default is linear.

  unit [optionnal] a string for GUI display.

  noinlet [optionnal] can be “true” or “false”, default is “false”. It is only used in modular hosts so that not all parameters appear as an input.

- Elements:

  < *caption* > [optionnal] the friendly name of the parameter.

  < *comment* > [optionnal] some comment about this parameter.

  < *code* > [optionnal] each parameter change can trigger a piece of code to update the plugin’s states, or ouptput a control value for example. cf. 4.1.2

- Example:
  Here we define a cutoff parameter for a filter, in the range [100, 10000] Herz, with a default value of 1000 Hz. It has a logarithmic mapping, because it is more convenient for a frequency. This parametere has an associated piece of code which is triggered when the parameter changes to actualize the internal state `lambda`. We can also note that in this computation, the parameter is directly known from its label (`cutoff`) and that it also requires the samplerate which is accessed with a macro defined in the XSPIF API.

```
<param label="cutoff" min="100.0" max="10000.0" default="1000.0"
       type="float" mapping="log" unit="Hz">
    <caption>Cutoff frequency (Hz)</caption>
```

```
    <code><![CDATA[
    // cutoff and samplerate are both in Hertz
    lambda = exp(- cutoff / XSPIF_GET_SAMPLE_RATE());
    ]]></code>
    <comment>This is the cutoff frequency of the plugin</comment>
</param>
```

4.1.5 < *state* >

The states are all the variables stored inside the plugin, different from the parameters and that are necessary for the DSP algorithm – e.g. you can map cutoff and resonance parameters to internal filter coefficients – . Note that, as states can be of any type and to be compatible with both C and C++, they should be manually allocated with `malloc()` (or `calloc()`) and freed with `free()`.

- Attributes:

  label It will be used to name the state. Their value is stored inside the plugin class or structure and are always available when calling them directly by their label.

  type type can be anything either buit-in types or user defined.

- Example of two internal states; one of type float, and a pointer to a buffer of floats:

```
<state type="float" label="lambda"></state>
<state type="float *" label="buffer"></state>
```

4.1.6 < *controlout* >

- Attributes:

  label It will be used as a selector used when sending control outside of the plugin.

  min It specifies the minimum value that the parameter can take and should be less than max and default.

  max It specifies the maximum value that the parameter can take and should be strictly superior than min and default.

  type can be float or int

  mapping [optionnal] can be lin for linear or log for logarithmic. Default is linear.

- Elements:

  < *caption* > [optionnal] cf. 4.1.2

  < *comment* > [optionnal] cf. 4.1.2

- Example:
  A controlout called `env` for outputting the enveloppe of an audio signal, the min, max and mapping are used when this is necessary to scale the value to put it into a specific range (e.g. 0-127 for MIDI CC or 0-1 for normalized parameters...)

```
<controlout label="env" min="0.0" max="10000.0" type="float" mapping="log">
    <caption>Enveloppe</caption>
    <comment>Peak amplitude envelope</comment>
</controlout>
```

4.1.7 < *callback* >

The callbacks are not mandatory, so that one can generate templates. On the other side, having the same callback twice or more isn't allowed as it wouldn't be meaningful.

instantiate is the callback where the user should implement the memory allocation of all structures he will need with `malloc` or `calloc`. Initialization of the states can also be done here.

deinstantiate is the callback where the user should free the memory allocated for the structures in *instantiate* with `free`.

activate is called every time the plugin user of host switch on the plugin. The states can be reinitialized if needed (for example, by clearing a buffer used in a delay, so that the delay does not ring again when the plugin is reactivated). The parameters should not usually not be reinitialized, though this is not forbidden. If a structure depends on the sample rate (e.g. a delay buffer), it can be the right place to check if it has changed, and in this case, reallocate the memory for this structure.

deactivate is called every time the plugin user of host switch off the plugin. The states can be reinitialized if needed. The parameters should usually not be reinitialized, though this is not forbidden.

process is the callback where the user should implement the DSP algorithm.

