IMPLICIT PATCHING FOR DATAFLOW-BASED AUDIO ANALYSIS AND SYNTHESIS

Stuart Bray and *George Tzanetakis
University of Victoria
Computer Science Department *(Also in Music)

ABSTRACT

Programming software for audio analysis and synthesis is challenging. Dataflow-based approaches provide a declarative specification of computation and result in efficient code. Most practitioners of computer music are familiar with some form of dataflow programming where audio applications are constructed by connecting components with "wires" that carry data. Examples include networks of unit generators in Music-V style languages and visual patches in Max/Msp or PD. Even though existing dataflow-based audio systems offer a concise conceptual model of signal computation, this model does have limitations. In many cases, these limitations are a consequence of the programmer having to explicitly specify connections between components. Two such limitations are the difficulty of handling spectral data and the need for fixed-size buffers between components. In this paper we introduce *Implicit Patching* (IP), a dataflow-based approach to audio analysis and synthesis that attempts to address these limitations. By extending dataflow semantics a large number of connections are automatically created and buffer sizes can be changed dynamically. We describe *Marsyas-0.2*, a software framework based on IP, and comment on the strengths and limitations of the proposed approach.

1. INTRODUCTION

There is a plethora of programming languages, frameworks and environments for the analysis and synthesis of audio signals. The processing of audio signals requires extensive numerical calculations over large amounts of data especially when real-time performance is desired. Therefore efficiency has always been a major concern in the design of audio analysis and synthesis systems. Dataflow programming is based on the idea of expressing computation as a network of processing nodes/components connected by a number of communication channels/arcs. 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. Examples from Computer Music include the unit generators in the Music *N* family of languages and visual programming environments such as Max/Msp and Pure data.

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. The resulting code can be very efficient and have low memory requirements as data just "flows" through the network without having complicated dependencies. In addition, dataflow approaches are particularly suited for visual programming. Despite these advantages, dataflow programming has not managed to replace existing imperative, object-oriented and functional languages. Some of the criticisms include: the difficulty of expressing complicated control information, the restrictions on using global state information, and the difficulty of expressing iteration and complicated data structures.

Computer music has been one of the most successful cases of dataflow applications. However, existing audio processing dataflow frameworks have difficulty handling spectral and filterbank data in an conceptually clear manner. Another restriction is the use of fixed buffer sizes/audio rates. Both of these limitations can be traced to the restricted semantics of patching as well as the need to explicitly specify connections. *Implicit Patching* the technique described in this paper is an attempt to overcome these problems while maintaining the advantages of dataflow computation. *Marsyas-0.2* is a software framework for audio analysis and synthesis that is structured around the idea of IP. In order to illustrate the concept of IP we provide specific examples from audio analysis, synthesis.

2. RELATED WORK

Motivated by criticisms of the classical von Neumann hardware architecture, dataflow architectures for hardware were proposed as an alternative in the 70s-80s. During the same period a number of textual dataflow languages such as Lucid [1] were proposed. During 90s dataflow concepts resurfaced in visual programming languages for specific domains. Commercial examples include Labview [1] and SimuLink [2]. A recent comprehensive review of the history of dataflow programming languages is [2]. Another recent trend has been to view dataflow computation as a software engineering methodology for building systems using existing programming languages [3].

[1] `http://www.ni.com/labview/`
[2] `http://www.mathworks.com/products/simulink/`

It is interesting to note that the use of dataflow ideas in Computer Music follows a similar trajectory. Initial experimentation started in the 1960-1970s with modular analog synthesizers that could be programmed by physically "patching" wires. Textual dataflow programming languages in Computer Music are exemplified by the the Music *N* family legacy whose most popular member today is Csound [4]. Today the use of visual dataflow programming environments such as Max/MSP and Pure Data (PD) is pervasive in the computer music community [5]. An object-oriented metamodel for digital signal processing that abstracts many of the dataflow ideas used in audio and music processing is presented in [6]. The use of object composition advocated in this paper has similar advantages to the use of expressions in [7].

Marsyas 0.2, the software framework for audio analysis and synthesis described in this paper, evolved from *Marsyas 0.1* [8] a framework that focused mostly on audio analysis and Music Information Retrieval. One of motivating factors for the rewrite of the code and architecture was the desire to add audio synthesis capabilities influenced by the design of the Synthesis Toolkit [9]. Other influences include the powerful but complicated flow architecture of CLAM [6], the interesting patching model of Chuck [10] and ideas from Aura [11]. The matrix model used in *Implicit Patching* was influenced by the design of SDIF [12] and the control naming scheme is inspired by the Open Sound Control (OSC) format [13]. *Implicit patching*, the technique described in this paper is illustrated with examples from audio analysis and synthesis. The phasevocoder is a computationally intensive audio synthesis algorithm. As an example of audio the feature extraction front-end described in [14] is used.

