

***MARSYAS-0.2: a case study in implementing Music
Information Retrieval Systems***

George Tzanetakis (gtzan@cs.uvic.ca)

Department of Computer Science

(also cross-listed in Music, Electrical and Computer Engineering)

Engineering/Computer Science Building (ECS), Room 504

University of Victoria

3800 Finnerty Road

Victoria, BC

Canada V8P 5C2

Phone: 250 472 5711

Fax: 250 472 5708

Abstract

Marsyas, is an open source audio processing framework with specific emphasis on building Music Information Retrieval systems. It has been under development since 1998 and has been used for a variety of projects in both academia and industry. In this chapter, the software architecture of Marsyas will be described. The goal is to highlight design challenges and solutions that are relevant to any MIR software.

Introduction

Music has always been transformed by advances in technology. Examples of technologies that transformed the way music was produced, distributed and consumed include musical instruments, music notation, recording and more recently digital music storage and distribution. Recently portable digital music players have become a familiar sight and online music sales have been steadily increasing. It is likely that in the near future anyone will be able to access digitally all of recorded music in human history. In order to efficiently interact with the rapidly growing collections of digitally available music it is necessary to develop tools that have some understanding of the actual musical content. Music Information Retrieval (MIR) is an emerging research area that deals with all aspects of organizing and extracting information from music signals.

In the past few years, interest in Music Information Retrieval (MIR) has been steadily increasing. MIR algorithms, especially when analyzing music signals in audio format, typically utilize state-of-the-art signal processing and machine learning algorithms. The large amounts of data that is processed together with the huge computational requirements of audio processing can stress current hardware to its limits. Therefore

efficient processing is critical for building functional MIR systems that scale to large collections of music and eventually to all of recorded music. Moreover, MIR is an inherently interdisciplinary field with practitioners with varying degrees of computer and programming expertise (examples of fields involved include musicology, information science, and cognitive psychology). Therefore it is desirable for MIR systems to support multiple hierarchical levels of usage and extensibility. These issues make the design and development of MIR systems and frameworks especially challenging.

MARSYAS (*M*usic *A*nalysis, *R*etrieval and *S*ynthesis for *A*udio *S*ignals), is an open source audio processing framework with specific emphasis on building MIR systems. It has been under development since 1998 and has been used for a variety of projects both in academia and industry. The guiding principle behind the design of *MARSYAS* has always been to provide a flexible, expressive and extensive framework without sacrificing computational efficiency. Addressing these conflicting requirements is the major challenge facing the software engineer of MIR systems.

The main objective of this chapter is to describe the software architecture of *MARSYAS* using examples from specific MIR applications. We highlight the design challenges and corresponding solutions that are probably relevant to any MIR software system. In many cases these solutions are informed by ideas originating from other fields of Computer Science and Software Engineering but have to be adapted to the particular needs and constraints of MIR research. After reviewing related work and background information, *MARSYAS* is described in the following subsections: History, Requirements, Architecture

and Projects. The next section (Specific Topics) describes in more detail specific topics that we believe are especially important for audio processing and how we have tried to address them in the framework. The chapter concludes with a description of future trends in *MARSYAS* and audio processing software frameworks in general. One of the major dilemmas facing any MIR researcher is whether to use existing tools or develop their own. By describing the tradeoffs and challenges we have faced with the design of our system we hope to help researchers make more informed decisions. Finally an underlying theme of this chapter is the importance of open source software for research and how it is different from other areas of open source development.

Background

Music Information Retrieval is a new area of content-based multimedia information retrieval. Although there was sporadic earlier work, a good reference starting point is the first international conference on MIR (ISMIR) which was held in 2000. These conferences (ISMIR) have been a forum for bringing together music researchers, audio engineers, computer scientists, musicologists, librarians, and music industry (Futrelle and Downie, 2002) MIR with audio signals typically requires signal processing and machine learning algorithms in order to achieve tasks such as classification, similarity-retrieval and segmentation.

MARSYAS 0.2, the software framework for audio analysis and synthesis described in this paper, evolved from *MARSYAS 0.1* (Tzanetakis and Cook, 2000), a framework that focused mostly on audio analysis. One of the motivating factors for the rewrite of the

code and architecture was the desire to add audio synthesis capabilities and was influenced by the design of the Synthesis Toolkit (Cook and Scavone, 1999). Other influences include the powerful but more complex architecture of CLAM (Amatriain, 2005), the patching model and strong timing of Chuck (Wang and Cook, 2003), and ideas from Aura (Dannenberg and Brandt, 1996). The matrix model used in *Implicit Patching* was influenced by the design of SDIFF and the default naming scheme for controls is inspired by the Open Sound Control (OSC) protocol (Wright and Freed, 1997). The code structure reflects many ideas from Design Patterns (Gamma et al., 1995).

The idea of dataflow programming has been fundamental in the design of *MARSYAS*. Dataflow programming has a long history. The original (and still valid) motivation for research into dataflow was to take advantage of parallelism. Motivated by criticisms of the classical von Neumann hardware architecture such as (Ackerman, 1982) dataflow architectures for hardware were proposed as an alternative in the 1970s and 1980s. During the same period a number of textual dataflow languages such as Lucid (Wadge and Ashcroft, 1978) were proposed. Despite expectations that dataflow architectures and languages would take over from von Neumann concepts this didn't happen. However during the 1990s there was a new direction of growth in the field of dataflow visual programming languages that were domain specific. In such visual languages programming is done by connecting processing objects with "wires" to create "patches". Successful examples include Labview (<http://www.ni.com/labview/>), SimuLink (<http://www.mathworks.com/products/simulink/>) and in the field of Computer Music Max/MSP (Zicarelli, 2002) and Pure Data (Puckette, 2002). A recent comprehensive

overview of dataflow programming languages can be found in (Johnston et al., 1985). Another recent trend has been to view dataflow computation as a software engineering methodology for building systems using existing programming languages (Manolescu, 1997). A comprehensive overview of audio processing frameworks from a Software Engineering perspective can be found in (Amatriain, 2005). More recently frameworks specific to MIR have been introduced. They include ACE and JAudio (McKay et al, 2006) and D2K/M2K (Downie and Futrelle, 2005).

More detailed information about some of the topics discussed in this chapter can be found in conference publications. Implicit Patching was introduced in (Bray and Tzanetakis, 2005a) and Flexible Scheduling in (Burroughs et al., 2006). Experiments with distributed audio feature extraction were described in (Bray and Tzanetakis, 2005b).

Description of Marsyas

MARSYAS is a software framework for rapid prototyping, design and experimentation with audio analysis and synthesis with specific emphasis on processing music signals in audio format. In this section we examine the history and motivation behind the design and development of MARSYAS. Some of the requirements used to design the latest version that are applicable to any MIR system are also discussed. The section ends with an overview of the software architecture of the framework.

History

The design and development of *MARSYAS* was initiated by George Tzanetakis while he was studying at Princeton University as part of his graduate research under the supervision of Perry Cook. The motivating application was a reimplementing of a well known music/speech discriminator work [Scheirer and Slaney, 1997]. In general, implementing an existing algorithm provides a much deeper understanding of how it works and in many cases suggests directions for improvement or extensions. Even though the initial version was only used internally at Princeton it was designed to be a general framework with reusable components rather than a specific implementation. Gradually *MARSYAS* was used to conduct new research in audio analysis and retrieval. Since then every publication by George Tzanetakis and many by others have used *MARSYAS*. In 2000 it was clear that certain major design decisions needed to be revised. A major rewrite/redesign of the framework was initiated and the first “public” version was released (numbered 0.1). Soon after Sourceforge (<http://sourceforge.net/index.php>) was used to host *MARSYAS*. Version 0.1 is still widely used by a variety of academic groups and industry around the world. It is well known in Software Engineering that even though major rewrites can be time consuming they are inevitable for complex software and typically result in much better code structure and organization. A second major rewrite was initiated in 2002 while George Tzanetakis was a PostDoctoral Fellow at Carnegie Mellon University working with Roger Dannenberg. The motivation was the desire to port algorithms from the Synthesis Toolkit (STK) (Cook and Scavone, 1999) into *MARSYAS*. This effort as well as many interesting discussions with Roger Dannenberg

informed the new design. The new version (numbered 0.2) uses a dataflow model of audio computation with general matrices (**Slices**) instead of 1-D arrays as data and an Open Source Control (OSC) (Wright and Freed, 1997) inspired hierarchical messaging system used to control the dataflow network. This is the version described in this chapter. Even though not immediately obvious, supporting audio synthesis is important for a successful MIR software framework. The main reason is that the ability to hear the results of audio analysis algorithms can be very useful for debugging and understanding algorithms. For example the ability to listen to extracted chords or beat patterns can provide insights that are impossible to achieve just by looking at numbers or graphs.

