Two Recent Extensions to the FAUST Compiler

K. Barkati
Ircam – SEL Team
karim.barkati@ircam.fr

D. Fober
Grame
fober@grame.fr

S. Letz
Grame
letz@grame.fr

Y. Orlarey
Grame
orlarey@grame.fr

ABSTRACT

We present two recently introduced extensions to the FAUST compiler. The first one concerns the *architecture* system and provides Open Sound Control (OSC) support to all FAUST generated applications. The second extension is related to preservation issues and provides a mean to automatically compute an all-comprehensive mathematical documentation of any FAUST program.

1. INTRODUCTION

FAUST [1] (*Functional Audio Stream*) is a functional, synchronous, domain specific language designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing languages like Max, PD, Supercollider, etc., is that programs are not interpreted, but fully compiled.

One can think of FAUST as a *specification language*. It aims at providing the user with an adequate notation to describe *signal processors* from a mathematical point of view. This specification is free, as much as possible, from implementation details. It is the role of the FAUST compiler to provide automatically the best possible implementation. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The compiler offers various options to control the generated code, including options to do fully automatic parallelization and take advantage of multicore machines.

From a syntactic point of view FAUST is a textual language, but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a *block-diagram algebra* of five composition operations (`: , ~ <: :>`).

For more details on the language we refer the reader to [1] [2]. Here is how to write a pseudo random number generator r in FAUST [2] :

```
r = +(12345)~ *(1103515245);
```

This example uses the recursive composition operator `~` to create a feedback loop as illustrated figure 1.

The code generated by the FAUST compiler works at the sample level, it is therefore suited to implement low-level DSP functions like recursive filters up to full-scale audio applications. It can be easily embedded as it is self-contained and doesn't depend of any DSP library or runtime system. Moreover, it has a very deterministic behavior and a constant memory footprint.

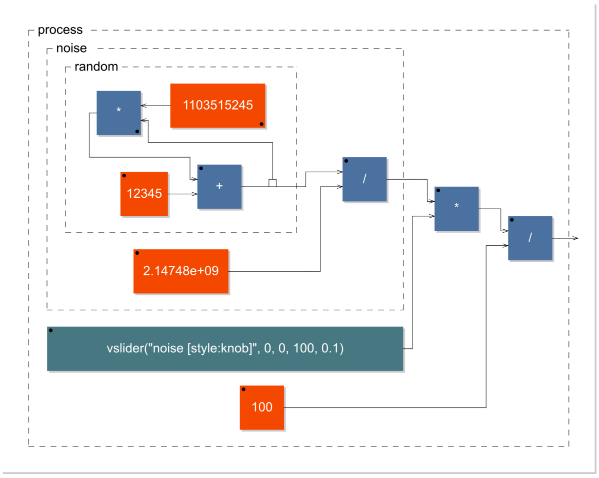


Figure 1. Block-diagram of a noise generator. This image is produced by the FAUST compiler using the `-svg` option.

The compiler can also wrap the generated code into an *architecture file* that describes how to relate the DSP computation to the external world. We have recently reorganized some of these architecture files in order to provide Open Sound Control (OSC) support. All FAUST generated applications can now be controlled by OSC. We will describe this evolution section 2.

Another recent addition is a new *documentation* backend to the FAUST compiler. It provides a mean to automatically compute an all-comprehensive mathematical documentation of a FAUST program under the form of a complete set of LaTeX formulas and diagrams. We will describe this *Self Mathematical Documentation* system section 3.

2. ARCHITECTURE FILES

Being a specification language, FAUST programs say nothing about audio drivers nor GUI toolkits to be used. It is the role of the architecture file to describe how to relate the DSP module to the external world. This approach allows a single FAUST program to be easily deployed to a large variety of audio standards (Max/MSP externals, PD externals, VST plugins, CoreAudio applications, Jack applications, iPhone, etc.). In the following sections we will detail this architecture mechanism and in particular the recently developed OSC architecture that allows FAUST programs to be controlled by OSC messages.

[1] http://faust.grame.fr
[2] Please note that this expression produces a signal $r(t) = 12345 + 1103515245 * r(t-1)$ that exploits the particularity of 32-bits integer operations.

2.1 Audio architecture files

A FAUST audio architecture typically connects the FAUST DSP module to the audio drivers. It is responsible for allocating and releasing the audio channels and to call the FAUST `dsp::compute` method to handle incoming audio buffers and/or to produce audio output. It is also responsible for presenting the audio as non-interleaved float data, normalized between -1.0 and 1.0.

