# INTEROPERABILITY AND THE MARSYAS 0.2 RUNTIME

*G.Tzanetakis, R.Jones, C.Castillo*
Computer Science
University of Victoria
Canada
gtzan@cs.uvic.ca

*L.G Martins, L.Teixeira*
INESC Porto, Porto
Portugal

*M. Lagrange*
Music Technology
McGill University
Canada

## ABSTRACT

Marsyas is a software framework for building efficient complex audio processing systems and applications. Although originally designed for Music Information Retrieval (MIR) tasks in the past few years it has been expanded to include any type of audio analysis or synthesis. Complex Audio processing systems are defined hierarchically through composition using implicit patching. Both the specification of the processing network and the control of it while data is flowing through can be performed at runtime without requiring recompilation. Compilation is required only when new processing objects need to be defined. Therefore the Marsyas runtime provides considerable functionality and flexibility. In this paper we demonstrate how the Marsyas runtime can be accessed using a variety of different ways allowing non-trivial interactions with common software frameworks and environments.

## 1. INTRODUCTION

Marsyas is an open source software framework for designing and building audio processing systems and applications. It is based on a dataflow model of computation in which any audio processing system is represented as a large network of interconnected basic audio processing units. This basic idea is familiar from a variety of computer music systems that use the unit generator abstraction including the Max/PD family [1], Chuck [2] and CSound [3]. One important difference is that in Marsyas the dataflow network is constructed using implicit patching [4] though hierarchical object composition. The behavior of the network can be controlled dynamically while data is flowing through it using controls. Both the network construction and control are runtime operations and recompilation is only required when new processing units are added to the framework. This runtime provides considerable expressive power while retaining high audio processing performance and in this paper we demonstrate how it can be accessed using a variety of different languages and software environments. Before describing these connections we first review the basic architecture of Marsyas.

### 1.1. Implicit Patching

To assemble audio processing systems, modules are implicitly connected using hierarchical composition. Special *"Composite"* modules such as *Series*, *Fanout*, *Parallel* are used for this purpose. For example, modules added to a *Series* composite will be connected in series, following the order they were added - the first module's output is shared with the second module's input and so on. Moreover, the "tick" method is called sequentially following the same order. Figure 1 shows an example of how composite and non-composite modules can be used. This paradigm differs from typical processing tools based on explicit patching such as CLAM [5], MAX/MSP or PD [1]. In explicit patching the user would first create the modules and then connect them by explicit patching statements. The interested reader may find a comprehensive discussion about the differences between implicit and explicit patching in [4]. A large variety of complex networks performing interesting audio processing tasks can be constructed this way. Examples that have been created with Marsyas include: a polyphonic real-time harmonizer based on multiple instances of a phasevocoder, a feature extraction and classification system for musical genre, and a sound source separation system.

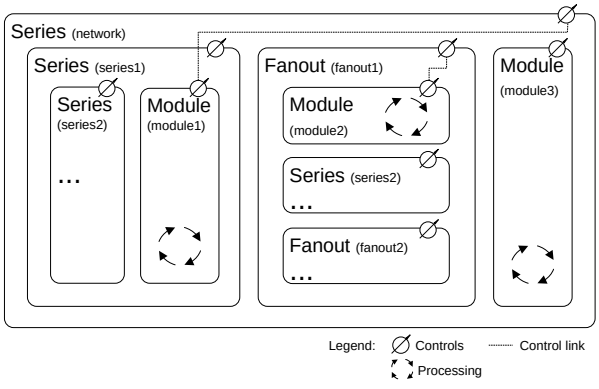


**Figure 1**. Building blocks in Marsyas 0.2.

For example, consider a small network consisting of *Series* module which contains three other modules: a source, a processing module and a sink. When a tick is called in the Series, implicitly the tick method is called in all three modules as shown in Figure 2.

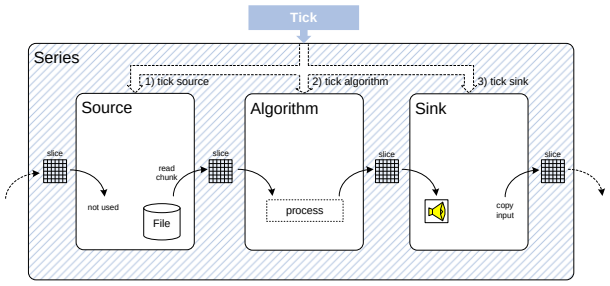


**Figure 2**. Processing chunks of data in Marsyas.

### 1.2. Dynamic access to modules and controls

In Marsyas, each module in a processing network can be accessed by querying the system with a path-like string. Taking the example shown in Figure 1, if we wanted to reach the processing module named `module1`, the query path would be : `/Series/network/Series/series1/Module/module1`. The first "/" indicates the outermost module and the rest of the path is always composed by the concatenation of `Type/Name` strings. This naming scheme was inspired from the way messages are exchanged in Open Sound Control (OSC) [6].It is possible to have access to some of the internal parameters of the modules using *controls*. Each module exports a list of controls which may be of different types (e.g. integers, floats, strings, vectors, or arbitrary user-defined types). They can be accessed for reading or writing by specifying the path to their parent module plus the `Type/Name` corresponding to the control.