Since the beginning, a major decision was to provide *MARSYAS* as free software under the GNU public license (GPL). Some of the common motivations and reasons behind free software in general are: 1) freedom to modify the software to suit the needs of individual users 2) better quality by having many people working and looking at the source code 3) support for multiple operating systems and porting by the ability to change the code appropriately 4) easier adoption/lower cost compared to proprietary software. In addition to these motivations there are additional specific motivations behind free software when it is used in research. These include: 1) source code is a means of communication. This is especially important in research where publications can not possibly describe all the nuances and details of how an algorithm is implemented, 2) replication of experiments is essential for progress in research especially in new emerging areas such as MIR. For complex systems and algorithms it is almost impossible to know if a reimplementations is correct and therefore the ability to run the original code is crucial, 3) researchers have

limited financial resources and therefore making the software free encourages adoption. One of the common objections to free software is why should someone invest all this time and effort into building something and then give it away for free. Most academics are comfortable with this idea as they do so with publications, reviewing and other activities. Even though many of these activities might not have direct financial reward they are part of the responsibilities of any academic. In addition there are indirect benefits with developing research software such as peer recognition, opportunities for collaboration and international visibility. Finally as will be described in more detailed in the section about projects it is possible to use free software developed in academia in industrial settings and receive money for it.

A frequent dilemma facing graduate students and researchers is whether to build their own tools or use existing ones. This decision frequently involves a tradeoff between short and long time investment. For example using a combination of existing tools a particular task might be accomplished in 1 hour. By spending a week writing your own tool the task might be accomplished in 1 minute. Whether programming for a week is worth the trouble is a tough question. Even though it is impossible to provide a definite answer we discuss how our experiences with *MARSYAS* can inform this choice. If the task at hand can take hours, as MIR algorithms frequently do, runtime performance becomes a critical issue that can affect research. For example, if an experiment that used to take 5 hours takes 10 minutes it can be executed multiple times with different parameters to find the best choice. Programming is a very time consuming task so choosing to build your own tools works best if you have a large enough timeframe to do it. For example a PhD

student has a better chance at completing a significant software framework than a Masters student pressed for time. A supportive advisor is also important and the best way to achieve this is to provide earlier proof that the time you spend developing software pays off in terms of research results. Finally an important consideration is that the development of the software framework itself especially in emerging applications such as MIR is research in itself. *MARSYAS* has been used a test bed for many ideas in Software Engineering and Computer Science that arise based on the particular characteristics and constraints of audio processing.

MARSYAS can be obtained from <http://marsyas.sourceforge.net>. It is written in portable C++ (as much as possible) and it compiles in Linux, OS X, Cygwin, and Windows Visual Studio 2003 and 2005. Subversion is used for version control and the latest unstable source code can be obtained from the webpage.

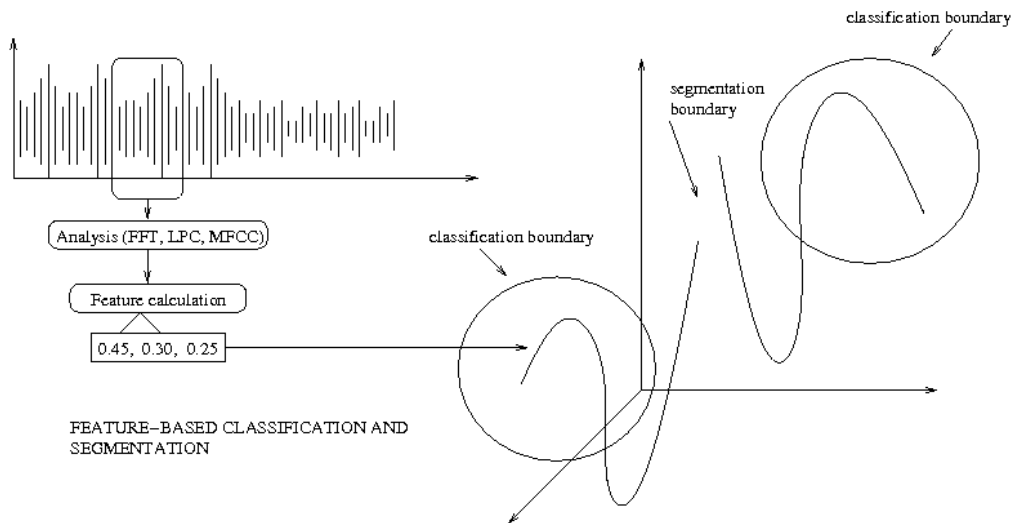


Figure 1: Audio feature extraction, segmentation and classification

Requirements

The canonical application of MARSYAS is audio feature extraction (Tzanetakis and Cook, 2002) which forms the basis of many MIR algorithms such as classification, segmentation, and similarity retrieval. Figure 1 shows a schematic diagram of audio feature extraction and how it can

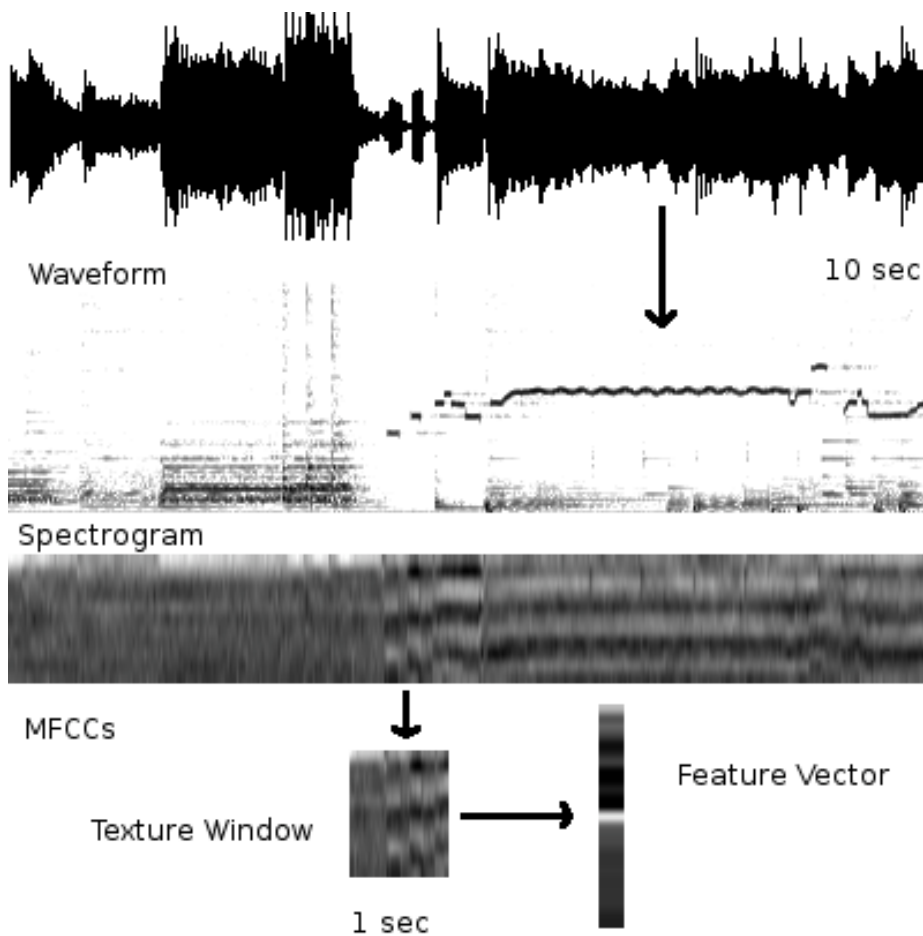


Figure 2: Frequency and time summarization in feature extraction

be used for segmentation and classification. The audio signal is broken into small slices and by performing some form of frequency analysis such as the Discrete Fourier Transform followed by a summarization step a set numbers (the feature vector) is calculated. The feature vector attempts to summarize/capture the content information of that short slice in time. A piece of music can then be represented as a sequence of feature vectors. By detecting abrupt changes in the trajectory of the feature vectors segmentation can be performed and by detecting regions in feature space classification can be performed. Most audio features are extracted in three stages: 1) spectrum calculation, 2) frequency-domain summarization 3) time-domain summarization. In spectrum calculation, a short-time slice (typically around 10 to 40 milliseconds) of waveform samples is transformed to a frequency domain representation. The most common such transformation is the Short Time Fourier Transform (STFT). During each short-time slice the signal is assumed to be approximately stationary and is windowed to reduce the effect of discontinuities at the start and end of the frame. This frequency domain transformation preserves all the information in the signal and therefore the resulting spectrum has high dimensionality. For analysis purposes, it is necessary to find a more succinct description that has significantly lower dimensionality while still retaining the desired content information. Frequency domain summarization converts the high dimensional spectrum (typically 512 or 1024 coefficients) to a smaller set of number features (typically 10-30). A common approach is to use various descriptors of the spectrum shape such as the Spectral Centroid and Bandwidth. Another widely used frequency domain spectrum summarization is Mel-Frequency Cepstral Coefficients (MFCCs) which originated from