3. MARSYAS-0.2 ARCHITECTURE

Marsyas-0.2 [3] is a software framework, written in C++, for rapid prototyping and experimentation with audio analysis and synthesis with specific emphasis on processing music signals. The main goal is to provide a general, extensible and flexible framework that allows easy 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). The goal of this section is not to provide an extensive overview of the system architecture but provide the necessary context to understand the ideas described in the paper.

In *Marsyas* terminology the processing nodes of the dataflow network are called *MarSystems* and provide the basic building blocks out of which more complicated systems are built. As will be shown in the next section essentially any audio processing can be expressed as a large composite *MarSystem* which is assembled by appropriately connected basic *MarSystems*. Some representative *MarSystems* provide in *Marsyas-0.2* are the following:

In addition to being able to process data, in order for *MarSystems* to be useful, they need additional information. For example a *SoundFileSource* needs to the name of the soundfile to be opened and a *Gain* could be adjusted while data is flowing through. This is accomplished by a separate message passing mechanism. Therefore, similarly to CLAM [6], *Marsyas-0.2* 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 [13]. For example *Series/playbacknet/Gain/g1/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 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 [13].

Dataflow in *Marsyas-0.2* is synchronous which means that at every "tick" a data slice is propagated across the entire dataflow network. This eliminates the need for queues between nodes and enables the use of shared buffers which improves performance. *MarSystems* are hierarchically composable and can be instantiated at run-time. Currently there are three ways to build audio applications in *Marsyas-0.2*. The first is writing directly C++ code and compiling an executable. The second is a simple scripting language for building the dataflow network, setting appropriately the controls and moving sound through the network. The third is to use a visual patch builder.

