A FLEXIBLE AND DYNAMIC C++ FRAMEWORK AND LIBRARY FOR DIGITAL AUDIO SIGNAL PROCESSING

Timothy Place
74 Objects LLC
tim@74objects.com

Trond Lossius
BEK
trond.lossius@bek.no

Nils Peters
McGill University, CIRMMT
nils.peters@mcgill.ca

ABSTRACT

This paper presents an object-oriented, reflective, application framework for C++, with an emphasis on real-time signal processing. The Jamoma Foundation and DSP Library provide a runtime environment and an expanding collection of unit generators for synthesis, processing, and analysis. It makes use of polymorphic typing, dynamic binding, and introspection to create a cross-platform API pulling ideas from languages such as Smalltalk and Objective-C while remaining within the bounds of the portable and cross-platform C++ context. Over the past several years this library has been used in both open source and commercial software projects in the sound and music computing field.

1. INTRODUCTION

"The SMC (Sound and Music Computing) Roadmap identifies two broad research challenges: (1) To design better sound objects and environments and (2) To understand, model, and improve human interaction with sound and music." [32] The Jamoma Foundation and DSP Library directly addresses the first task as a means by which to address the second task. Before discussing the approach and relative merits of the Jamoma project, we will first lay out some definitions and quickly review similar projects.

1.1. Terminology

For the course of the paper, the usage of various computer science jargon and terminology needs to be defined. In *object-oriented programming* functionality related to a set of data is treated as a unit. These units, or objects, are created and then often passed using a reference or pointer to the memory in which the object's contents are stored. These objects comprise *methods* (functions) and *attributes* (properties or data which represent part of an object's state).

Polymorphism is a means by which a programming language generalizes different types of functions or data using a common *API*, or Application Programming Interface. An example of a polymorphic data-type of the variety in which we are interested is a 'var' in the Javascript language [7]. That is to say that a variable may contain any data-type internally (including an object or array), the details of which are not necessary in order to use or pass the data type amongst functions.

Introspection refers to the ability to determine the characteristics of an object at runtime. This means that when handed a pointer in C++, we can take the pointer and query for an object's name, its type or *class*, the messages that it understands, the attributes it possesses, etc. By extension, **reflection** refers to the ability to then modify the behavior of an object at runtime [19]. In practical terms this means adding messages, changing attributes, over-riding audio processing methods, and extending existing instances of objects as the software is executing and without stopping the software to re-compile the code.

Introspection and reflection are often implemented by making use of a **dynamic binding model**. Programming languages such as C++ and Java link function and method calls when a program is compiled, which is known as static binding. A dynamically bound model does not link these functions at compile-time, but instead waits until a method is called at runtime to resolve its address. For this reason, we say we are 'sending messages' to objects when using a dynamic binding model. Dynamic binding is the hallmark of Smalltalk [14], Objective-C [6], and Ruby.

A confusing gaggle of terminology exists for classifying systems of objects throughout the literature of the computer music field. These include *framework*, *library*, *environment*, and *toolkit*. For the purposes of this paper the following definitions will be used. A **unit generator** is a class or object that implements a well-defined DSP task such as generating, analyzing, or processing audio data. A **library** is a collection of pre-built and ready-to-use unit generators. A **toolkit** is a collection of functions, utilities and helpers, possibly with an API, for creating unit generators. A **framework** is an architectural structure that underlies a system of unit generators. A **runtime** is a daemon or framework operating in real-time when a framework, toolkit, or unit generator is in use. A runtime's role is typically for dispatching messages, balancing processor loads, or otherwise running the background machinery such as in the Objective-C or Java runtime environments. An **environment** is a full-fledged system intended for use by an end-user. Examples include Max, SuperCollider, ChucK, DAW applications, and CSound.

1.2. Requirements

The authors are involved in multiple divergent and parallel efforts which requires both a framework for creating unit generators and a library of ready-to-go unit generators. These efforts include both open-source and closed-source commercial applications targeting multiple platforms and environments. To meet the manifold demands of these efforts, we stipulate the following list of requirements regarding how the framework must perform and behave.

- **Licensing** allowing open source and commercial use
- **Cross-platform** compliant (e.g. Mac, Windows, Linux, Embedded Devices, Mobile Platforms)
- **64-bit** audio resolution
- **Multichannel audio support**
- **Reasonably efficient**, i.e. frame-based audio processing, but no cryptic optimizations
- **User-extensible**, adding functionalities without recompiling core frameworks
- **Dynamically reconfigurable** classes at runtime
- **Adaptable** process method for varying input (frame sizes, channel configurations, etc.) on-the-fly
- **Effortless use** of classes in different environments (Max/MSP, VST, AudioUnit, ChucK, Pure Data)

Having met these technical requirements, the authors also deem an additional set of process requirements to be important. These requirements adhere to contemporary philosophies for good coding practice, facilitating the readability, debugging, maintenance, and distribution of code.