speech and speaker recognition. The goal of time domain summarization is to characterize the musical signal at a longer time scale than the short-time analysis slices. Typically this summarization is performed across so called “texture” windows of approximately 2-3 seconds or it can be also performed over the entire piece of music. Figure 2 shows graphically frequency and time summarization. Later in this chapter we show how this process can be represented in *MARSYAS*.

MARSYAS 0.1 mainly supported audio analysis. One of the main motivations behind the redesign/rewrite of version 0.2 was the desire to also support audio synthesis. Even though not immediately obvious, audio synthesis can be very useful in MIR systems and applications. For example while researching an algorithm for drum transcription it is very useful to be able to hear a re-synthesized sound contains only drum sounds. Listening to the result can reveal information that maybe hard to obtain from plots or just numeric results. Similarly synthesizing the result of pitch extraction can be very informative about the nature of pitch errors. Although it is possible to use a variety of different tools for these purposes having them all integrated under one system can be very helpful.

The central challenge in the design of an audio processing framework is the need for very efficient runtime performance while retaining expressivity. Frequently audio researchers are faced with a hard dilemma. They can use interpreted programming environments such as MATLAB that provide a lot of necessary components but are not as efficient as compiled code or write code from scratch which requires a significant investment of time just to build the necessary infrastructure. *MARSYAS* attempts to balance these two

extremes. As will be described later in this chapter a lot of flexibility and functionality is provided at run-time. For example the user can easily create complicated networks of processing objects and control them without recompiling code. However the resulting system is still very efficient as it relies on compiled code. The only time that code recompilation is required is when new processing objects need to be written.

Two other important requirements for a successful audio framework are interoperability and portability. Most researchers utilize a large ecology of different tools to accomplish their task. To be useful a framework must provide ways to communicate with other tools. For example *MARSYAS* provides a variety of tools for communicating with specific programs such as WEKA (for machine learning) and MATLAB (for numerical calculation) as well as general ways for communication with graphical user interfaces. Portability is also very important as operating systems and hardware change all the time. For example initially *MARSYAS* was developed on SGI machines running IRIX which is not supported any more but OS X which didn't exist at the time is supported now.

These requirements make developing of audio processing frameworks challenging. However there are some characteristics of audio that can help the designer. Audio processing has a strong notion of time. There is a constant flow of data through the system. In most cases there is little need for complicated dependencies between processing chunks of data. Therefore for a specific application it is relatively easy to

create efficient code with a fixed memory footprint. With a little bit more work it is possible to generalize the process so that most audio applications can be expressed easily without sacrificing run-time performance.

In the following sections we describe the general architecture of *MARSYAS* and how these requirements are addressed by specific design decisions and concepts. We believe that many of these concerns and their solutions are relevant not only to programmers and users of MARSYAS but developers of any MIR software system.

Architecture

Dataflow programming is based on the idea of expressing computation as a network of processing nodes/components connected by a number arcs/communication channels.

Computer Music is possibly one of the most successful application areas for the dataflow programming paradigm. The origins of this idea can possibly be traced to the physical re-wiring (patching) employed for changing sound characteristics in early modular analog synthesizers.

Expressing audio processing systems as dataflow networks has several advantages. The programmer can provide a declarative specification of what needs to be computed without having to worry about the low level implementation details of how it is computed. The resulting code can be very efficient and have a small memory footprint as data just “flows” through the network without having complicated dependencies. In addition,

dataflow approaches are particularly suited for visual programming. One of the initial motivations for dataflow ideas was the exploitation of parallel hardware and therefore dataflow systems are particularly good for parallel and distributed computation.

The main goal of *MARSYAS* is to provide a general, extensible and flexible framework that enables experimentation with algorithms and provides the fast performance necessary for developing real-time audio analysis and synthesis tools. A variety of existing building blocks that form the basis of many published algorithms are provided as dataflow components that can be composed to form more complicated algorithms (black-box functionality). In addition, it is straightforward to extend the framework with new building blocks (white-box functionality).

In *MARSYAS* terminology the processing nodes of the dataflow network are called *MarSystems* and provide the basic components out of which more complicated networks are constructed. Essentially any audio processing algorithm can be expressed as a large composite *MarSystem* which is assembled by appropriately connected basic *MarSystems*. Some representative *MarSystems* provided are the following:

- Input/Output (Sources and Sinks)
 - SoundFile I/O for .wav, .au, .mp3 and .ogg files
 - Real-time audio I/O using RtAudio []
 - Matlab, Weka, Octave I/O
- Feature Extraction

- Short-Time Fourier Transform (STFT)
- Discrete-Wavelet Transform (DWT)
- Centroid, Rolloff, Flux, Contrast
- Mel-frequency Cepstral Coefficients (MFCC)
- Auditory Filterbanks
- Synthesis
 - Wavetable synthesis
 - FM synthesis
 - Phasevocoder
- Machine Learning
 - Gaussian Mixture Models (GMM)
 - K-Nearest Neighbor
 - Principal Component Analysis
 - K-Means Clustering
 - Support Vector Machine (SVM)

In addition to being able to process data *MarSystems* need additional information that can change their functionality while they are running (processing data). For example a *SoundFileSource* needs the name of the sound file to be opened and a *Gain* can be adjusted while data is flowing through. This is achieved by a separate message passing mechanism. Therefore, similarly to CLAM (Amatriain, 2005), *MARSYAS* makes a clear distinction between data-flow which is synchronous and control flow which is asynchronous. Because *MarSystems* can be assembled hierarchically the control

mechanism utilizes a path notation similar to OSC (Wright and Freed, 1997). For example *Series/playbacknet/Gain/g1/mrs_real/gain* is the control name for accessing the *gain* control of a *Gain MarSystem* named *g1* in a *Series* composite named *playbacknet*. A mechanism for linking top-level controls (with shorter names that act as aliases) to the longer full path control names is provided. For example a single gain control at the top-level can be linked to the gain controls of 20 oscillators in a synthesis instrument. That way, one-to-many mappings can be achieved in a similar way to the use of regular expressions in OSC (Wright and Freed, 1997).

Dataflow in Marsyas is synchronous which means that at every “tick” a specific slice of data is propagated across the entire dataflow network. This eliminates the need for queues between processing nodes and enables the use of shared buffers which improves performance. This is similar to the way Unix pipes are implemented but with audio specific semantics (see section on Implicit Patching).

One of the most useful characteristics of *MarSystems* is that they can be instantiated at run-time. Because they are hierarchically composable that means that any complicated audio computation expressed as a dataflow network can be instantiated at run-time. For example multiple instances of any complicated network can be created as easily as the basic primitive *MarSystems*. This is accomplished by using a combination of the *Prototype* and *Composite* design patterns (Gamma et al., 1995).

Projects

MARSYAS has been used for a variety of projects both in academia and industry. In this section we briefly describe some specific representative examples. The list is by no means exhaustive. An open source framework enables collaboration possibilities. Two research publications that resulted from such collaborations are (Lippens et al, 2004), (Li and Tzanetakis, 2003). *Musicream* is a new music playback interface for streaming, sticking, sorting and recalling musical pieces that uses *MARSYAS* for calculating features and content-based audio similarity (Goto and Goto, 2005). The SndTools software also uses *MARSYAS* under the graphical user interface (Misra et al, 2005).