- Attributes:

< *label* > the label can be any of instantiate (constructor), deinstantiate (destructor), activate (on), deactivate (off), process.

- Elements:

< *code* > [optionnal] The code which will be associated to the callback defined by *label*. cf. 4.1.2

- Examples:
Here we define the *process* callback which, as all the callback, only contains a piece of code. Note that there are 2 macros from the XSPIF API that are used in this example: $XSPIF_GET_VECTOR_SIZE()$ to know the size of the current buffer and $XSPIF_WRITE_SAMPLE()$ to write the processed samples to the the output buffers.

```
<callback label="process">
    <code><![CDATA[
    int i = 0;
    for(i=0;i < XSPIF_GET_VECTOR_SIZE();i++)
    {
      // in and out names are derived from the label in the pin declaration
      lp1 = (1.f-lambda)*input1[i] + lambda*lp1;
      lp2 = (1.f-lambda)*input2[i] + lambda*lp2;
      XSPIF_WRITE_SAMPLE(output1, i, saturate(lp1));
      XSPIF_WRITE_SAMPLE(output2, i, saturate(lp2));
    }
   ]]></code>
</callback>

<callback label="instantiate">
    <code><![CDATA[
    // Initialize the internal states
    lambda = lp1 = lp2 = 0.;
    ]]></code>
</callback>

<callback label="deactivate">
    <code><![CDATA[
    // clears memories so that no sound is output
    // when plugin is reactivated with no input
    lp1 = lp2 = 0.;
    ]]></code>
</callback>
```

4.1.8 XSPIF API

XSPIF API is quite simple and is limited to a few macros required to have a full abstraction from the plugin standards behind XSPIF. Their meaning is explained here after, as well as the part of the code in which they are allowed. *None of these macros are available in the code elements associated to the < plugin > element*

- $XSPIF_WRITE_SAMPLE(dest, index, value)$:
  It handles automatically adding or replacing process according to the standard.
  Availability: process callback only.
  dest has to be a pointer on a float array.
  index is the index in this array.
  value is the value to be written to the array.

- $XSPIF_CONTROLOUT(label, index, value)$:
  It handles control outputs, with sample accuracy in the process callback.
  Availability: process callback and parameters' attached code.
  label is the label given in the corresponding controlout tag and is used as selector for sending control.
  index if the offset in samples relative to the current buffer inside the process callback or zero if used outside.
  value is the value to be sent outside the plugin and can be converted automatically to a specific range depending on the standard (e.g. 0 to 127 if MIDI CC are used).

- $XSPIF_GET_SAMPLE_RATE()$:
  It returns a 32-bit float value corresponding to the actual sample-rate in Hz (44100Hz by default).
  Availability: any callback, and in the parameters' attached code.

- $XSPIF_GET_VECTOR_SIZE()$:
  It returns an integer value corresponding to the current vector size i.e. the size of the current size of the buffer to process.
  Availability: process callback only.

4.1.9 C vs C++

As some of the standards supported by XSPIF are natively written in the C language and that the C++ syntax is compatible with C, it has been decided that the code has to be written in C inside the meta-plugin description. However if you only plan to use standards written in C++ you still can use C++ inside the meta-plugin as it is just a matter of copy and paste, but you'll lose the advantage of quickly porting your algorithm to many standards.

4.2 Code generation

Now that you have a well writen meta-plugin inside your xspif file, you'll want to generate the sources for you plugin. For that purpose, there is a python script located in `xspif/python` named `xspif.py`. the syntax is:

python xspif.py *standard* [*path*]/*filename*.xspif

If we are inside xspif dsitribution, we could type:

python python/xspif.py vst examples/lowpass.xspif

Note that you can replace the standard by '**all**' which will write the sources for all the supported plateforms.