A FAUST audio architecture derives an *audio* class defined as below:

```
class audio {
 public:
           audio() {}
  virtual ~audio() {}
  virtual bool init(const char*, dsp*) = 0;
  virtual bool start()                 = 0;
  virtual void stop()                  = 0;
};
```

The API is simple enough to give a great flexibility to audio architectures implementations. The `init` method should initialize the audio. At `init` exit, the system should be in a safe state to recall the `dsp` object state.

Table 1 gives the audio architectures currently available for various operating systems.

Audio system	Operating system
Alsa	Linux
Core audio	Mac OS X, iOS
Jack	Linux, Mac OS X, Windows
Portaudio	Linux, Mac OS X, Windows
OSC (see 2.3.2)	Linux, Mac OS X, Windows
VST	Mac OS X, Windows
Max/MSP	Mac OS X, Windows
CSound	Linux, Mac OS X, Windows
SuperCollider	Linux, Mac OS X, Windows
PureData	Linux, Mac OS X, Windows
Pure [3]	Linux, Mac OS X, Windows

Table 1. FAUST audio architectures.

2.2 GUI architecture files

A FAUST UI architecture is a glue between a host control layer (graphic toolkit, command line, OSC messages, etc.) and the FAUST DSP module. It is responsible for associating a FAUST DSP module parameter to a user interface element and to update the parameter value according to the user actions. This association is triggered by the `dsp ::buildUserInterface` call, where the `dsp` asks a UI object to build the DSP module controllers.

Since the interface is basically graphic oriented, the main concepts are *widget* based: a UI architecture is semantically oriented to handle active widgets, passive widgets and widgets layout.

A FAUST UI architecture derives an *UI* class (Figure 2).

```
class UI
{
 public:
           UI() {}
  virtual ~UI() {}

   -- active widgets
  virtual void addButton(const char* l, float* z)       = 0;
  virtual void addToggleButton(const char* l, float* z) = 0;
  virtual void addCheckButton(const char* l, float* z)  = 0;

  virtual void addVerticalSlider(const char* l, float* z,
          float init, float min, float max, float step) = 0;

  virtual void addHorizontalSlider(const char* l, float* z,
          float init, float min, float max, float step) = 0;

  virtual void addNumEntry(const char* l, float* z,
      float init, float min, float max, float step)     = 0;

   -- passive widgets
  virtual void addNumDisplay(const char* l, float* z,
                                               int p) = 0;

  virtual void addTextDisplay(const char* l, float* z,
             const char* names[], float min, float max) = 0;

  virtual void addHorizontalBargraph(const char* l,
                         float* z, float min, float max) = 0;

  virtual void addVerticalBargraph(const char* l,
                         float* z, float min, float max) = 0;

   -- widget layouts
  virtual void openTabBox(const char* l)                = 0;
  virtual void openHorizontalBox(const char* l)         = 0;
  virtual void openVerticalBox(const char* l)           = 0;
  virtual void closeBox()                               = 0;

   -- metadata declarations
  virtual void declare(float*, const char*, const char* ) {}
};
```

Figure 2. UI, the root user interface class.

2.2.1 *Active widgets*

Active widgets are graphical elements that control a parameter value. They are initialized with the widget name and a pointer to the linked value. The widget currently considered are `Button`, `ToggleButton`, `CheckButton`, `VerticalSlider`, `HorizontalSlider` and `NumEntry`.
A GUI architecture must implement a method
`addXxx (const char* name, float* zone, ...)` for each active widget. Additional parameters are available for `Slider` and `NumEntry`: the `init` value, the `min` and `max` values and the `step`.

2.2.2 *Passive widgets*

Passive widgets are graphical elements that reflect values. Similarly to active widgets, they are initialized with the widget name and a pointer to the linked value. The widget currently considered are `NumDisplay`, `TextDisplay`, `HorizontalBarGraph` and `VerticalBarGraph`.
A UI architecture must implement a method
`addxxx (const char* name, float* zone, ...)` for each passive widget. Additional parameters are available, depending on the passive widget type.

2.2.3 *Widgets layout*

Generally, a GUI is hierarchically organized into boxes and/or tab boxes. A UI architecture must support the following methods to setup this hierarchy :

```
  openTabBox (const char* l)
  openHorizontalBox (const char* l)
  openVerticalBox (const char* l)
  closeBox (const char* l)
```

Note that all the widgets are added to the current box.

2.2.4 *Metadata*

The FAUST language allows widget labels to contain metadata enclosed in square brackets. These metadata are handled at GUI level by a `declare` method taking as argument, a pointer to the widget associated value, the metadata key and value:

```
declare(float*, const char*, const char*)
```

UI	Comment
console	a textual command line UI
GTK	a GTK-based GUI
Qt	a multi-platform Qt-based GUI
FUI	a file-based UI to store and recall modules states
OSC	OSC control (see 2.3.1)

Table 2. Available UI architectures.