Controls can be linked as shown in Figure 1, so that changes to the value of one control are automatically propagated to all the others. There are plenty of interesting uses for this feature: parameter values that must be passed to more than one module in a system; feedback loops where results from modules ahead in the processing network are sent back to the first modules in the chain; shortcuts for other links, etc. Links can be defined (both at compile-time and at run-time) for controls with the same value type, either belonging to a same module, or to any other module in the network. Links can also be used to create *proxy controls* – in order to create a shortcut to a control from a module deep inside other composite modules, it is possible to link it to a *proxy control* in the outmost module, created on demand for this task. This way multiple and easy to understand views for the control of the same algorithm can be created.

## 2. INTEGRATION WITH THE QT FRAMEWORK

Trolltech's Qt [1] is a comprehensive development toolkit that includes features, capabilities and tools that enable the development of cross-platform C++ applications. Such features include multi-platform APIs and classes for the development of Graphic User Interfaces (GUIs), signaling, and multi-threaded execution. In its 4th version, Qt is available as a dual-license software toolkit for all the supported platforms (i.e. Linux, MacOSX and Windows). For open source applications such as Marsyas one of the licenses is open-source GPL. Marsyas, although not bound specifically to Qt, uses this toolkit as its preferred solution for the development of GUIs. Its use is however totally optional, allowing the developer to choose any other library, or even including no GUI support at all (missing in this case all of the GUI extra features already implemented in some Marsyas classes and applications). There are currently two approaches available for GUI development using Qt4 in Marsyas.

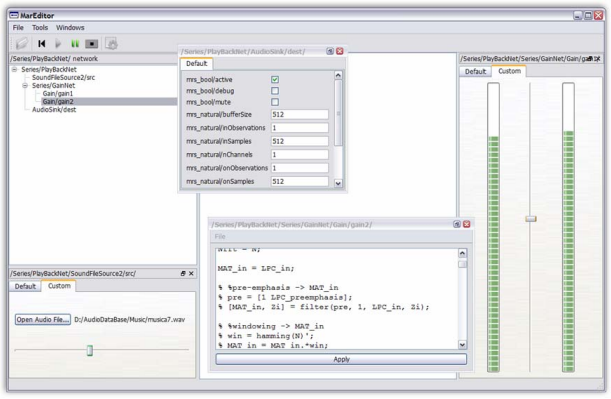

**Figure 3**. Common and specialized GUIs for Marsyas modules.

The first way is using a *delegation design pattern*, where the core C++ Marsyas classes in charge of the actual processing are wrapped by an entity that takes care of all the message passing between the GUIs and the processing network. Additionally, using Qt's multithread features, this wrapper makes sure that GUIs and the processing code are executed in independent threads. This allows the implementation of responsive GUIs and the best use of the last generation multi-core processors. This approach is most suited for the development of customized GUI frontends for Marsyas based applications. It allows interacting with the processing network (by means of reads/writes to its module's controls) in an intuitive and real-time manner.

The second way is conditionally making all Marsyas modules inherit from Qt's base class *QObject*. This automatically embeds Qt's most advanced features (such as *signals* and *slots*) into most Marsyas core classes avoiding the use of middle-layers for message exchange. Additionally to improving efficiency, this approach facilitates the implementation of more advanced functionalities for GUI and multi-threaded processing. As a drawback, this implies a tighter compile-time bind between Marsyas and Qt, which makes independence between the two frameworks more difficult to maintain. This approach allows the implementation of GUIs for all Marsyas modules, such as widgets for viewing/modifying the list of controls from any module, or the creation of specialized GUIs for data plotting or parameter modification (see Figure 3).

[1] `http://www.trolltech.com/products/qt/`

### 2.1. Open Sound Control

Open Sound Control (OSC) [6] is a protocol for communication among computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology. There many implementations of OSC and most computer music environments (such as Max/MSP, PD, Chuck, CSound) have the ability to send and receive open sound control messages.