4.3 Compilation and integration

4.3.1 Windows

VST

With visual C++, create a new empty "win32 dynamic-link library"project. Add *yourplugin*.vst.cpp (which describes you plugin) and *yourplugin*.def (which exports the entry point of your library) and be sure that your include directories point to the vstsdk headers. if you're experienced with Visual C++, you can also integrate the XSPIF script as the first step of your building process using "custom build".

Pure Data

A makefile is generated automatically by XSPIF for PureData, but you will also need Microsoft Visual C++.
To build the plugin binary, open the windows shell and type `nmake pd_nt` in the folder where the PureData source are. The makefile is assuming some paths for PureData and Visual Studio: edit the Makefile if these paths do not match yours.

4.3.2 Linux

A makefile is generated automatically by XSPIF for LADSPA and PureData.
For LADSPA:

- `make` builds the dynamic library.

- `make install` builds the plugin and copy it in the LADSPA directory. Thus the environment variable LADSPA_PATH should be defined, as recommanded by the Linux Audio Developers.

- `make clean` remove objects, temporary and core files.

For PureData:

- `make pd_linux` builds the dynamic library.
- `make clean` remove the plugin binary, and objects, and core files.

4.3.3 Mac OSX

VST and AudioUnits

It is a good choice for VST and AudioUnits to use Project Builder as IDE[1]. Using XSPIF with Project Builder just consist in adding the script above as the first step of your building process. That way you can generate new sources each time you want to build your plugin. Please refer to the ProjectBuilder examples provided with the XSPIF distribution as a base for your own work.

PureData

For PD, the makefile generated automatically is quite straight forward: Open a shell, and type `make pd_darwin` in the directory where the plugin's source files are.

[1]Integrated Development Environment

Chapter 5

Standards specific considerations

5.1 VST

With VST 2.3, the only way to output control, is to use MIDI Continuous Controlers. For that purpose the values have to be normalized in the range $[0 - 127]$, and then sent to the host as MIDI data as defined by by the MMA[1].

5.2 Audio Units

Control output is not yet featured in this standard, but should come soon with Mac OSX 10.3 also called panther. Note that AudioUnit provides a set of predefined types – namely decibels, hertz, boolean, percent, seconds, phase, cents, Degrees. . . – with the mapping done automatically. However, as XSPIF can use arbitrary units with an arbitrary mapping, we only implemented AudioUnit's "generic parameters" which only have a range and default value with a linear mapping. Hence, if you specify a *log* mapping for a parameter, it will be ignored. Futur improvement to XSPIF, could use the *unit* – which can be arbitraty – attribute to guess the nature of the parameter.

5.3 LADSPA

In LADSPA, the control outputs undergo the same mechanism as writing output audio buffers, and it consists in writing a float value to a memory location which is or can be shared.
Thus, unlike VST and PD, synchronicity is not ensured and depends on the host management of the different threads.

[1]Midi Manufacturer Association `http://www.midi.org/`

5.4 PureData, Max/Msp, jMax

With Pure data, Max/Msp and jMax, if the parameter's optionnal attribute `noinlet` is set to *true*, it means that this specific parameter will not appear as an inlet on the object's layout. It can be useful if there are many parameters and you want to save space on the screen to keep some visibility. However, this parameter will still be controlable by sending the following message to the left-most inlet *parameter_name* value. Note that, in pure data, the object can also receive the message "print" which will display some information about the plugin, in particular, the parameters' names, and their range and the messages "on" and "off" which will call respectively *activate* and *deactivate*.

Bibliography

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

[Gc03] Yvan Grabit and col. *VST SDK 2.3 Online Documentation*, 2003. http://ygrabit.steinberg.de/users/ygrabit/public_html/vstsdk/OnlineDoc/vstsdk2.3/index.html.

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

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

[Gra03] Yvan Grabit. Vst 3rd party developer support. http://ygrabit.steinberg.de/users/ygrabit/public_html/, 2003.

[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.