2.3 OSC architectures

The OSC [4] support opens the FAUST applications control to any OSC capable application or programming language. It also transforms a full range of devices embedding sensors (wiimote, smart phones, ...) into physical interfaces for FAUST applications control, allowing a direct use as music instruments (which is in phase with the new FAUST physical models library [5] adapted from STK [6]).

The FAUST OSC architecture is twofold: it is declined as a UI architecture and also as an audio architecture, proposing a new and original way to make digital signal computation.

2.3.1 *OSC GUI architecture*

The OSC UI architecture transforms each UI active widget addition into an `addnode` call, ignores the passive widgets and transforms containers calls (`openXxxBox, closeBox`) into `opengroup` and `closegroup` calls.

The OSC address space adheres strictly to the hierarchy defined by the `addnode` and `opengroup, closegroup` calls. It supports the OSC pattern matching mechanism as described in [4].

A node expects to receive OSC messages with a single float value as parameter. This policy is strict for the parameters count, but relaxed for the parameter type: OSC int values are accepted and casted to float.

Two additional messages are defined to provide FAUST applications discovery and address space discoveries:

- the `hello` message: accepted by any module root address. The module responds with its root address, followed by its IP address, followed by the UDP ports numbers (listening port, output port, error port). See the network management section below for ports numbering scheme.

- the `get` message: accepted by any valid OSC address. The `get` message is propagated to every terminal node that responds with its OSC address and current values (value, min and max).

Audio system	Environment	OSC support
	Linux	
Alsa	GTK, Qt	yes
Jack	GTK, Qt, Console	yes
PortAudio	GTK, Qt	yes
	Mac OS X	
CoreAudio	Qt	yes
Jack	Qt, Console	yes
PortAudio	Qt	yes
	Windows	
Jack	Qt, Console	yes
PortAudio	Qt	yes
	iOS (iPhone)	
CoreAudio	Cocoa	not yet

Table 3. OSC support in FAUST applications architectures.

Example:
Consider the *noise* module provided with the FAUST examples:

- it sends `/noise 192.168.0.1 5510 5511 5512`

  in answer to a `hello` message,

- it sends `/noise/Volume 0.8 0. 1.`
  in answer to a `get` message.

The OSC architecture makes use of three different UDP port numbers:

- 5510 is the listening port number: control messages should be addressed to this port.

- 5511 is the output port number: answers to query messages are send to this port.

- 5512 is the error port number: used for asynchronous errors notifications.

When the UDP listening port number is busy (for instance in case of multiple FAUST modules running), the system automatically looks for the next available port number. Unless otherwise specified by the command line, the UDP output port numbers are unchanged.

A module sends its name (actually its root address) and allocated ports numbers on the OSC output port on startup.

Ports numbers can be changed on the command line with the following options:

```
[-port | -outport | -errport] number
```

The default UDP output streams destination is `localhost` . It can also be changed with the command line option
`-dest address` where address is a host name or an IP number.

2.3.2 *OSC audio architecture*

The OSC audio architecture implements an audio architecture where audio inputs and outputs are replaced by OSC messages. Using this architecture, a FAUST module accepts arbitrary data streams on its root OSC address, and handles this input stream as interleaved signals. Thus, each incoming OSC packet addressed to a module root triggers a computation loop, where as much values as the number of incoming frames are computed.

The output of the signal computation is sent to the OSC output port as non-interleaved data to the OSC addresses `/root/n` where `root` is the module root address and `n` is the output number (indexed from 0).

For example:
consider a FAUST program named *split* and defined by:

```
process = _ <: _,_
```

the message

```
/split 0.3
```

will produce the 2 following messages as output:

```
/split/0 0.3
/split/1 0.3
```

The OSC audio architecture provides a very convenient way to execute a signal processing at an arbitrary rate, allowing even to make step by step computation. Connecting the output OSC signals to Max/MSP or to a system like INScore [3] , featuring a powerful dynamic signals representation system [7], provides a close examination of the computation results.

2.4 Open issues and future works

Generally, the labeling scheme for a GUI doesn't result in an optimal OSC address space definition. Moreover, there are potential conflicts between the FAUST UI labels and the OSC address space since some characters are reserved for OSC pattern matching and thus forbidden in the OSC naming scheme. The latter issue is handled with automatic characters substitutions. The first issue could be solved using the metadata scheme and will be considered in a future release.

Another issue, resulting from the design flexibility, relies on dynamic aggregation of multiple architectures covering the same domain: for example, it would be useful to embed both a standard and the OSC audio architecture in the same module and to switch dynamically between (for debugging purposes for example). That would require the UI to include the corresponding control and thus a mechanism to permit the UI extension by the UI itself would be necessary.