- expressive syntax, idioms, and conventions
- adhere to the DRY (Don't Repeat Yourself) principle, which states that "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [11]
- convention over configuration [A1]
- tag-based searching for class categorization and object instantiation
- integrated unit testing and benchmarking [A2]

2. PRIOR ART

A myriad of existing libraries, toolkits, frameworks, and environments are available for digital signal processing. To justify the effort of creating a new framework the merits of the extant members in this field should be considered, particularly with regard to our previously stated requirements.

2.1. Choice of Language

An immediate winnowing of the field of contenders can be accomplished by discussing the choice of programming language. There are popular and well structured DSP libraries for Java [10, 4] and Objective-C [12, 13], for example, but these languages also carry restrictions and baggage. Java is not installed on Windows systems by default and is not available in the context of many embedded devices. Objective-C is available on Windows only through the clumsy and inadequate GnuStep project[1] [A3] and also is not available in the context of many embedded or mobile devices. Interpreted languages such as Ruby and Python are quite powerful, but do not provide the required speed for real-time DSP performance in embedded environments and also create portability problems.

Another class of languages are domain-specific languages for audio signal processing. These include SuperCollider [20], ChucK [34], and CSound. For our purposes we will also consider the Max family (including MSP [35] and PureData [28]) to be domain-specific-languages . These domain-specific languages do provide facilities both for creating and using Unit Generators, but often have portability limitations and very frequently have licensing limitations[2]. These environments also frequently have a large footprint in that they are very resource demanding, while we desire a light-weight framework with a minimum of dependencies.

The C++ language and its compilers are ubiquitous across platforms and capable of creating extremely high-performance code optimized for digital signal processing. Plug-ins and extensions for sundry environments and languages can be compiled using C and C++.

2.2. Plug-in APIs

A related subject is the creation of audio plug-ins using existing APIs. VST, RTAS, LADSPA/DSSI, and AudioUnits are all APIs for creating UnitGenerators in C/C++. None of these technologies, however, meet our requirements. None are actually libraries, though there is a standard set of the non-cross-platform AudioUnit plug-ins provided by Apple.

2.3. Licensing

As stated, the authors require a framework that can be used in both open-source and commercial applications. This immediately rules out the use of any existing work licensed under the GNU GPL, which stipulates that all works using it are themselves licensed under the GNU GPL. Among the options this rules out are SndObj [16], CLAM [1], and

[1]E.g., one cannot natively compile using Microsoft's MSVC compiler.

[2]ChucK and SuperCollider are examples licensed as GPL and not available for commercial development, while Max/MSP is not available on embedded devices nor in plug-in host environments (besides Ableton Live)

Marsyas [33]. Additionally, the CSL framework [27] requires licensing through the University of California.

Due to these licensing restrictions, none of the aforementioned packages are suitable for our use. They do, however, contain many valuable ideas that serve to inform our own work.

2.4. Dynamic Binding

One of our core concerns is the requirement for dynamically reconfigurable classes at runtime. For this, dynamic binding is of critical importance. Dynamic binding is implemented to some degree in many of the aforementioned environments including Marsyas, Max/MSP, and the NeXT SoundKit for Objective-C. Due to our cross-platform and liberal licensing requirements, however, they are not options. Of the remaining DSP libraries and toolkits, the STK [5], CMix [15], and TANGA [30] are all statically-bound.

An interesting middle-of-the-road option is Kronos. In fact, the problem domain of the Kronos system is the same as our problem domain: " the musician may want to change the program during its execution. This was possible in the analog music studio, where swapping out patch cords often resulted in immediate gratification. In the digital world programs often have to be aborted, edited, re-compiled, linked and launched. The all-important musical hacking suffers from such a heavy compilation cycle, making a traditional programming language less desirable for real time artistic expression." [21]

Dynamically-bound frameworks and runtimes, such as PureData, Objective-C, or Marsyas, solve this problem by precompiling the unit generators and then directing messages to these objects at runtime. Kronos takes an alternative approach where the graph of objects, indeed the unit generators themselves, are not precompiled at all but rather compiled 'Just in Time' from a custom meta-language. This results in better performance from the code, while still maintaining much of the flexibility offered by a dynamically-bound runtime. The performance results are compelling. Unfortunately, a just-in-time compilation still requires compilation every time you change the interconnections between objects, and the resulting domain-specific language may be limited to only the domain for which it is written.