In industry *MARSYAS* has been used to design and prototype the audio fingerprinting software used by Moodlogic Inc. (<http://moodlogic.com>). After the fingerprinting was designed and evaluation experiments were conducted using *MARSYAS* a new proprietary source code just for the fingerprinting was written for the actual working product. This method of using research free software framework for prototyping and then providing a full proprietary rewrite is one possibility of how academic free software and industry can co-exist. Another possibility is the gender (male/female) voice detection scheme developed for *Teligence Communications Inc* (<http://www.teligence.net/>) using *MARSYAS*. In that case the developed software is part of the free software distribution. However it is based on machine learning and requires training data that is not publicly available and belong to *Teligence Communications Inc*. As the company is mainly concerned in using the tool internally everything work out ok for both parties. In both of these industrial collaborations it would have been possible for the company to just develop the software on their own. However by collaborating and paying the companies

benefit from fast turnaround and experienced knowledge and the free software authors benefit from consulting payments. We hope that these strategies might provide some information about how industrial collaborations with academic free software projects can be established.

Specific Topics

In this section we discuss in more detail some specific topics that we believe are particularly interesting to the designer of audio processing frameworks.

Implicit Patching

The basic idea behind *Implicit Patching* (Bray and Tzanetakis, 2005) is to use object composition rather than explicitly specifying connections between input and output ports in order to construct the dataflow network. For example the following pseudo-code example (Figure 3) illustrates the difference between *Explicit* and *Implicit Patching* in a simple playback network.

```
# EXPLICIT PATCHING

create source, gain, dest

# connect the appropriate in/out ports

connect(source.out1, gain.in1);

connect(gain.out1, dest.in1);

#IMPLICIT PATCHING

create source, gain, dest

#create a composite that is the network

create series(source, gain, dest)
```

Figure 3: Explicit and Implicit Patching

The idea of *Implicit Patching* evolved from the integration of three different ideas that were developed independently in previous versions of *MARSYAS*. These three ideas and how they are integrated are described below. In addition, examples illustrating the expressive power of *Implicit Patching* are presented.

The first idea originated from the desire not to be constrained to fixed buffer sizes and to have proper semantics for spectral data. The majority of existing audio processing environments require that all processing objects in a flow network/visual patch, process fixed buffers of audio samples (typical numbers are 64 and 128 samples). Having fixed buffers simplifies memory management and patching as all connections are treated the same way. However, some applications like audio feature extraction require a variety of different buffer sizes to flow through the network (for example feature vectors typically have much lower dimensionality than audio data). Even though it is possible to have dynamic buffer sizes in *Explicit Patching* it is complex to implement and frequently requires a lot of work from the programmer to appropriately set the connections. In addition, these fixed-sized buffers are reused for holding spectral data and it is up to the programmer to correctly connect the spectral data to objects that process such data. The result is that the exact details of the Short-Time Fourier Transform are encapsulated as a black box and the programmer has little control over the process. Our proposed solution to these two problems is to extend the semantics of the data that is processed. In *MARSYAS*, processing objects (*MarSystems*) operate on chunks of data called *Slices*. *Slices* are matrices of floating point numbers characterized by three parameters: number of samples (things that are “measured” at different instances in time), number of observations (things that are “measured” at the same time instance) and sampling rate. This approach is similar to the Sound Description Interchange Format (SDIF) (Schwarz and Wright, 1997).

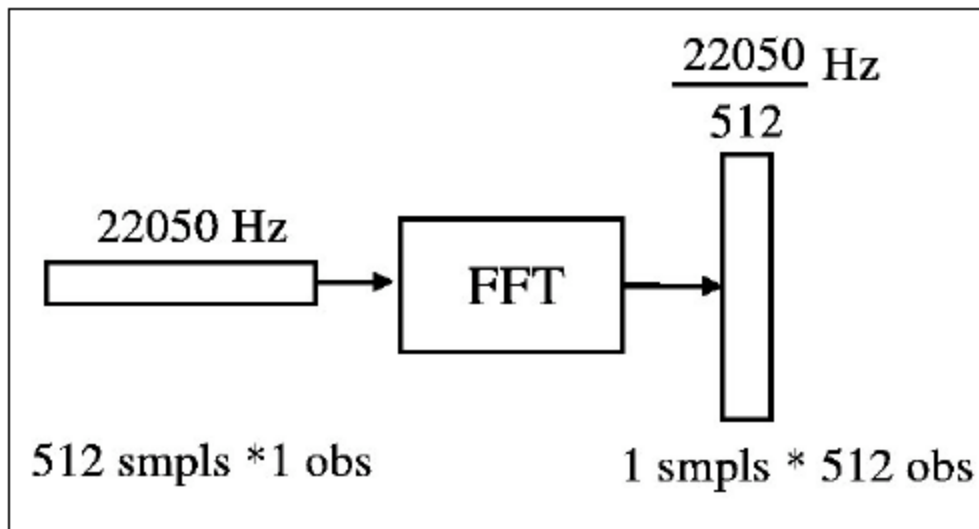


Figure 4: *MarSystem* and corresponding slices for spectral processing

Figure 4 shows a *MarSystem* for spectral processing that converts an incoming audio buffer of 512 samples of 1 observation at a sampling rate of 22050 Hz to 1 sample of 512 observations (the FFT bins) at a lower sampling rate of $22050/512$ Hz. By propagating information about the sampling rate and number of observations through the dataflow network, the use of *Slices* provides more correct and flexible semantics for spectral processing and feature extraction. *MarSystems* are designed so that they can handle *Slices* with arbitrary dimensions with one important constraint: they need to be able to calculate their output *Slice* parameters from their input *Slice* parameters. For example it is possible to change the input number of samples to the *MarSystem* to shown in Figure 4 to 1024 and the *MarSystem* will automatically determine that the number of observations of the output *Slice* is also 1024.

The second major idea behind *Implicit Patching* is the use of the Composite design pattern (Gamma et al., 1995) as a mechanism for constructing dataflow networks. The extended semantics of Slices require careful manipulation of buffer sizes especially if run-time changes are desired. The most important composite is Series which connects a list of *MarSystems* in series so that the output of the first one becomes the input to the second etc. (similarly to UNIX pipes). The pseudo-code in Figure 3 uses a Series composite. Initially composites were used as programming shortcut. However, gradually we discovered that they offer many advantages and we decided to make them the main mechanism for constructing complicated *MarSystems* out of simpler ones. Their advantages include hierarchical encapsulation, automatic dynamic handling of all internal buffers, and run-time instantiation. More specifically, any dataflow network, no matter how complicated, is represented as a single *MarSystem* that is hierarchically composed of simpler *MarSystems*. Multiple instance of any *MarSystem* can be instantiated at run-time and all internal patching and memory handling is encapsulated.

The third idea was the unification of *Sources* and *Sinks* as regular *MarSystems* that have both input and output. *Sources* are processing objects that have only output and *Sinks* only have input. In order to be able to use them as any *MarSystem* we extend them in the following way: *Sources* mix their output with their input and *Sinks* propagate their input to their output while at the same time playing/writing their input as a side-effect. This way, for example, one can connect a *SoundFileSink* to an *AudioSink* in series and the data will be written both to a sound file and played using the audio device. Basically this way both *Sources* and *Sinks* can be used anywhere inside a network.

Implicit Patching is made feasible by the integration of these three ideas. In this approach, each *MarSystem* has only one input port and one output port and consumes/produces only one token. However because of the extended semantics of Slices one can essentially have multiple input/output ports (as observations) and consume/produce multiple tokens (as samples). This enables non-trivial *Composites* such as *Fanout* to be created. The expressive power of composition is increased and a large variety of complex dataflow networks can be expressed only using object composition and therefore no *Explicit Patching*. Another side benefit of *Implicit Patching* is that it enforces the creation of trees and therefore avoids problems with cycles in the dataflow graph.