3. SELF MATHEMATICAL DOCUMENTATION

Another recent addition to the FAUST compiler is the *Self Mathematical Documentation* developped within ASTREE, an ANR funded research project (ANR 08-CORD-003) on preservation of real-time music works involving IRCAM, GRAME, MINES-PARISTECH and UJM-CIEREC.

The problem of documentation is well known in computer programming at least since 1984 and Donald Knuth's claim [8]: "*I believe that the time is ripe for significantly better documentation of programs [...].*"

A quarter-century later, general purpose programming languages can use `doxygen`, `javadoc` or others *Literate Programming* tools. But computer music languages lack integrated documentation systems and preservation of real-time music works is a big issue [9].

[3] http://inscore.sf.net

The *self mathematical documentation* extension to the FAUST compiler precisely addresses this question for digital signal processing (unfortunately not yet the asynchronous and more complex part). It provides a mean to automatically compute an all-comprehensive mathematical documentation of a FAUST program under the form of a complete set of LaTeX formulas and diagrams.

One can distinguish four main goals, or uses, of such a self mathematical documentation:

1. *Preservation*, i.e. to preserve signal processors, independently from any computer language but only under a mathematical form;

2. *Validation*, i.e. to bring some help for debugging tasks, by showing the formulas as they are really computed after the compilation stage;

3. *Teaching*, i.e. to give a new teaching support, as a bridge between code and formulas for signal processing;

4. *Publishing*, i.e. to output publishing material, by preparing LaTeX formulas and SVG block diagrams easy to include in a paper.

The first and likely most important goal of preservation relies on the strong assumption that *maths will last far longer than any computer language*. This means that once printed on paper, a mathematical documentation becomes a *long-term preservable* document, as the whole semantics of a DSP program is translated into two languages independant from any computer language and from any computer environment: the mathematical language, mainly, and the natural language, used to structure the presentation for the human reader and also to precise some local mathematical items (like particular symbols for integer operations). Thus, the mathematical documentation is self-sufficient to a programmer for reimplementing a DSP program, and shall stay self-sufficient for decades and probably more!

3.1 The `faust2mathdoc` Command

The FAUST *self mathematical documentation* system relies on two things: a new compiler option `--mathdoc` and a shell script `faust2mathdoc`. The script first calls `faust --mathdoc`, which generates:

- a top-level directory suffixed with "`-mdoc`",
- 5 subdirectories (`cpp/`, `pdf/`, `src/`, `svg/`, `tex/`),
- a LaTeX file containing the formulas,
- SVG files for the block diagrams;

then it just finishes the work done by the FAUST compiler,

- moving the output C++ file into `cpp/`,
- converting all SVG files into PDF files,
- launching `pdflatex` on the LaTeX file,

- moving the resulting pdf file into `pdf/`.

For example, the command

```
faust2mathdoc noise.dsp
```

will generate the following hierarchy of files :

- ▾ `noise-mdoc/`
  - ▾ `cpp/`
    - ◇ `noise.cpp`
  - ▾ `pdf/`
    - ◇ `noise.pdf`
  - ▾ `src/`
    - ◇ `math.lib`
    - ◇ `music.lib`
    - ◇ `noise.dsp`
  - ▾ `svg/`
    - ◇ `process.pdf`
    - ◇ `process.svg`
  - ▾ `tex/`
    - ◇ `noise.pdf`
    - ◇ `noise.tex`

3.2 Automatic Mode

The user has the possibility to introduce in the FAUST program special tags to control the generated documentation. When no such tags are introduced, we are in the so-called *automatic mode*. In this case everything is automatic and the generated PDF document is structured in four sections:

1. "Mathematical definition of `process`"
2. "Block diagram of `process`"
3. "Notice"
4. "Faust code listings"

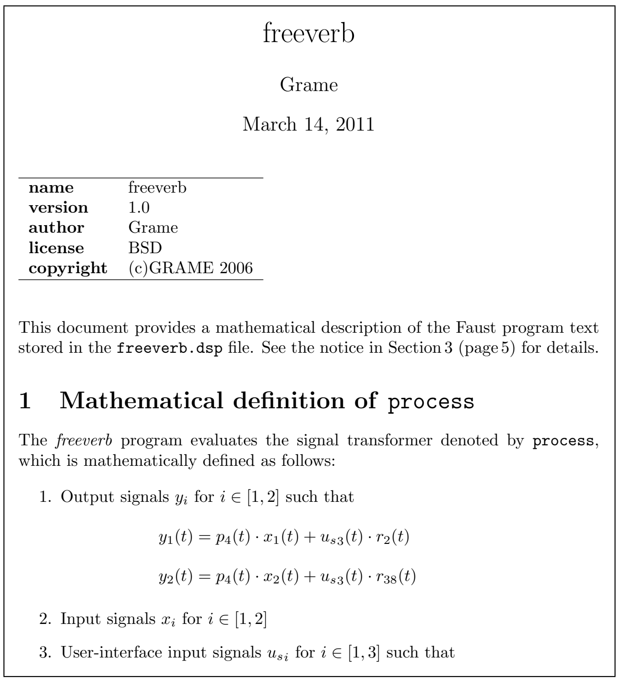
3.2.1 Front Page

First, to give an idea, let's look at the front page of a mathematical documentation. Figure 3 shows the front page of the PDF document generated from the `freeverb.dsp` FAUST program (margins are cropped).