One interesting cross-platform, dynamically-bound, and liberally licensed architecture is ZenGarden [A4]. One feature of ZenGarden is the use of PureData as an authoring environment to define a graph of ZenGarden's own unit generators, as was done for the popular RjDj iPhone app [A5]. Unfortunately, at the time of this writing, ZenGarden does not pass our next requirement: 64-bit audio fidelity.

2.5. 64-bit Audio

The higher resolution of 64-bit audio improves stability and fidelity in processes that are vulnerable to numerical instability. This includes IIR filters at extreme settings and signal processing algorithms solving differential equations, as used in physical modeling. If audio signals are used as wavetable lookup indices, a 32-bit float signal with a 24-bit signed mantissa is able to address only 6'20" with sample accuracy at 44.1 kHz and no more than 1'26" at 192 kHz. In a 64-bit audio chain this problem is nonexistent for all practical concerns. The computational overhead required for 64-bit processing is counteracted by increasingly commonplace 64-bit processors.

2.6. Conclusion

In the end we found no one framework to possess all of the requirements set forth in Section 1.2. The de facto standard for audio DSP libraries, the STK, comes perilously close as of more recent revisions [31], and offers a rich and mature array of unit generators. As a statically-bound toolkit, the STK may offer potentially faster runtime execution than a dynamically-bound framework such as the one we propose. For a given application or problem domain the characteristics of each should be weighed, as there are clear justifications for both to coexist within the greater computer music ecology.

3. PLATFORM ARCHITECTURE

3.1. A Layered Framework Structure

It is common to organize software frameworks into a number of layers from low level system services at the bottom towards higher level abstraction at the top. Frameworks may build upon and extend frameworks below them in the hierarchy, but the lower frameworks have no knowledge of the higher frameworks. Divisions between the frameworks help to establish clarity as to what functionalities belong to which part of the system. Examples of such layered architectures are the Open Systems Interconnection network model (OSI) [A6] for layered communications and computer network protocol design or the iPhone Software Architecture. A similar layered approach has been proposed by the authors for spatialization systems [23].

Jamoma, originally conceived as a modular standard for structuring patchers in Max [24], has now evolved into a layered architecture of frameworks providing comprehensive infrastructure for creating computer music systems in general, not just for Max/MSP. This section presents each of the five frameworks currently comprising the Jamoma Platform: Jamoma Foundation, Jamoma DSP, Jamoma Graph and Audio Graph, Jamoma Graphics and Jamoma Modular (see Figure 1). We will emphasize the Foundation and DSP layers to show how we meet the requirements of Section 1.2.

It is possible to use only a subset of the provided frameworks according to a project's complexity. All frameworks within the Jamoma Platform share a common structure. A

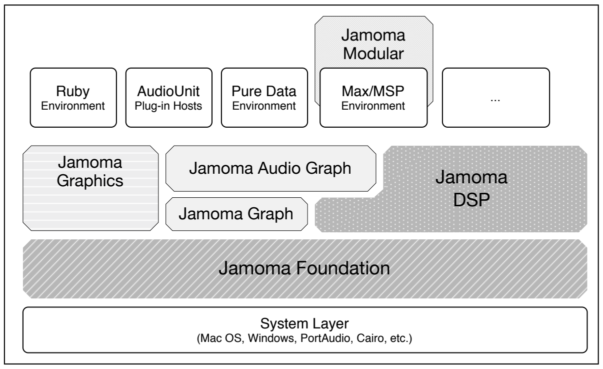


Figure 1. The Jamoma Platform as Layered Architecture.

shared-library implements base classes and core functionality. This functionality is then augmented and enhanced by creating extensions. An extension is a plug-in library dynamically loaded at runtime. In this way the system can be expanded without recompiling core components.

3.2. Foundation

The Jamoma Foundation [A7] is analogous to the Objective-C Foundation, which "Defines the 'nuts and bolts' classes for Objective-C programming" [A8]. The Jamoma Foundation defines base classes including the primary base class, `TTObject`. Object life-cycle facilities include factories for creating, destroying, and referencing these classes. A message-passing and attribute system inspired by Smalltalk and Objective-C is implemented to enable dynamically-bound object topologies.

The Jamoma Foundation classes are informed by many best-practices of software development. Unit Testing is integrated directly into the class design. There is also an emphasis on the use of design patterns [8]. In particular, all objects possess a built-in observer notification system.