The control mechanism in Marsyas was inspired from OSC so the mapping of controls to OSC messages is very straightforward. The path notation is used to specify the full name of the control and the value of the message is directly mapped to the value of the control. The mapping of OSC messages to Marsyas controls is part of the Qt4/Marsyas integration code. *OscMapper* is the interface between OSC, Marsyas and Qt4. It acts as both an OSC server and client and allows particular OSC hosts and clients to be associated with particular MarSystems. The communication is abstracted as signals and slots following the way Qt4 structures communication between interface components. The user interface programmer only needs to specify the information about where the OSC messages will be coming from and all the rest is taken care directly by the mapping layer. For example, this way it is straightforward to use PureData to send OSC messages to modify the parameters of a phasevocoder running in Marsyas. The data flowing through a Marsyas network is also accessible through controls so audio information can also be exchanged.

## 3. MATLAB ENGINE

MATLAB is a powerful and widely used tool in several areas of research and development, with a large community of users and available routines for math and multimedia analysis and processing algorithms. Additionally, MATLAB provides easy to use and advanced plotting facilities, a major asset for researchers developing algorithms for audio, image and video processing. Until recently, developers always had to make a hard choice regarding their development language: either opt for the flexibility and ease of use of MATLAB or decide in favor of efficiency and performance as provided by an OOP language like C++. In its latest versions, MATLAB includes the ability to exchange data in run-time with applications developed in Fortran, C or C++, through and API named MATLAB Engine [2] . Marsyas implements a singleton wrapper class for the MATLAB Engine API, enabling Marsyas developers to easily and conveniently send and receive data (i.e. integers, doubles, vectors and matrices) to/from MATLAB in run-time. It is also possible to execute commands in MATLAB from calls in the C++ code as if they have been called in the MATLAB command line. This enables the execution of MATLAB scripts and the access to all MATLAB functions and toolboxes from within Marsyas C++ code. The MATLAB wrapper class in Marsyas provides three basic methods:

1. `PUTVAR(Marsyas_var, MATLAB_var_name);`
2. `GETVAR(Marsyas_var, MATLAB_var_name);`
3. `EVALUATE(MATLAB_cmd);`

By means of function overloading, these three methods allow exchanging different types of variables from Marsyas/C++. They can be called from anywhere in the Marsyas C++ code without any need of changes in the Marsyas interfaces, making it simple to send data structures to MATLAB for convenient inspection and analysis, calculations and plotting, and then get them back in Marsyas for additional processing. These features are only available when MATLAB is installed in the system and Marsyas is built with MATLAB Engine support. Any MATLAB Engine calls in Marsyas code are automatically ignored otherwise. Next is presented a code snippet in C++ illustrating how a vector of real values can be processed and exchanged with MATLAB, using the Marsyas MATLAB Engine wrapper class:

```
// create a std::vector of real numbers
std::vector<double> vector_real(4);
vector_real[0] = 1.123456789;
vector_real[1] = 2.123456789;
vector_real[2] = 3.123456789;
vector_real[3] = 4.123456789;

// send a std::vector<double> to MATLAB
PUTVAR(vector_real, "vector_real");

// do some dummy math in MATLAB
EVALUATE("mu = mean(vector_real);");
EVALUATE("sigma = std(vector_real);");
EVALUATE("vector_real = vector_real/max(vector_real);");

// get values from MATLAB
double m, s;
GETVAR(m, "mu");
GETVAR(s, "sigma");
GETVAR(vector_real, "vector_real");
```

## 4. MARSYAS RUNTIME AS A MAX/MSP EXTERNAL

Max/MSP allows the creation of so called "external" processing units which can be written in C/C++ following a specific API. These externals can then be used as building blocks in the visual programmming environment. We have implemented a general external that can be used to load any audio processing system expressed in Marsyas. The main challenge was to completely decouple the audio buffer rate of Max/MSP from the audio buffer size used by the Marsyas runtime. This is achieved through a dynamic rate adjusting sound source and sound sink for the input and output to the Marsyas part of the patch. For example if the audio buffer size of the Max/MSP patch is 64 samples and Marsyas requires buffers of 256 samples then four buffers of 64 samples will be accumulated before sent to Marsyas for processing. Similarly at the Marsyas output the 256 samples will be broken into 64 sample buffers to be sent back to Max/MSP. Arbitrary sizes are supported and there is no requirement that one buffer should be smaller than the other. This is achieved by using circular buffers with dynamically adjustable length. Controls can be read and written through control outlests and inlets of the external.

[2] `http://www.mathworks.com`

## 5. SWIG BINDINGS

There are only a few commands that need to be supported in order to interface the Marsyas runtime. They basically consist of commands for assembling the dataflow network through hierarchical composition (create, addMarSystem) and commands for linking, updating and setting controls (linkctrl, updctrl, setctrl). SWIG `http://www.swig.org` is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. It includes support for both scripting and non-scripting languages and can be used to create high-level interpreted programming environments. We have used it to provide Marsyas bindings for several programming languages (currently Lua, Ruby, Python, Java).