The header items are extracted from the metadatas declared in the FAUST file:

```
declare name        "freeverb";
declare version     "1.0";
declare author      "Grame";
declare license     "BSD";
declare copyright   "(c)GRAME 2006";
```

The date of the documentation compilation is inserted and some glue text is added to introduce each section and the document itself. So, in addition to the mathematical language, the document also relies on the natural language, but one can legitimately expect it to last far longer than any current computer language.



freeverb

Grame

March 14, 2011

name	freeverb
version	1.0
author	Grame
license	BSD
copyright	(c)GRAME 2006

This document provides a mathematical description of the Faust program text stored in the `freeverb.dsp` file. See the notice in Section 3 (page 5) for details.

1 Mathematical definition of `process`

The *freeverb* program evaluates the signal transformer denoted by `process`, which is mathematically defined as follows:

1. Output signals y_i for $i \in [1,2]$ such that

$$y_1(t) = p_4(t) \cdot x_1(t) + u_{s3}(t) \cdot r_2(t)$$

$$y_2(t) = p_4(t) \cdot x_2(t) + u_{s3}(t) \cdot r_{38}(t)$$

2. Input signals x_i for $i \in [1,2]$
3. User-interface input signals u_{si} for $i \in [1,3]$ such that

Figure 3. Front page excerpt.

3.2.2 Mathematical definition of `process`

The first printed section contains whole mathematical definition of `process`. Obviously, the computation of the formulas printing is the most important part of the mathematical documentation.

To handle a LaTeX output for the mathematical documentation, instead of using a simple pattern matching substitution, the FAUST compiler has been extended from within, by reimplementing the main classes, in order to print a normalized form of the equations. This means that like the standard C++ output of the compiler, the LaTeX output is computed after the compilation of the signal processors, thus benefiting from all simplifications and normalizations that the FAUST compiler is able to do.

Some printed formulas are shown on Figure 3 (from the `freeverb.dsp` file) and Figure 4 (from `HPF.dsp`, a high-pass filter), as they appear in the corresponding generated PDF documents.

On Figure 3, one can see the definition of three kinds of signals, while on Figure 4 one can see two other kinds, and these are exactly the five families of signals that are handled:

- "Output signals",
- "Input signals",
- "User-interface input signals",
- "Intermediate signals",
- "Constant signals".

In fact, the *documentator* extension of the FAUST compiler manages several kinds of signals and makes a full use of FAUST signal tagging capabilities to split the equations.

4. Intermediate signals p_i for $i \in [1,8]$ and r_1 such that

$$p_1(t) = k_1 \cdot \max\left(0, u_{s1}(t)\right)$$

$$p_2(t) = \cos\left(p_1(t)\right)$$

$$p_3(t) = 2 \cdot p_2(t)$$

$$p_4(t) = 0.5 \cdot \frac{\sin\left(p_1(t)\right)}{\max\left(0.001, u_{s2}(t)\right)}$$

$$p_5(t) = \left(p_4(t) - 1\right)$$

$$p_6(t) = \left(1 + p_2(t)\right)$$

$$p_7(t) = 0.5 \cdot p_6(t)$$

$$p_8(t) = \frac{1}{1 + p_4(t)}$$

$$\begin{aligned} r_1(t) = p_8(t) \cdot (x_1(t{-}1) \cdot (0 - (p_6(t) - x_2(t))) + p_7(t) \cdot x_1(t) \\ + p_7(t) \cdot x_1(t{-}2) + p_5(t) \cdot r_1(t{-}2) + p_3(t) \cdot r_1(t{-}1)) \end{aligned}$$

5. Constant k_1 such that

$$k_1 = \frac{6.28318530717959}{f_S}$$

Figure 4. Some printed formulas.