Complexity, 'glue code', and the mechanics of writing esoteric C++ are hidden from the programmer as much as possible in adherence with a convention-over-configuration paradigm whereby the clarity of the code is dramatically improved. This results in code that is not only less time consuming to create and maintain, but also more enjoyable. This aim is further aided by emphasizing DRY principles throughout the Jamoma Platform.

Functionality specific to audio or digital signal processing is not present in this particular framework. The Jamoma Foundation is a general multipurpose framework and runtime used as a dynamically-bound API layer for C++. The Jamoma Graphics Library exemplifies this.

3.2.1. Messages and Attributes

As with other similar runtime systems, the Jamoma Foundation defines a symbol table for efficient message dispatch and lookup. This symbol table is leveraged in the implementation of messages and attributes. A *message* is defined as a method of a class or instance that is then bound to a symbol. Messages may optionally possess arguments for passing data to and from the object.

An *attribute* is defined as a data member of a class or instance, whose access is bound to a symbol. Typically, the setting or retrieving of the value then uses a built-in accessor method. If needed, a custom setter or getter method can be defined to override the built-in mechanism.

Additionally, attributes may possess *properties*. Properties are implemented as attributes of the attribute. They include the ability to define ranges for an attribute, the behavior of an attribute's value when the range is exceeded, etc. The design of this system is consistent with the authors' previous proposals for more sophisticated control in parametric systems [25].

3.2.2. Implementation

The Jamoma Foundation relies upon two primary cornerstones to actualize the notion of sending messages to objects and, by extension, setting attributes of objects. The first is a polymorphic data-type, `TTValue`, that enables us to pass data to and from methods regardless of the kind of data actually contained, while using a common interface.

Second, by using `TTValue` to represent any arbitrary data, we are able to abstract the function/method prototype for any message or attribute. By doing so, we create a system by which all communication to and from objects occurs using a singular interface. The messages and attributes of an object are managed internally by fast hash tables of pointers. Changing the contents of the hash table or the values of the pointers at runtime then reconfigures how messages are directed and handled, while not requiring any recompiling or relinking of the code.

`TTObject` implements an observer pattern by maintaining linked-lists of other objects that wish to be notified of events. The objects registered as observers are then sent messages from `TTObject`. If the objects respond to the message, then they can respond appropriately. If they do not respond to the message, then they will simply ignore it. In this case no intervention is required by the user or programmer, and this situation is not considered as an error. In a statically-bound system this situation would likely lead to a compile-failure, thrown exceptions, crashes during operation, or worse.

3.3. DSP Layer

The Jamoma DSP Layer [A9] augments the Foundation by extending `TTObject` to create a new `TTAudioObject` base class. `TTAudioObject` provides the core functionality for processing multichannel, 64-bit, audio samples singly or in blocks, while providing basic thread protection. It also provides attributes and audio processing methods to control muting, bypass, sample-rate, and others that are inherited by subclasses.

In addition to the framework and base-classes, the DSP layer also provides toolkit functionality with convenience functions for filtering denormals and dc-offsets, and a library of classes as extensions implementing a variety of unit generators. The provided unit generators include basic trigonometry functions, filters, oscillators, noise generators, analysis, effects, etc. These classes are organized and classified on several different levels. First, every object is classified using tags when the object is registered with the Foundation at runtime. The Foundation then manages this registry and its metadata for use by the factory methods for creating instances. Secondly, the classes are organized into dynamically loaded *extensions* that share common interfaces and functionality.

3.3.1. Extensions

User extensions may be created for any of the layers built-upon the Jamoma Foundation. At this time, extensions are almost exclusively for the purpose of creating unit generator classes with Jamoma DSP. An extension may implement zero or more classes, which are registered with the Foundation when the extension is loaded.

The extensions included with Jamoma DSP are organized into groups of classes that share a common interface. For example, the *FilterLib* implements more than two dozen audio filters including Butterworth and Linkwitz-Riley algorithms for various frequency responses. All classes in the FilterLib use shared semantics for defining message and attribute names, such as ‘frequency’, and can thus be easily substituted for one another. Similarly, the *FunctionLib* implements a number of algorithms designed for use in gestural mapping scenarios.

While common attribute and message names are preferred, some unit generators will necessarily provide additional controls when compared to simpler or different classes. When substituting one class for another, we leverage the dynamically-bound architecture because we can send messages to an object, even if it does not understand them. In this case the messages are simply ignored. With introspection features, all classes can be queried to find out what attributes they do possess, what ranges characterize those attributes, etc. Objects can also be modified at runtime to add handling for messages not envisaged at compile time using reflective techniques, allowing them to be adapted for use in different contexts.