Figure 5 shows how a *Series Composite* consisting of a *SoundFileSource src*, *Gain g*, and *AudioSink dest* can be assembled in C++. At every iteration of the loop the audio rate is incremented starting from 1 sample (similar to Chuck or STK) until the block size of 1000 samples is reached. All the intermediate shared buffers between the *MarSystems* are adjusted automatically and the sound plays without interruption. Even though this example might seem artificial the need for dynamically adjusting window sizes occurs frequently in audio analysis for example in pitch-synchronous overlap-add (PSOLA).

```
MarSystem* net = mng.create("Series", "net");

net->addMarSystem(src);

net->addMarSystem(g);

net->addMarSystem(dest);

for (int i=1; i<1000; i++)
{
    net->updctrl("natural/inSamples", i);

    net->tick();
}
```

Figure 5: Dynamic adjusting of analysis window size

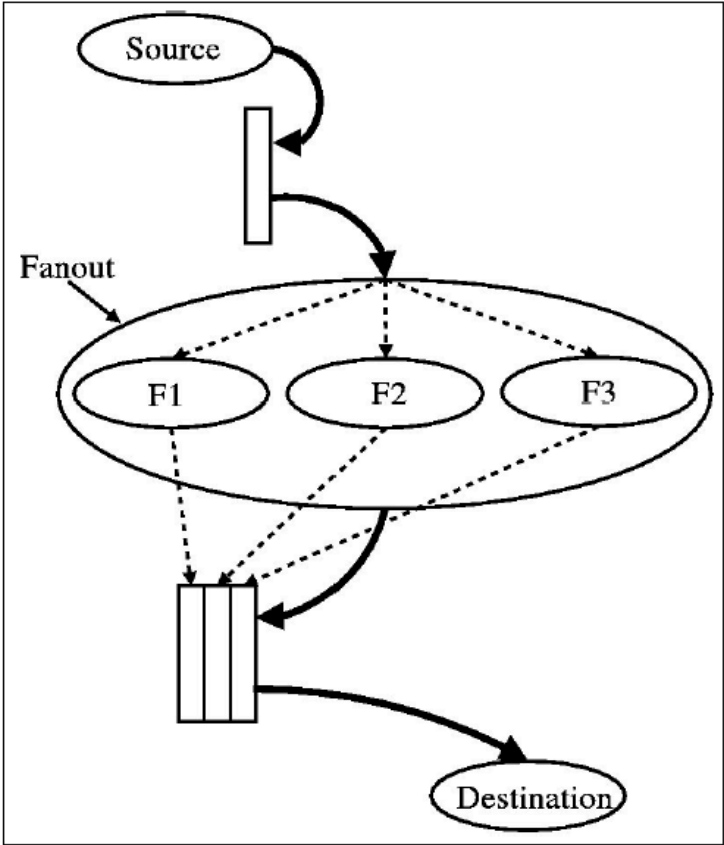


Figure 6: Fanout Composite

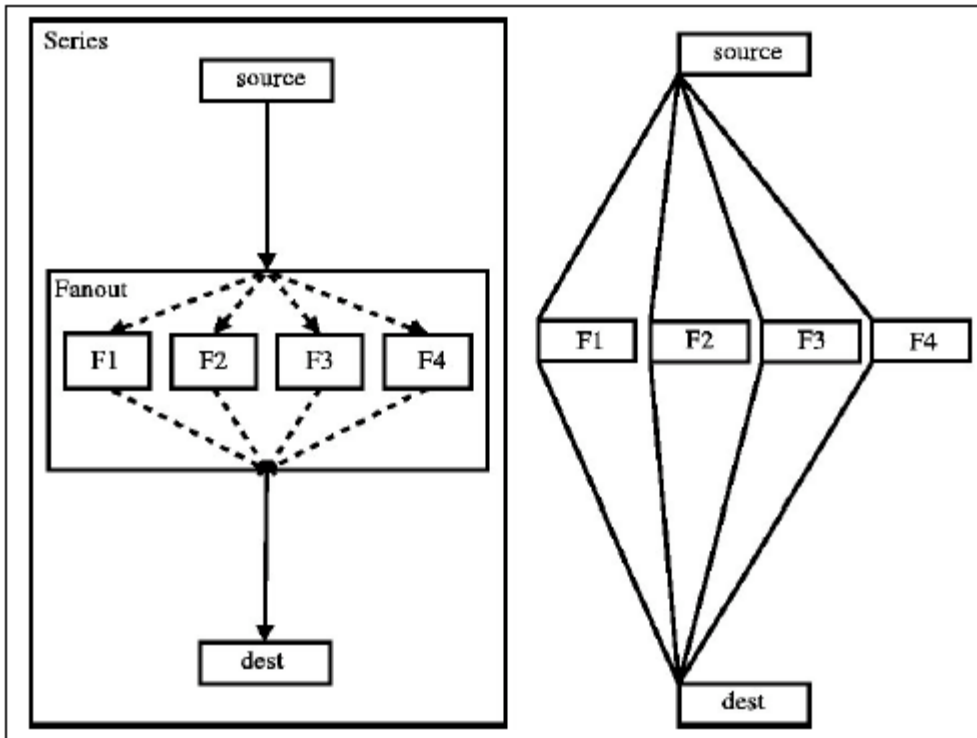


Figure 7: Fanout Composite with Implicit Patching (left) and Explicit Patching (right)

To illustrate this approach, consider the *Fanout Composite* which takes as input a slice and is build from a list of *MarSystems*. The input *Slice* is shared as input to all each internal *MarSystem* and their outputs are stacked as observations in the output *Slice* of the *Fanout*. For example a filterbank can be easily implemented as a *Fanout* where each filter is an internal component *MarSystem*. The filterbank will take as input a slice of N samples by 1 observations and write to an output slice of N samples by M observations, where M is the number of filters. Because the inner loops of *MarSystems* iterate over both samples and observations if we connect the filterbank with a *Normalize MarSystem* each row of samples corresponding to a particular observation (each channel of the

filterbank) will be normalized accordingly. This can be very handy in large filterbanks as the part of the network after the filterbank doesn't need to know how many filter outputs are produced. This information is taken implicitly from the number of observations.

Figure 6 shows graphically how *Slices* are used in *Fanout*. The dotted lines show the patching that is done implicitly by the *Fanout*. The black arrows show the main flow created implicitly by a *Series Composite*. In contrast, in environments with explicit patching such as Max/MSP each connection between the filters of the filterbank and the input would have to be created by the user. Figure 7 shows the difference between Implicit Patching (left) where the dotted lines are created automatically from the semantics of Composites, and Explicit Patching (right) where each connection must be created separately. Even though environments such as Max/MSP or PD provide sub-patching, the burden of internal patching is still on the user.