Letter	**Signal Type**
$y(t)$	Output signal
$x(t)$	Input signal
$u_b(t)$	User-interface button input signal
$u_c(t)$	User-interface checkbox input signal
$u_s(t)$	User-interface slider input signal
$u_n(t)$	User-interface numeric box input signal
$u_g(t)$	User-interface bargraph output signal
$p(t)$	Intermediate parameter signal (running at control rate)
$s(t)$	Intermediate simple signal (running at sample rate)
$r(t)$	Intermediate recursive signal (depends on previous samples $r(t-n)$)
$q(t)$	Intermediate selection signal (2 or 3-ways selectors)
$m(t)$	Intermediate memory signal (1-sample delay explicitely initialized)
$v(t)$	Intermediate table signal (read or read-and-write tables)
$k(t)$	Constant signal

Table 4. Sub-signal formulas naming.

This is very important for human readability's sake, or else there would be only one very long formula for `process`! The *documentator* pushes this idea a step further than the five main signal families, using letters and numeric indices to name the left member of each subequation.

The indices are easy to understand: on Figure 3 for example, mentions like "$y_1(t)$", "$y_2(t)$" and "Input signals x_i for $i \in [1,2]$" clearly indicates that the `freeverb` block diagram has two input signals and two output signals, i.e. is a stereo signal transformer.

The letter choice is a bit more complex, summarised in Table 4.

3.2.3 *Fine mathematical automatic display*

3.2.4 *Block diagram of `process`*

The second section draws the top-level block diagram of `process`, i.e. a block diagram that fits on one page. The appropriate fitting is computed by the FAUST compiler part that handles the SVG output.

Figure 1 shows the block diagram computed from the `noise.dsp` file (a noise generator). By default, the top-level SVG block diagram of `process` is generated, converted into the PDF format through the `svg2pdf` utility (using the 2D graphics Cairo library), entitled and inserted in the second section of the documentation as a floating LaTeX figure (in order to be referenceable).

3.2.5 *Notice*

The third section presents the notice, to enlighten the documentation, divided in two parts:

- a common header (shown on Figure 6);
- a dynamic mathematical body (an example is shown on Figure 7, from the `capture.dsp` file).

For later reading improvement purposes, the first part intensively uses the natural language to contextualize the documentation as much as possible, giving both contextual information – with the compiler version, the compilation date, a block diagram presentation, FAUST and SVG URLs, the generated documentation directory tree – and key explanations on the FAUST language itself, its (denotational) mathematical semantics – including the `process` identifier, signals and signal transformers semantics.

3.2.6 *Faust code listings*

The fourth and last section provides the complete listings. All FAUST code is inserted into the documentation, the main source code file and all needed librairies, using the pretty-printer system provided by the `listings` LaTeX package.

You may wonder why we print FAUST code listings while the FAUST language is also affected by our mathematical abstraction *moto* that maths will last far longer than any computer language... It is mainly to add another help item for contextualization! Indeed, depending on the signal processing algorithms and implementations, some FAUST code can prove extremely helpful to understand the printed formulas, in the view of reimplementing the same algorithm in decades under other languages.

3.3 Manual Documentation

You can specify yourself the documentation instead of using the automatic mode, with five xml-like tags. That permits to modify the presentation and to add your own comments, not only on `process`, but also about any expression you'd like to. Note that as soon as you declare an `<mdoc>` tag inside your FAUST file, the default structure of the au-

```
        \begin{dmath*}
                p_{6}(t) =  \left(1 + p_{2}(t)\right)
        \end{dmath*}
        \begin{dmath*}
                p_{7}(t) = 0.5 * p_{6}(t)
        \end{dmath*}
        \begin{dmath*}
                p_{8}(t) = \frac{1}{1 + p_{4}(t)}
        \end{dmath*}
    \end{dgroup*}


    \begin{dgroup*}
        \begin{dmath*}
                r_{1}(t) = p_{8}(t) *  \left(x_{1}(t\!-\!1) *  \left(0 -
  \left(p_{6}(t) - x_{2}(t)\right) \right)  + p_{7}(t) * x_{1}(t) + p_{7}(t) *
x_{1}(t\!-\!2) + p_{5}(t) * r_{1}(t\!-\!2) + p_{3}(t) * r_{1}(t\!-\!1)\right)
        \end{dmath*}
    \end{dgroup*}

\item Constant $k_$ such that
    \begin{dgroup*}
        \begin{dmath*}
                k_{1} = \frac{6.28318530717959}{f_S}
        \end{dmath*}
    \end{dgroup*}
```

Figure 5. Corresponding LaTeX formulas' code.

tomatic mode is ignored, and all the LaTeX stuff becomes up to you!