In addition to the FilterLib and FunctionLib, the growing number of extensions for Jamoma DSP include the AnalysisLib, GeneratorLib, MathLib, EffectsLib, and WindowFunctionLib.

3.4. Additional Layers

Additional layers have been created on top of the Jamoma Foundation, both in series and in parallel with the DSP layer.

The **Graphics Layer** [A10], based on Cairo [A11], provides a platform-independent and host-independent way to create 2D graphical user interfaces (GUI). It has already been used in Max/MSP and for AudioUnit plug-ins.

The **Graph Layer** provides a means by which Jamoma Foundation objects may be networked (’patched’) together to perform either synchronous or asynchronous tasks. The **Audio Graph Layer** [A12] specializes the Graph Layer to combine unit generators from Jamoma DSP into audio processing topologies. Many of the initiatives reviewed in Section 2 conflate both a means to create unit generators and a method by which those objects are combined into audio graphs. In the Jamoma Platform we provide a clear division between creating and using unit generators versus combining them into a graph. This allows the unit generators to be assembled in any way that seems desirable for a particular context.

The **Modular Framework** [A13], the highest layer of the Jamoma Platform, provides a modular structure for developing and controlling Max/MSP/Jitter patches [24]. It builds upon the Foundation, DSP, Graphics, and Graph/AudioGraph frameworks.

3.5. Ruby Language Bindings

As a dynamically-bound API, the Jamoma Foundation is a natural fit for control from the Ruby environment. Language bindings for Jamoma in Ruby exist via the Jamoma Ruby project [A14]. This enables use in a wide range of applications including live coding using *irb* [A15] and integration with web applications using *Ruby on Rails* [A16].

4. APPLICATIONS

The technical underpinnings of the Jamoma Foundation and DSP Library lend themselves to a variety of applications. Jamoma DSP’s API provides a clear interface for creating and using unit generators in the C++ and Ruby languages. Additional tools and facilities tailored to the requirements of various host environments make Jamoma DSP’s unit generators readily accessible.

4.1. Max/MSP

The easiest way to create a Max/MSP external using a Jamoma DSP unit generator is by using the provided Jamoma DSP “Class Wrapper”. The class wrapper takes an existing class and creates all of the required bindings and interfaces for the target environment. Following is the complete code listing for an N-channel MSP external that wraps the Jamoma DSP lookahead-limiter class.

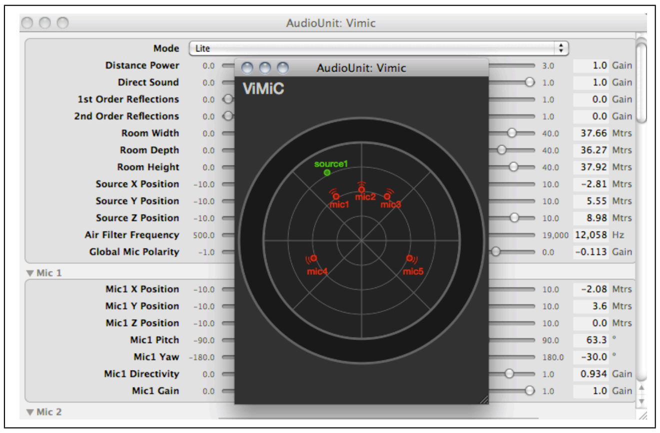
```
#include "TTClassWrapperMax.h"
int main(void)
{
  TTDSPInit();
  return wrapTTClassAsMaxClass(TT("limiter"),
          "myMaxLimiter~", NULL);
}
```

Besides simply wrapping existing classes there are other ways to create Jamoma DSP-based externals. One example is the *jcom.filter∼* external provided with Jamoma DSP. This object dynamically loads any of the filter classes from the DSP Library. A filter is loaded and swapped on-the-fly without requiring Max's DSP chain to be rebuilt. The code for the *jcom.filter∼* external searches the Foundation's object registry for available filter classes to present a list of choices for the user. This list will then be dynamically updated if new filters are added, also without the need to recompile the external.

4.2. Plug-ins

Jamoma DSP provides several example projects that include VST and AudioUnit plug-ins. AudioUnit plug-ins requiring only a generic interface may be created using a class wrapper similar to the one for creating Max/MSP externals.