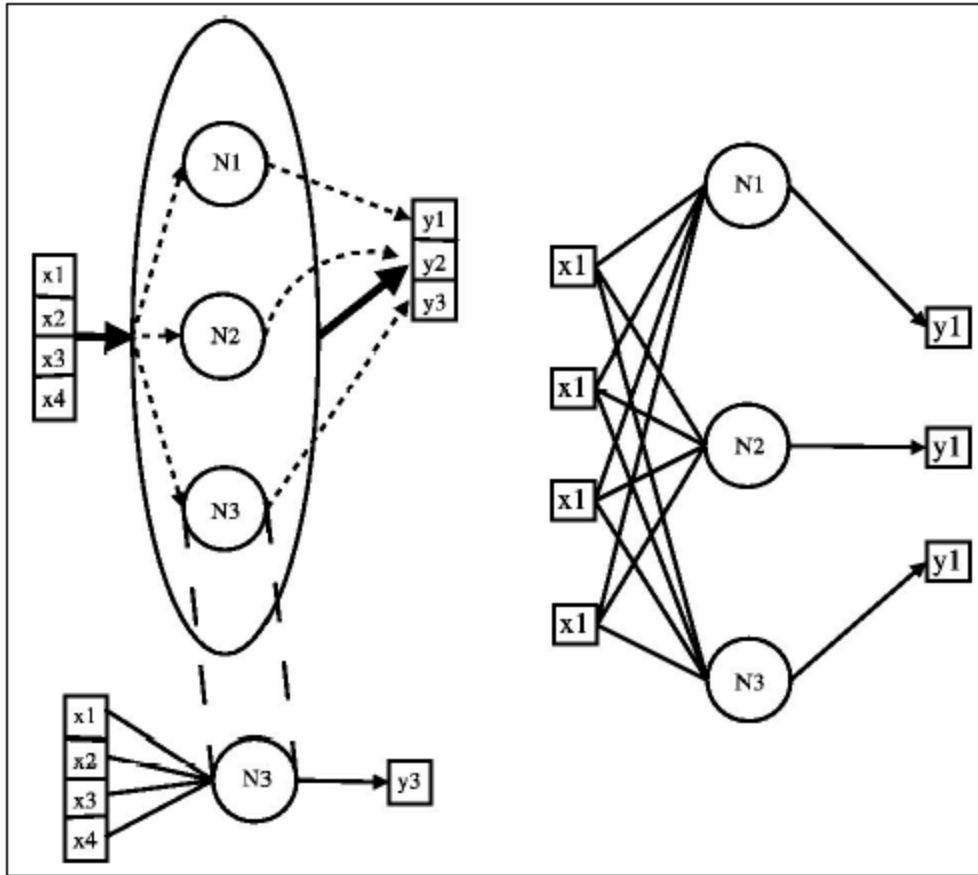


Figure 8: Artificial Neural Network layer using the Fanout Composite

We conclude this section with a non-trivial example illustrating the expressive power of *Implicit Patching*. Figure 8 shows how a layer of nodes in an Artificial Neural Network (ANN) can be expressed using a *Fanout*. The input to the layer (the output of the previous layer) consists of 4 number x_1, x_2, x_3, x_4 . These 4 numbers (observations) based on the *Fanout* semantics become the input to each individual neuron (N_i) of the layer. Each neuron forms a weighted sum (with weights specific to each neuron) of the input, applies a sigmoid function to the sum and outputs a single output. The outputs using the *Fanout* semantics are stacked as observations y_1, y_2, y_3 (one for each neuron) ready for

processing for the next layer. Figure 8 illustrates this process graphically (left side) and contrasts it with Explicit Patching (right side). In *MARSYAS*, creating an ANN using an *AnnNode MarSystem* is simply a *Series of Fanout of AnnNodes*. More specifically $\text{seriesNet}(\text{fanoutLayer1}, \text{fanoutLayer2}, \dots, \text{fanoutLayerM})$ where $\text{fanoutLayer1}(\text{annNode11}, \text{annNode12}, \dots, \text{annNode1N})$. All the connections are created implicitly.

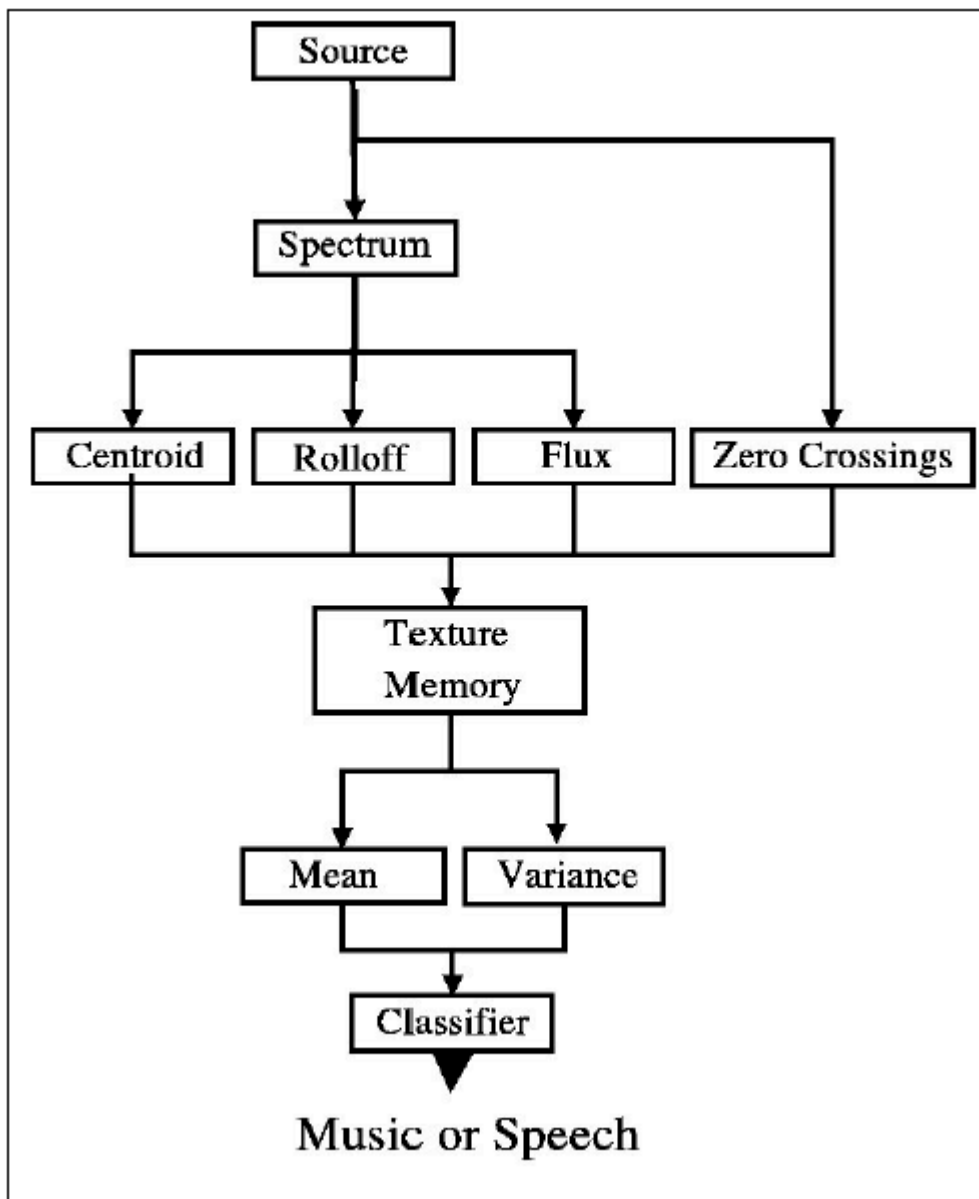


Figure 9: Music/Speech classification as a dataflow network in *MARSYAS*

Figure 9 shows a dataflow network for extracting audio feature for real-time music/speech classification. *Series* connections are top to bottom and *Fanout* connections are shown by horizontal forking. For example the output of the *Spectrum* calculation is

used as input to the *Centroid*, *Rolloff* and *Flux MarSystems*. The input to texture memory is 4 observations (the features) by 1 sample and the output is 4 observations by 40 samples consisting of the last 40 feature vectors (approximately corresponding to 1 second). Means and variances of the feature vectors over the texture window are calculated and the input to the classifier is a 8-dimensional feature vector (4 means and 4 variances). The entire network can be created at run-time without requiring any code recompilation. The complete feature extraction front-end described in [Tzanetakis and Cook, 2002] has been implemented as a dataflow network in *MARSYAS* in a similar fashion.

Flexible Scheduling

Scheduling is central to any computer music system. A scheduling request consists of an event and a time. The scheduler keeps track of pending requests. Computer music schedulers use times which are typically references to a single clock that corresponds to real (physical) time. In addition to actual time, *MARSYAS* supports the notion of virtual time which is based directly on the data. This is especially important in non-realtime applications. In addition it is often convenient to have a time reference or references that do not correspond to real time. For example consider scheduling events in beats that are defined in relation to tapping a MIDI keyboard to extract beat-synchronous audio

features. In this section we show how multiple notions of time and events are supported through object orientation in *MARSYAS*.