Here are the six specific tags:

- `<mdoc></mdoc>` to open a documentation field in the FAUST code,
  - `<equation></equation>` to get equations of a FAUST expression,
  - `<diagram></diagram>` to get the top-level block-diagram of a FAUST expression,
  - `<metadata></metadata>` to reference FAUST metadatas,
  - `<notice />` to insert the "adaptive" notice of all formulas actually printed,
  - `<listing [attributes] />` to insert the listing of FAUST files called,
    - @ `mdoctags=[true|false]`
    - @ `dependencies=[true|false]`
    - @ `distributed=[true|false]`

3.4 Practical Aspects

3.4.1 *Installation Requirements*

Here follows a summary of the installation requirements to generate the mathematical documentation:

- `faust`, of course!
- `svg2pdf` (from the Cairo 2D graphics library), to convert block diagrams, as LaTeX doesn't handle SVG directly yet...
- `breqn`, a LaTeX package to manage automatic breaking of long equations,
- `pdflatex`, to compile the LaTeX output file.

3 Notice

- This document was generated using Faust version 0.9.36 on March 14, 2011.
- The value of a Faust program is the result of applying the signal transformer denoted by the expression to which the `process` identifier is bound to input signals, running at the f_S sampling frequency.
- Faust (*Functional Audio Stream*) is a functional programming language designed for synchronous real-time signal processing and synthesis applications. A Faust program is a set of bindings of identifiers to expressions that denote signal transformers. A signal s in S is a function mapping[1] times $t \in \mathbb{Z}$ to values $s(t) \in \mathbb{R}$, while a signal transformer is a function from S^n to S^m, where $n, m \in \mathbb{N}$. See the Faust manual for additional information (http://faust.grame.fr).
- Every mathematical formula derived from a Faust expression is assumed, in this document, to having been normalized (in an implementation-dependent manner) by the Faust compiler.
- A block diagram is a graphical representation of the Faust binding of an identifier I to an expression E; each graph is put in a box labeled by I. Subexpressions of E are recursively displayed as long as the whole picture fits in one page.
- The `BPF-mdoc/` directory may also include the following subdirectories:
  - `cpp/` for Faust compiled code;
  - `pdf/` which contains this document;
  - `src/` for all Faust sources used (even libraries);
  - `svg/` for block diagrams, encoded using the Scalable Vector Graphics format (http://www.w3.org/Graphics/SVG/);
  - `tex/` for the LaTeX source of this document.

Figure 6. Common header of the notice.

3.4.2 *Generating the Mathematical Documentation*

The easiest way to generate the complete mathematical documentation is to call the `faust2mathdoc` script on a FAUST file, as the `-mdoc` option leave the documentation production unfinished. For example:

```
faust2mathdoc myfaustfile.dsp
```

The PDF file is then generated in the appropriate directory `myfaustfile-mdoc/pdf/myfaustfile.pdf`.

3.4.3 *Online Examples*

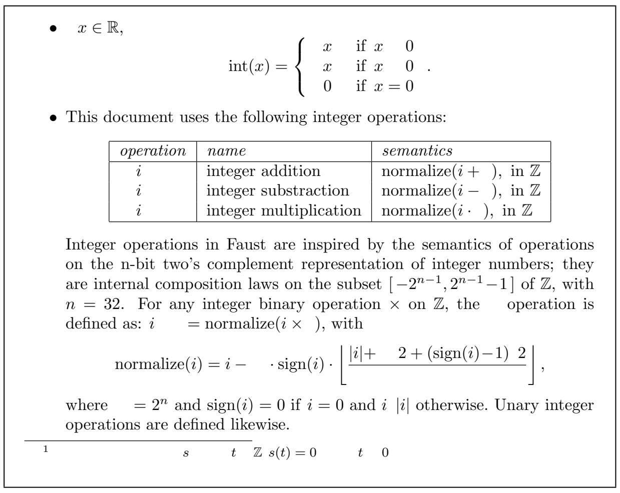
To have an idea of the results of this mathematical documentation, which captures the mathematical semantic of FAUST programs, you can look at two pdf files online:

- http://faust.grame.fr/pdf/karplus.pdf (automatic documentation),
- http://faust.grame.fr/pdf/noise.pdf (manual documentation).

3.5 Conclusion

We have presented two extensions to the FAUST compiler : an *architecture* system that provides OSC support to FAUST generated applications, and an automatic documentation generator able to produce a full mathematical description of any FAUST program.

The idea behind the FAUST's architecture system is *separation of concerns* between the DSP computation itself and its use. It turns out to be a flexible and powerful idea: any new or improved architecture file, like here OSC support, benefits to all applications without having to modify the FAUST code itself. We have also split some of these architectures into separate Audio and UI modules that are



- $x \in \mathbb{R}$,

$$\mathrm{int}(x) = \begin{cases} x & \text{if } x \quad 0 \\ x & \text{if } x \quad 0 \\ 0 & \text{if } x = 0 \end{cases} .$$

- This document uses the following integer operations:

operation	*name*	*semantics*
i	integer addition	normalize($i +$), in $\mathbb{Z}$
i	integer substraction	normalize($i -$), in $\mathbb{Z}$
i	integer multiplication	normalize($i \cdot$), in $\mathbb{Z}$