The spatial sound rendering technique ViMiC (Virtual Microphone Control) is a virtual multi-microphone recording environment [2]. ViMiC is implemented and compiled as a Max external as well as an AudioUnit plug-in. The virtual microphones and sound sources can be controlled and manipulated in a custom interface developed using Jamoma Graphics. Additional control features are accessible using the generic AudioUnit interface (see Fig. 2).



```
Mode                     Lite
Distance Power           0.0      3.0      1.0 Gain
Direct Sound             0.0      1.0      1.0 Gain
1st Order Reflections    0.0      1.0      0.0 Gain
2nd Order Reflections    0.0      1.0      0.0 Gain
Room Width               0.0     40.0    37.66 Mtrs
Room Depth               0.0     40.0    36.27 Mtrs
Room Height              0.0     40.0    37.92 Mtrs
Source X Position      -10.0     10.0    -2.81 Mtrs
Source Y Position      -10.0     10.0     5.55 Mtrs
Source Z Position      -10.0     10.0     8.98 Mtrs
Air Filter Frequency   500.0   19,000   12,058 Hz
Global Mic Polarity     -1.0      0.0   -0.113 Gain
▼ Mic 1
Mic1 X Position        -10.0     10.0    -2.08 Mtrs
Mic1 Y Position        -10.0     10.0      3.6 Mtrs
Mic1 Z Position        -10.0     10.0      0.0 Mtrs
Mic1 Pitch             -90.0     90.0     63.3 °
Mic1 Yaw              -180.0    180.0    -30.0 °
Mic1 Directivity         0.0      1.0    0.934 Gain
Mic1 Gain                0.0      1.0      1.0 Gain
▼ Mic 2
```

Figure 2. ViMiC AU custom (front) and generic UI (back).

4.3. Ruby, IRB, Rails

Ruby Language Bindings make the entirety of the DSP library accessible in the Ruby environment. Together with Jamoma Audio Graph, Ruby's interactive shell (irb) can be used for live-coding by creating objects and manipulating the graph of those objects in real-time. The following code shows a very simple irb session creating an instance of a lowpass filter and interactively passing values one at a time through the filter.

```
# irb
>> require 'TTRuby'
JamomaFoundation -- Version 0.6
JamomaDSP -- Version 0.6
=> true
>> f = TTRuby.new("lowpass")
=> #<TTRuby:0x1011db170>
>> f.calculate(1.0)
=> 0.25
>> f.calculate(1.0)
=> 0.4375
```

Leveraging the popular Ruby on Rails web framework, Jamoma DSP classes are made available for a multitude of web applications. The authors have begun employing one such Ruby on Rails application to graph related functions in real-time for comparison and algorithm analysis.

4.4. Additional Environments

Jamoma DSP includes further example projects wrapping classes for PureData and SuperCollider.

5. DISCUSSION AND FUTURE WORK

The class wrapper demonstrated in Section 4.1 illustrates the power of the Jamoma Foundation for use as a rapid prototyping environment. It leverages all of the features of dynamic binding to make classes available to many different environments on many platforms with a minimal amount of coding and effort required. We can combine objects together in the environment of our choice: Max, Pd, DAWs by means of plug-ins, or a web-browser using Ruby on Rails. This allows for easy code transfer between environments. In the future the class wrappers can be expanded to cover even more environments, including VST, SuperCollider, and CSound.

The Jamoma frameworks are all user-extensible through the creation of extension classes that are loaded and registered at runtime. We are continuing to add support for more audio processing algorithms in the Jamoma DSP, including spectral processing and granulation. We also plan to add support for additional spatialization algorithms such as VBAP [29], Ambisonics [9, 26] and DBAP [18] to support ongoing development on spatialization within Jamoma Modular [23].

A myriad of mundane, but critical, details for DSP classes are taken care of by the DSP framework. This includes a 64-bit audio signal class that automatically adapts its channel configurations and vector-sizes based on input. For developers, multi-threaded environments are the source of many perpetual headaches. Jamoma DSP applies a lightweight thread-protection model where needed, but avoids causing performance problems.

Introspection features of the Jamoma Foundation make it possible to query objects to automate the process of creating mappings and advanced control of the objects such as those cataloged in [22].

A strength of the architecture of Jamoma DSP is the ease with which one can combine, connect, reconnect, and reconfigure unit generators on the fly. This characteristic lends itself to exploring a variety of paradigms for connecting unit generators into processing graphs. The initial development of Jamoma Audio Graph implements an explicitly constructed topology of unit generators using dynamic multichannel connections between nodes in the graph. We have begun discussing alternative models that resemble the implicit patching pattern of Marsyas [3]. In particular we have initiated a design for processing signals through a grouping of objects into an array.