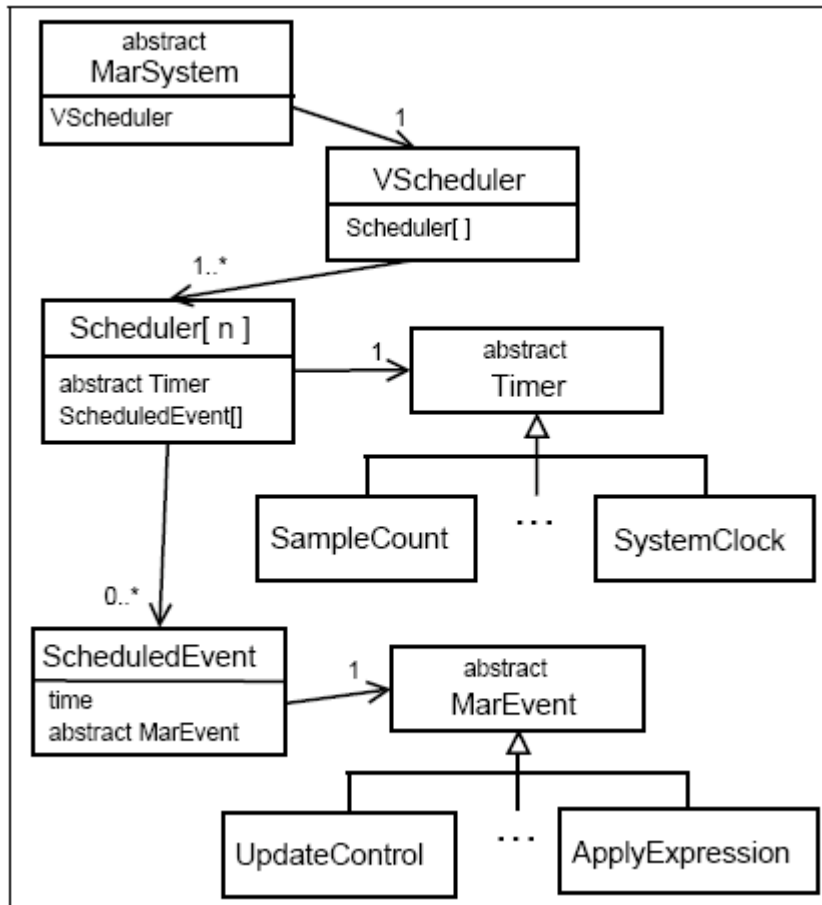


Figure 10: UML class diagram of the *MARSYAS* scheduler architecture

Each *MarSystem* has its own *Virtual Scheduler* that manages an arbitrary number of *Event Schedulers*. Each *Event Scheduler* contains its own *Timer* that controls the rate at which time passes and events are dispatched. *Schedulers* themselves do not keep track of

time but leave this task solely to the *Timer*. Structured this way events may be scheduled to any number of different *Timers*.

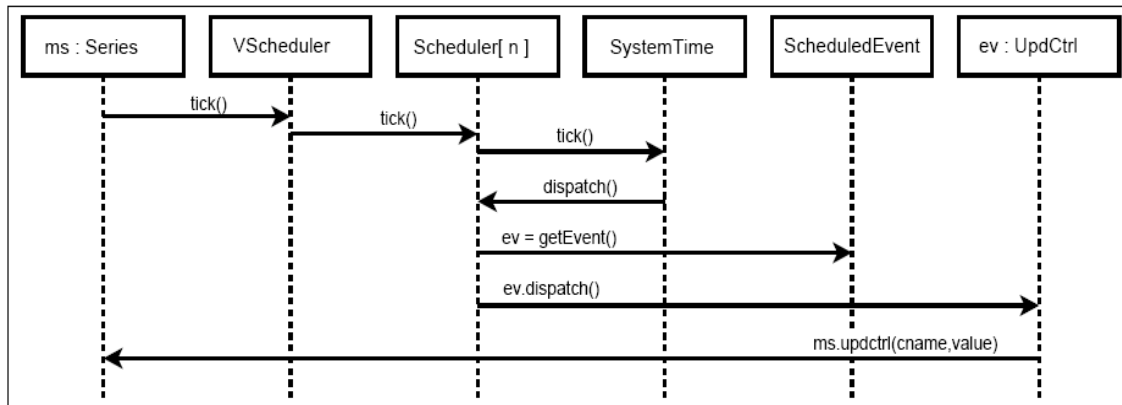


Figure 11: UML sequence diagram of event dispatch

A *Timer* in *MARSYAS* is any object that can read from a time source and triggers some action on each tick. A *Timer* must also provide some way to specify units of time in its time base. The one restriction in a time source is that time must always advance. *Timers* are definable by the user provided they support the *AbstractTimer* interface. The interface requires specification of the following: a method for determining the interval of time since the last reading, a method for comparing time intervals for that particular *Timer*, a trigger method which calls the Scheduler dispatch, and a method for converting time representations to the specific notion of time used by the *Timer*. This generalization of

Timers allows for many different possibilities in controlling event scheduling. Linear and non-linear advancing are both possible.

Events are user definable actions that are “frozen” until dispatched. They are distinct from the normal flow of audio data. There are no restrictions on the types of *Events* that can be defined. Perhaps the most common event is updating a *MarSystem* control (for example creating a new file with feature every 1 minute). Another example is the “wire” event which updates the value of a control based on the value of another control (similar to a control wire in Max/MSP, PD). *Events* must supply a *dispatch()* method that the *Scheduler* that the Scheduler can call at the appropriate time. *Events* may take place immediately or some time in the future. The time at which an *Event* happens may be specified by the user and depends on the Timer that the requested time is with respect to (for example the user might specify that an *Event* should happen after 4 beats using a tempo-based *Timer* or 3 seconds using a time-based *Timer*). The *Event* is then sent to the scheduler and removed when its time interval has passed. The *Scheduler* then calls the *dispatch()* method on the event.

Figure 10 shows a UML class diagram of the scheduling architecture. Notice how multiple, user-defined *Timers* and *Events* can be supported by abstraction and are decoupled from the *Scheduler*. The UML sequence diagram of Figure 11 shows the order of method calls between objects for performing a control update (a specific kind of *Event*).

Marsyas Scripting Language

```
Series net is [  
    Fanout mix is [ SineSource src1,  
                   SineSource src2 ],  
    Sum sum,  
    AudioSink dac  
]  
  
do [ (Math.rand() * 10000)+100 => net/mix/src/frequency  
    ] every 2beats using TempoTimer  
  
do [ (net/mix/src2/frequency + 400) % 10000  
    => net/mix/src2/frequency  
    ] every 0.5s using SystemTimer  
  
run net run
```

Figure 12: *MSL* code example

MSL is a scripting language for building and controlling *MarSystem* networks at runtime. It is more accessible and readable than writing raw C++ code. Essentially the interpreter translates MSL code into the corresponding C++ *Marsyas* code and then executes it. Lexical analysis (or scanning) is performed using the Flex scanner generator, and parsing performed using the Bison parser generator ². The output of the parsing stage from Bison is an abstract syntax tree representation of the MSL script, which is then traversed to generate and execute the equivalent *Marsyas* C++ code. The “do” construct allows multiple events to be scheduled at particular times based on a Timer. Figure XXX shows a more complicated example with two voice polyphony using a Fanout composite. Two sine oscillators are controlled by separate timers one based on SystemTimer and the other based on TempoTimer which is based on MIDI input

Distributed audio processing

One of the most important challenges facing MIR of audio signals is scaling algorithms to large collections. Typically, analysis of audio signals utilizes sophisticated signal processing and machine learning techniques that require significant computational

resources. Processing time is a major bottleneck. For example, the number of pieces utilized in the majority of existing work in audio MIR is at most a few thousand files. Computing audio features over thousands of files can sometimes take days of processing. The dataflow architecture of MARSYAS facilitates partitioning of audio computations over multiple computers. Using a declarative dataflow specification approach the programmer can distribute audio analysis algorithms with minimum effort. In contrast, distributing traditional sequential programs places a significant burden to the programmer who has to decide which parts of the code can be parallelized and deal with load distribution, scheduling, synchronization and communication.

There are two standard data communication protocols used on the Internet: transmission control protocol (TCP), and user datagram protocol (UDP). TCP provides reliability mechanisms to ensure that all packets are received exactly in the order they are sent; on the other hand, UDP provides no such mechanisms but is significantly faster due to less overhead. UDP is therefore the protocol of choice for real-time streaming applications where data is time critical.

MARSYAS supports both the UDP and TCP protocols. IN order to send data to another machine, a *NetworkSink MarSystem* is simply inserted somewhere in the “flow” of a *Composite MarSystem*. In order to receive data a *NetworkSource MarSystem* is inserted. Control flow and data flow are managed separately so that controls can be changed from the sender and propagate through the system. The idea is that the user can operate several

“worker” machines and the view of the distributed system is abstracted as one large *Composite MarSystem*.