The following piece of code shows a simple soundfile player written in Ruby using the bindings:

```
msm = Marsyas::MarSystemManager.new
file = msm.create "SoundFileSource","file"
sink = msm.create "AudioSink","sink"
gain = msm.create "Gain", "gain"
net = msm.create "Series","net"
net.addMarSystem file
net.addMarSystem gain
net.addMarSystem sink

fn = net.getctrl "SoundFileSource/file/mrs_string/filename"
ne = net.getctrl "SoundFileSource/file/mrs_bool/notEmpty"
filename.setctrl_string "test.wav"

while ne.to_bool
      net.tick
end
```

A more complex examples shows how feature extraction can be written in Python:

```
import marsyas_python
# Create top-level patch
mng = marsyas_python.MarSystemManager()
fnet = mng.create("Series", "featureNetwork");

# functional short cuts to speed up typing
create = mng.create
add = fnet.addMarSystem
link = fnet.linkControl
upd = fnet.updControl
get  = fnet.getControl

# Add the MarSystems
add(create("SoundFileSource", "src"));
add(create("TimbreFeatures", "featExtractor"));
add(create("TextureStats", "tStats"));
add(create("Annotator", "annotator"));
add(create("WekaSink", "wsink"));

# link the controls to coordinate things
link("mrs_string/filename",
     "SoundFileSource/src/mrs_string/filename");
link("mrs_bool/notEmpty",
     "SoundFileSource/src/mrs_bool/notEmpty");
link("WekaSink/wsink/mrs_string/currentlyPlaying",
     "SoundFileSource/src/mrs_string/currentlyPlaying");
link("Annotator/annotator/mrs_natural/label",
     "SoundFileSource/src/mrs_natural/currentLabel");
link("SoundFileSource/src/mrs_natural/nLabels",
     "WekaSink/wsink/mrs_natural/nLabels");

upd("mrs_string/filename", "bextract_single.mf");
upd("WekaSink/wsink/mrs_string/labelNames",
    get("SoundFileSource/src/mrs_string/labelNames"));

while (get("mrs_bool/notEmpty")):
    fnet.tick();
```

As an example of using the bindings with a compiled language we show how the same network as the Ruby example can be created and used to play sound in Java. This approach utilizes the JNI (Jave Native Interface) and therefore would not be portable across different operating systems. However it allows Java programs to utilize the Marsyas functionality which has much higher run-time performance than a full Java port would have.

```
import edu.uvic.marsyas.*;

class Test {
    static {
        System.loadLibrary("marsyas");
   }
 public static void main (String [] args){
     MarSystemManager msm = new MarSystemManager();

     MarSystem file = msm.create("SoundFileSource","file");
     MarSystem gain = msm.create("Gain", "gain");
     MarSystem sink = msm.create("AudioSink","sink";)
     MarSystem net = msm.create("Series","net");
     net.addMarSystem(file);
     net.addMarSystem(gain);
     net.addMarSystem(sink);

     MarControlPtr filename =
      net.getControl("SoundFileSource/file/mrs_string/filename");
     MarControlPtr notempty =
      net.getControl("SoundFileSource/file/mrs_bool/notEmpty");

     filename.setValue_string("test.wav");
     while (notempty.to_bool()) net.tick();
}
```

## 6. SUMMARY

Marsyas 0.2 (`http://marsyas.sness.net`) is a software audio framework that provides a lot of flexibility and control to the programmer at run-time without requiring recompilation. At the same time it retains the high computational performance of compiled code for the processing units. We demonstrate the power of this runtime functionality through several examples of different interactions within a large software ecology.

## 7. REFERENCES


[1] M. Puckette, "Combining event and signal processing in the MAX graphical programming environment," *Computer Music Journal*, vol. 15, no. 3, pp. 68–77, 1991.

[2] G. Wang and P. Cook, "Chuck: A programming language for on-the-fly, real-time audio synthesis and multimedia," in *ACM Multimedia*, New York, USA, 2004.

[3] R. Boulanger, *The Csound book*, MIT Press, 2000.

[4] S. Bray and G. Tzanetakis, "Implicit patching for dataflow-based audio analysis and synthesis," in *In Proceedings of International Music Conference (ICMC)*, 2005.

[5] X. Amatriain, "CLAM, a framework for audio and music application development," *IEEE Software*, vol. 24, no. 1, pp. 82–85, Jan./Feb. 2007.

[6] M. Wright, A. Freed, and A. Momeni, "Opensound control: State of the art 2003," in *International Conference on New Interfaces for Musical Expression (NIME'03)*, Montreal, Canada, 2003.