The GNU LGPL license chosen for the Jamoma frameworks have enabled them to be used for open-source as well as commercial software development. Electrotap's Tap.Tools [A17] is a collection of externals for Max/MSP with an emphasis on audio effects processing. Hipno [A18] was a set of audio effects plug-ins in the VST, RTAS and AudioUnit formats using the now discontinued Cycling'74 Pluggo environment. Plug-in development expanding upon the ideas from Hipno is underway, using the Jamoma Platform as a means of gaining independence from the restrictions of proprietary frameworks.

6. SUMMARY

Jamoma Foundation and DSP provide a flexible, user-extendable, runtime environment for creating and using audio and digital signal processing objects. Due to its advanced use of dynamic binding and message-passing paradigm, the building blocks can be reconfigured at runtime without requiring re-compilation. The unit generators themselves are compiled as C++, and by performing block-processing we retain the performance characteristics of a compiled language.

Perhaps more important, but more difficult to quantify, we believe we have created a context in which code is 'pleasant to work with'. As stated in [17], "A well-defined API can also speed up the development process, since the implementation can focus more on the algorithmic aspects and less on implementation issues like API design."

The power of this runtime is demonstrated through the ability to compile objects for Max, Pd, AudioUnits, VST, and Jamoma Audio Graph.

7. ACKNOWLEDGEMENTS

We would like to thank Dave Watson, Théo de la Hogue, Tristan Matthews, and Joshua Kit Clayton for their feedback and contributions to the development of Jamoma DSP.

References

[1] X. Amatraian, P. Arumi, and D. Garcia, "A framework for efficient and rapid development of cross-platform audio applications," *Multimedia Systems*, vol. 14, pp. 15–32, 2008.

[2] J. Braasch, N. Peters, and D. L. Valente, "A loudspeaker-based projection technique for spatial music applications using virtual microphone control," *Computer Music Journal*, vol. 32, no. 3, pp. 55 – 71, 2008.

[3] S. Bray and G. Tzanetakis, "Implicit patching for dataflow-based audio analysis and synthesis," in *Proc. of the 2005 International Computer Music Conference*, 2005.

[4] P. Burk, "JSyn – a real-time synthesis API for Java," in *Proc. of the 1998 International Computer Music Conference*, Ann Arbor, USA, 1998.

[5] P. Cook and G. Scavone, "The synthesis toolkit (STK)," in *Proc. of the 1999 International Computer Music Conference*. Beijing, China: ICMA, 1999.

[6] B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*. Addison Wesley, 1987.

[7] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., 2002, pp. 46–47.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] M. Gerzon, "Surround-sound psychoacoustics: Criteria for the design of matrix and descrete surround-sound systems." *Wireless World. Reprinted in An anthology of articles on spatial sound techniques, part 2: Multichannel audio technologies. Edited by F. Rumsey. Audio engineering society, Inc, 2006.*, pp. 483–486, December 1974.

[10] M. Guillemard, C. Ruwwe, and U. Zölzer, "J-dafx - digital audio effects in java," in *Proc. of the 8th Int. Conference on Digital Audio Effects (DAFx-05)*, Madrid, Spain, 2005.

[11] A. Hunt and D. Thomas, *The Pragmatic Programmer*. Addison-Wesley, 1999.

[12] D. Jaffe and L. Boynton, "An overview of the sound and music kits for the next computer," *Computer Music Journal*, vol. 13, no. 2, pp. 48–55, Summer 1989.

[13] D. A. Jaffe, "Musical and extra-musical applications of the next music kit," in *Proc. of the 1991 International Computer Music Conference*, Montreal, Canada, 1991.

[14] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view-controller user interface paradigm in smalltalk-80," *JOOP*, August September 1988.

[15] P. Lansky, "The architecture and musical logic and cmix," in *Proc. of the 1990 International Computer Music Conference*, 1990.

[16] V. Lazzarini, "Sound processing with the sndobj library: An overview," in *Proc. of the COST G-6 Conference on Digital Audio Effects (DAFX-01)*, Limerick, Ireland, 2001.

[17] A. Lerch, G. Eisenberg, and K. Tanghe, "Feapi: A low level feature extraction plugin API," in *Proc. of the 8th Int. Conference on Digital Audio Effects (DAFx-05)*, Madrid, Spain, 2005.

[18] T. Lossius, P. Baltazar, and T. de la Hogue, "DBAP - Distance-Based Amplitude Panning," in *Proc. of 2009 International Computer Music Conference*, Montreal, Canada, 2009, pp. 489–492.