For the experiments described in this section a feature vector consisting the means and variances of smoothed Mel-Frequency Cepstral Coefficients (MFCC) as well as STFT-based features such as spectral centroid and rolloff was used. The data consists of 30-second audio clips and a single 35-dimensional feature vector is calculated for each clip. The actual audio waveform samples are transmitted over the network corresponding to a scenario where all the audio files reside on a single machine but multiple processing machines are available (for example the computer of a lab during nighttime). The experimentation was done on a 100Base-T Ethernet local area network of Apple G5 computers.

In the described experiments a dispatcher node (computer) sends separate audio clips from an audio collection to each worker node in the network. The job of a worker node is to simply calculate features for each clip it receives and then send the results to a collector process (possibly running on the same machine as the dispatcher) that gathers the results. The audio collection was partitioned into sub-collections which were sent to each worker. All the audio clips are stored on the dispatcher. We found that the optimal number of worker nodes in this model was three, after which there was no time benefit of using extra machines. In fact, it was costly to add any more than five worker nodes due to the network capacity of the dispatcher collector. The use of multiple dispatchers in

hierarchical fashion can improve results. More details can be found in [Bray and Tzanetakis, 2005b].

Table 1 shows results of using the collection dispatcher model, using up to five worker nodes and audio collections of up to 10,000 files. The format is *hours:minutes:seconds*.

TABLE 1: Parallelization results for Collection Dispatcher

	10	100	1000	10000
Local	00:05	00:58	09:39	1:36:49
1W	00:07	01:10	11:48	1:58:49
2W	00:03	00:38	06:01	1:10:46
3W	00:04	00:34	05:49	59:46
4W	00:03	00:34	05:52	1:04:56
5W	00:04	00:36	05:54	1:08:36

The problem with the collection dispatcher approach is that some nodes may complete processing the features of their respective subcollection before others and have to sit idle. Thus the time it takes to process the entire collection is dependent on the slowest node in the system. In order to alleviate this problem and make sure of idle nodes, an adaptive approach is used where the dispatcher sends data as necessary to each worker. That way, each node in the system is working until the dispatcher has finished processing the files in the collection. Table 2 shows the increase in performance based on this approach.

TABLE 2: Parellelization results for Adaptive Dispatcher

	10	100	1000	10000
Local	00:05	00:58	09:39	1:36:49
1W	00:07	01:10	11:48	1:58:49
2W	00:04	00:40	06:21	1:02
3W	00:03	00:33	05:33	57:10
4W	00:03	00:31	05:24	54:20
5W	00:03	00:31	05:25	54:15

Typically feature extraction tests run on audio collections of around 10,000 files. Based on our results we expect a linear trend as collection size increases. To test that hypothesis a large-scale test using the data-partitioning model (two dispatchers with half the audio data each) with the adaptive dispatcher was conducted on 100,000 files. As expected, it took approximately ten times the amount of time to complete the 100,000 clip test as it took to complete the 10,000 file test (5:00:44). Experimental results show that using 5 computers we can perform audio feature extraction for 100,000 30-second clips in 5 hours.

Conclusions and Future Work

In this chapter we described MARSYAS a free software framework for audio application with specific emphasis on MIR. We showed how *MARSYAS* addresses some of the requirements and challenges facing the designer of audio processing frameworks. It is our hope that the ideas and concepts presented in this chapter can be applied to other MIR software frameworks and tools. As a general conclusion dataflow architectures can help express easily complicated processing structures while retaining efficiency and being easy to parallelize. Finally the development of free software in an academic setting is not only possible but has many benefits such as increase in visibility, collaboration opportunities, communication and even monetary rewards.

MARSYAS is an on-going project and therefore there are always many directions of future work. In addition to obvious directions such as expanding the number of building blocks provided by the framework and improving reliability and performance there a number of more large scale initiative. A large effort is underway to build a visual patching environment and a library for creating widgets and audio processing graphical user interfaces (GUIs). Another initiative is to port *MARSYAS* to the Java programming language (the previous version 0.1 was partly ported). A more radical complete redesign and implementation is underway in the functional programming language OCAML.

Audio programming in general provides a rich fertile area for ideas in Software Engineering as it combines many interesting areas such as digital signal processing, machine learning, efficient numerical processing, and interactivity.

Acknowledgments

There are many individuals that have helped in one way or another with the design and development of Marsyas. These include in no particular order: Luis Gustavo Martins, Jennifer Murdoch, Start Bray, Neil Burroughs, Adam Tindale, Adam Parkin, Ajay Kapur, Manj Benning, Douglas Turnbull, George Tourtellot, Taras Glek, Ari Lazier, Andrey Ermolinskyi, Carlos Castillo, Perry Cook, and Malcolm Slaney.

References

International Conferences on Music Information Retrieval, <http://www.ismir.net>

Amatriain, X. (2005), "An Object-Oriented Metamodel for Digital Signal Processing with a Focus on Audio and Music," PhD. Dissertation, Univ. of Pompeu Fabra, Spain.

Bray, S. and Tzanetakis, G. (2005a), "Implicit Patching for Dataflow-Based Audio Analysis and Synthesis," in *Proc. Int. Computer Music Conf. (ICMC)*.

Bray, S. and Tzanetakis, G. (2005b), “Distributed Audio Feature Extraction for Music,” in *Proc. Int. Conf. on Music Information Retrieval (ISMIR)*.

Burroughs, N., Parkin, A., and Tzanetakis, G. (2006), “Flexible Scheduling for DataFlow Audio Processing,” in *Proc. Int. Computer Music Conf. (ICMC)*, New Orleans, USA.

Cook, P., and Scavone, G. (1999) “The Synthesis Toolkit (STK) version 2.1,” in *Proc. Int. Computer Music Conf (ICMC)*, Beijing, China.

Dannenberg, R., and Brandt, E. (1996) “A flexible real-time software synthesis system,” in *Proc. Int. Computer Music Conf. (ICMC)*, pp. 270-273.

Downie, S.J. and Futrelle, J. (2005) “Terascale music mining,” in *Proc. ACM/IEEE Super Computing Conference*, 2005.

Futrelle, J. and Downie, S. J. (2002), “Interdisciplinary Communities and Research Issues in Music Information Retrieval,” in *Proc. Int. Conf. on Music Information Retrieval (ISMIR)*, Paris, France, pp. 215-221.

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

Goto and Goto (2005) "Musicream: New Music Playback Interface for Streaming, Sticking, Sorting and Recalling Musical Pieces" In *Proc. Int. Conf. on Music Information Retrieval (ISMIR)*, London, UK.

Lippens, S. et al. (2004) "A Comparison of Human and Automatic Musical Genre Classification" In *Proc. IEEE Int. Conf. on Audio, Speech and Signal Processing* Montreal, Canada

Li, T. and Tzanetakis, G. (2003) "Factors in Automatic Musical Genre Classification of Audio Signals" In *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)* New Paltz, NY.

Manulescu, D.A. (1997), "A dataflow pattern language," in *Proc. Pattern Languages of Programming*, Monticello, Illinois.

Misra, A., Wang, G. and Cook, P. (2005), "SndTools: Real-Time Audio DSP and 3D Visualization," in *Proc. Int. Computer Music Conference (ICMC)*, Barcelona, Spain.

Puckette, M., (2002), "Max at seventeen," *Computer Music Journal*, vol. 26(4).

Schwarz, D. and Wright, M. (2000), "Extensions and Applications of the SDIF sound description interchange format," in *Proc. Int. Computer Music Conf (ICMC)*.

Scheirer, E. and Slaney, M. (1997), "Construction and evaluation of a robust multi-feature music/speech discriminator" IEEE Int. Conf. on Audio, Speech and Signal Processing (ICASSP).

Tzanetakis, G. and Cook, P. (2000), "MARSYAS: A Framework for Audio Analysis" *Organized Sound, Cambridge University Press* 4(3).

Tzanetakis, G. and Cook, P. (2002), "Musical Genre Classification of Audio Signals," *IEEE Trans. on Speech and Audio Processing*, vol 10(5).

Wang, G., and Cook, P. (2003), "Chuck: A concurrent, on-the-fly audio programming language," in *Proc. Int. Computer Music Conference. (ICMC)*, Singapore.

Wright, M. and Freed, A. (1997), "Open Sound Control: A new protocol for communicating with sound synthesizers," in *Proc. Int. Computer Music Conf. (ICMC)*, Thessaloniki, Greece.

Zicarelli, D. (2002), "How I learned to love a program that does nothing," *Computer Music Journal*, vol 26(4), pp. 44-51.