4. IMPLICIT PATCHING

The basic idea behind *Implicit Patching* is to use object composition rather than explicitely specifying connections between input and output ports in order to construct the dataflow network. For example the following pseudo-code examples illustrates the difference of *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 essentially the network
create series(source, gain, dest);
```

EXAMPLE 2

[3] `http://marsyas.sourceforge.net`

Implicit Patching evolved from the integration of three different ideas. The first idea originated from the desire for dynamic buffer sizes and proper semantics for spectral data. The majority of existing audio environments require that all processing nodes in a patch process fixed size buffers (typical 64 or 128 samples). This 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). Dynamic buffers in explicit patching systems are complex to implement and not transparent to the programmer. In addition, audiod buffers are reused for holding spectral data and it is up to the programmer to correctly interpret them. Therefore 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-0.2, processing objects (*MarSystems*) process chunks of data called *Slices*. *Slices* are matrices of floating point numbers characterized by three parameters: number of samples (things that "happen" at different instances in time), number of observations (things that "happen" at the same time) and sampling rate. This approach is similar to the matrix approach used in the Sound Description Interchange Format (SDIF) [12]. For example a *MarSystem* for spectral processing that converts an incoming audio buffer of 512 samples of 1 observation at a sampling rate of 22050 to 1 sample of 512 observations (the FFT bins) at a lower sampling rate of 22050/512. By propagating information about the sampling rate and the number of observations through the dataflow network, the use of *Slices* provides more correct and flexible semantics for dynamic buffers and spectral processing.

The second major idea behind *Implict Patching* is the use of *Composite* design pattern [15] 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 first composite used was *Series* which connects a list of *MarSystems* in series so that the output of the first one becomes the input to the second etc (similar to Unix pipe mechanism). The pseudo-code Example 2 above uses a *Series* composite. Initially composites were used a programming shortcuts. However, gradually we discovered that they offer many advantages and we decide to made 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* hierarchically composed of other simpler *MarSystems*, multiple instances of any *MarSystems* can be instantiated at run-time and all internal patching and memory handling is encapsulated.

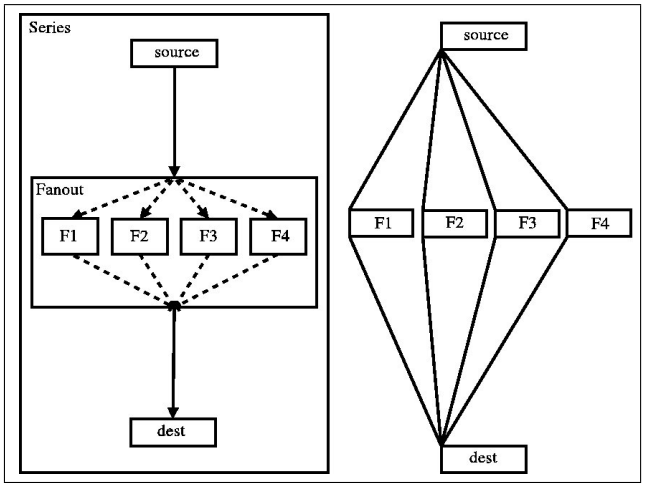

Figure 1. Comparison of Implicit Patching (left) and Explicit Patching (right)

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. This way, for example, one can connect a *SoundFileSink* to a *AudioSink* and the data will be written both to a sound file and played using the audio device. Basically in both *Sources* and *Sinks* data gets injected into the network as a side effect but they 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 a single input port and one output port process a single 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. That way, 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*.

To illustrate this approach, consider the *Fanout* composite which takes as input a slice and is built from a list of *MarSystems*. The input slice is then used as input to 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 a internal component *MarSystem*. The filterbank *MarSystem* 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 innter loops of *MarSystems* iterate over both samples and observations if we connect the filterbank with, for example, a *Normalize MarSystem* each row of samples corresponding to a particular observation (each channel of the filterbank) will be normalized appropriately. This can be

extremely 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 implicitely from the number of observations. Figure 1 shows the difference between *Implicit Patching* (left) where the dotted lines are created automatically from the semantics of compositing and *Explicit Patching* (right) where each connection must be created separately. Even though environments such as Max/MSP or PD provide subpatching the burden of internal patching is still on the user.

Some examples from audio analysis, synthesis and distributed computation are used to illustrate how *Implicit Patching* can be used in practice. They all have been implemented in *Marsyas-0.2* and their source code is part of the distribution. Figure 2 shows how a layer of nodes in an Artificial Neural Network can be expressed using a *Fanout*. The input to the layer (the output of the previous layer) consists 4 numbers 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 2 illustrates this process graphically (left side) and contrasts it with explicit patching (right side). In *Marsyas-0.2*, creating an artificial neural network using an annNode *MarSystem* is simply a series of fanouts of annNodes (more specifically seriesNet(fanoutLayer1, fanoutLayer2, ..., fanoutLayerM) where fanoutLayer1(annNode11, annNode12, ..., annNode1N). All the connections are created implicitely. The complete feature extraction frontend described in [14], has also been implemented as a dataflow network in *Marsyas-0.2*. The Phasevocoder is a computationally intensive audio algorithm that allows independent control of pitch and time shifting of sounds. The phasevocoder, in *Marsyas-0.2*, can be fully specified as a dataflow network created with *Implicit Patching*. Even on a Pentium III laptop, this implementation real-time.

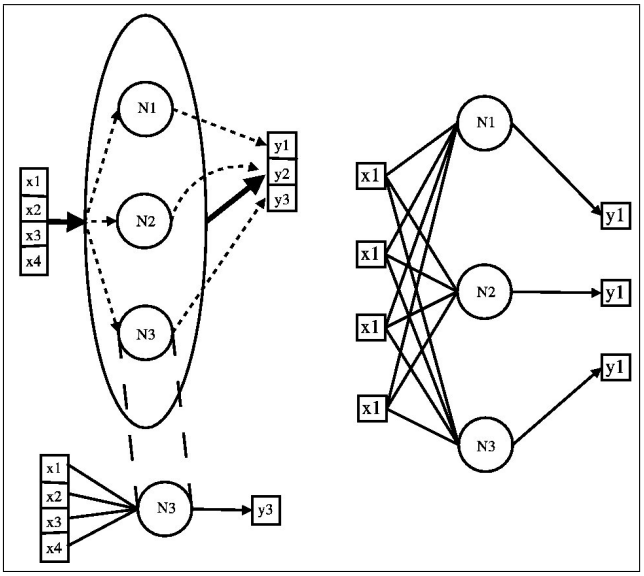

Figure 2. Layer of an Artificial Neural Network

5. REFERENCES

[1] W. Wadge and E. Ashcroft, *Lucid, the dataflow programming language*, ser. APIC Studies in Data Processing. New York, NY: Academic Press, 1985.

[2] W. Johnston, J. Paul Hanna, and R. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, 2004.

[3] D.-A. Manolescu, "A data flow pattern language," in *Proceedings of the 4th Pattern Languages of Programming*, Monticello, Illinois, September 1997.

[4] R. Boulanger, *The Csound book*. Cambridge, Mass.: MIT Press, 2000.

[5] M. Puckette, "Max at seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[6] X. Amatriain, "An object-oriented metamodel for digital signal processing with a focus on audio and music," Ph.D. dissertation, Univ. of Pompeu Fabra, 2005.

[7] R. Dannenberg, "Machine tongues xix: Nyquist, a language for composition and osound," *Computer Music Journal*, vol. 21, no. 3, pp. 50–60, 1997.

[8] G. Tzanetakis and P. Cook, "Marsyas: A framework for audio analysis," *Organised Sound*, vol. 4(3), 2000.

[9] P. Cook and G. Scavone, "The Synthesis Toolkit (STK), version 2.1," in *Proc. Int. Computer Music Conf. ICMC*. Beijing, China: ICMA, Oct. 1999.

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

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

[12] D. Schwarz and Wright.M., "Extensions and applications of the sdif sound description interchange format," in *Proc. Int. Computer Music Conf.*, 2000.

[13] M. Wright and A. Freed, "Open sound control: A new protocol for communicating with sound syntesizers," in *Proc. Int. Computer Music Conf. (ICMC)*, Thessaloniki, Greece, 1997.

[14] G. Tzanetakis and P. Cook, "Musical Genre Classification of Audio Signals," *IEEE Trans. on Speech and Audio Processing*, vol. 10, no. 5, July 2002.

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