[19] J. Malenfant, M. Jacques, and F.-N. Demers, "A tutorial on behavioral reflection and its implementation," in *Proc. of the Reflection '96 Conference*, Xerox Palo Alto Research Center, San Francisco, CA, April 1996, pp. 1–20.

[20] J. McCartney, "SuperCollider: a new real time synthesis language," in *Proc. of the 1996 International Computer Music Conference*, Hong Kong, China, 1996, pp. 257– 258.

[21] V. Norilo and M. Laurson, "Kronos - a vectorizing compiler for music dsp," in *Proc. of the 12th Int. Conference on Digital Audio Effects (DAFx-09)*, 2009.

[22] C. Pendharkar, M. Gurevich, and L. Wyse, "Parameterized morphing as a mapping technique for sound synthesis," in *Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06)*, 2006.

[23] N. Peters, T. Lossius, J. Schacher, P. Baltazar, C. Bascou, and T. Place, "A stratified approach for sound spatialization," in *Proc. of 6th Sound and Music Computing Conference*, Porto, Portugal, 2009, pp. 219–224.

[24] T. Place and T. Lossius, "Jamoma: A modular standard for structuring patches in Max," in *Proc. of the 2006 International Computer Music Conference*, New Orleans, US, 2006, pp. 143 – 146.

[25] T. Place, T. Lossius, A. R. Jensenius, and N. Peters, "Flexible control of composite parameters in Max/MSP," in *Proceeding of the International Computer Music Conference*, T. I. C. M. Association, Ed., 2008, pp. 233–236.

[26] M. A. Poletti, "A Unified Theory of Horizontal Holographic Sound Systems," *JAES*, vol. 48, no. 12, pp. 1155–1182, December 2000.

[27] S. T. Pope and C. Ramakrishnan, "The create signal library ("sizzle"): Design, issues, and applications," in *Proc. of the 2003 International Computer Music Conference*, Singapore, 2003.

[28] M. Puckette, "Pure Data: another integrated computer music environment." in *Proc. of the 1996 International Computer Music Conference*, Hong Kong, China, 1996.

[29] V. Pulkki, "Virtual sound source positioning using vector base amplitude panning," *J. Audio Eng. Soc.*, vol. 45(6), pp. 456–466, 1997.

[30] U. Reiter, "Tanga - an interactive object-based real time audio engine," in *Conference Proceedings - Audio Mostly 2007*, 2007.

[31] G. Scavone and P. Cook, "RtMidi, RtAudio, and a synthesis toolkit (STK) update," in *Proc. of the 2005 International Computer Music Conference*. Barcelona, Spain: ICMA, 2005.

[32] X. Serra, "The origins of DAFX and its future within the sound and music computing field," in *Proc. of the 10th Int. Conference on Digital Audio Effects (DAFx-07)*, Bordeaux, France, 2007.

[33] G. Tzanetakis, R. Jones, C. Castillo, L. G. Martins, L. F. Teixeira, and M. Lagrange, "Interoperability and the marsyas 0.2 runtime," in *Proc. of the 2008 International Computer Music Conference*, Belfast, UK, 2008, pp. 588 – 591.

[34] G. Wang, "The ChucK audio programming lanuage: A strongly-timed and on-the-fly envon/mentality," Ph.D. dissertation, Princeton University, 2008.

[35] D. Zicarelli, "An extensible real-time signal processing environment for Max," in *Proc. of the 1998 International Computer Music Conference*. Ann Arbor, Michigan, USA: San Francisco: ICMA, 1998, pp. 463–466.

Web Resources

[A1] `http://softwareengineering.vazexqi.com/files/pattern.html`

[A2] `http://www.extremeprogramming.org/rules/unittests.html`

[A3] `http://www.gnustep.org`

[A4] `http://github.com/mhroth/ZenGarden`

[A5] `http://rjdj.me`

[A6] `http://en.wikipedia.org/wiki/OSI_model`

[A7] `http://github.com/jamoma/JamomaFoundation`

[A8] `http://developer.apple.com/cocoa`

[A9] `http://github.com/jamoma/JamomaDSP`

[A10] `http://github.com/jamoma/JamomaGraphics`

[A11] `http://cairographics.org`

[A12] `http://github.com/jamoma/JamomaAudioGraph`

[A13] `http://github.com/jamoma/JamomaModular`

[A14] `http://github.com/jamoma/JamomaRuby`

[A15] `http://ruby-doc.org/docs/ProgrammingRuby/html/irb.html`

[A16] `http://rubyonrails.org`

[A17] `http://electrotap.com/taptools`

[A18] `http://74objects.com/hipno`

Note, all quoted web resources in this document were updated on September 18, 2010.