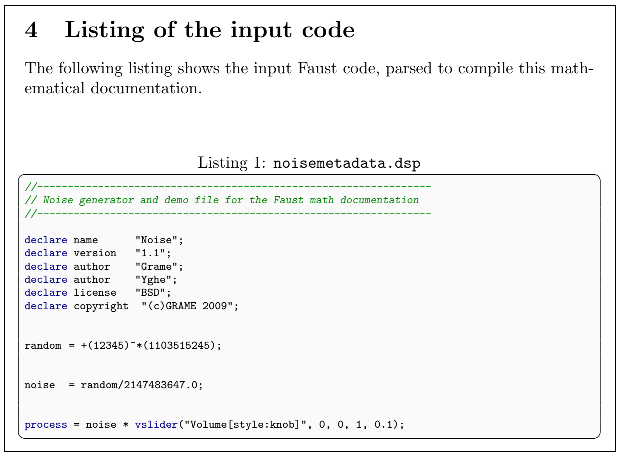
Integer operations in Faust are inspired by the semantics of operations on the n-bit two's complement representation of integer numbers; they are internal composition laws on the subset $[-2^{n-1}, 2^{n-1}-1]$ of $\mathbb{Z}$, with $n = 32$. For any integer binary operation $\times$ on $\mathbb{Z}$, the operation is defined as: $i \quad = \mathrm{normalize}(i \times \ )$, with

$$\mathrm{normalize}(i) = i - \quad \cdot \mathrm{sign}(i) \cdot \left\lfloor \frac{|i| + \quad 2 + (\mathrm{sign}(i) - 1) \quad 2}{\quad} \right\rfloor ,$$

where $\quad = 2^n$ and $\mathrm{sign}(i) = 0$ if $i = 0$ and $i \quad |i|$ otherwise. Unary integer operations are defined likewise.

1 $s \qquad t \quad \mathbb{Z} \ s(t) = 0 \qquad t \quad 0$

Figure 7. Dynamic part of a printed notice.



4 Listing of the input code

The following listing shows the input Faust code, parsed to compile this mathematical documentation.

Listing 1: `noisemetadata.dsp`

```
//-----------------------------------------------------------------
// Noise generator and demo file for the Faust math documentation
//-----------------------------------------------------------------

declare name      "Noise";
declare version   "1.1";
declare author    "Grame";
declare author    "Yghe";
declare license   "BSD";
declare copyright  "(c)GRAME 2009";


random = +(12345)~*(1103515245);


noise  = random/2147483647.0;


process = noise * vslider("Volume[style:knob]", 0, 0, 1, 0.1);
```

Figure 8. Faust code listing.

easier to maintain or evolve. This provides another layer of flexibility.

The self mathematical documentation system, while not simple to develop, turns out to be feasible because FAUST has a simple and well defined semantic. It is therefore possible to compute a semantic description of what a FAUST program does whatever its complexity. Moreover this semantic description was readily available inside the FAUST compiler because already used to optimize the generated C++ code.

This example shows that semantics is not only of theoretical interest and can have very practical benefits. We would like therefore to encourage developers to consider this aspect, as well as preservation issues, when designing new audio/music tools or languages.

3.6 Acknowledgments

This work has been partially supported by the French National Agency (ANR) in the context of the ASTREE, project (ANR-2008-CORD-003-01).

4. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, "An algebra for block diagram languages," in *Proceedings of International Computer Music Conference*, ICMA, Ed., 2002, pp. 542–547.

[2] ——, "Faust : an efficient functional approach to dsp programming," in *New Computational Paradigms for Computer Music*. Editions DELATOUR FRANCE, 2009, pp. 65–96.

[3] A. Graef, "Signal processing in the pure programming language," in *Proceedings of the Linux Audio Conference LAC2009*, 2009.

[4] M. Wright, *Open Sound Control 1.0 Specification*, 2002. [Online]. Available: http://opensoundcontrol.org/spec-1_0

[5] R. Michon and J. O. Smith, "Faust-stk: a set of linear and nonlinear physical models for the faust programming language." in *submitted to DAFx 2011*, 2011.

[6] P. Cook, "The synthesis toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC)*, Beijing, China, Oct., 1999, pp. 299–304.

[7] D. Fober, C. Daudin, Y. Orlarey, and S. Letz, "Interlude - a framework for augmented music scores," in *Proceedings of the Sound and Music Computing conference - SMC'10*, 2010, pp. 233–240.

[8] D. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[9] N. Bernardini and A. Vidolin, "Sustainable live electro-acoustic music," in *Proceedings of the International Sound and Music Computing Conference*, Salerno, Italy, 2005.