# An Open Architecture for Real-Time Audio Processing Software

Amar Chaudhary Adrian Freed Matthew Wright
Center For New Music and Audio Technlogies
University of California, Berkeley, CA 94709, USA
{amar,adrian,matt}@cnmat.berkeley.edu

**Abstract**

OSW, or "Open Sound World," allows development of audio applications using patching, C++, high-level specifications and scripting. In OSW, components called "transforms" are dynamically configured into larger units called "patches." New components can be expressed using familiar mathematical definitions without deep knowledge of C++. High-level specifications of transforms are created using the "Externalizer," and are compiled and loaded into a running OSW environment. The data used by transforms can have any valid C++ type. OSW uses a reactive real-time scheduler that safely and efficiently handles multiple processors, time sources and synchronous dataflows.

## 1 Introduction

We introduce "Open Sound World" (OSW), a scaleable, extensible object-oriented language that allows sound designers and musicians to process sound in response to expressive real-time control.

Real-time software synthesis packages such as Max/MSP [1] and FTS are designed using a simple software component model. Users specify the signal and event flow through instantiations of high-level components, which are themselves created in the C programming language [2] and loaded on demand. This scheme works well until a new component function is needed. The component programmer is exposed to low-level efficiency and scheduling concerns and must express ideas in a low-level language (C) using predetermined, constrained data structures for inter-component communication. Developing components is difficult even for experienced programmers. Pd [3] introduced hierarchical, user-definable data structures for components. OSW builds on these ideas with an extensible object-oriented model which allows users to develop at multiple levels including visual patching, high-level C++ and Tcl scripts.

OSW includes the "Externalizer," a tool that allows users to view and extend the functionality of existing components or specify entirely new components as high-level specifications. The Externalizer automatically converts a specification into high-performance C++ code. The "transform" code is then compiled and can be loaded into a running OSW environment. Thus, new transforms can be specified and tested *in vivo* without having to stop and switch between development and testing phases.

The real-time scheduler used by OSW supports symmetric multiprocessor computers, as well as configurations with multiple audio devices and time sources. Fine-grain synchronization primitives (i.e., locks) are included to protect against non-deterministic behavior that arises in such parallel systems without severely compromising performance.

The remainder of this paper is organized as follows: Section 2 describes the basic features of programming in OSW; section 3 discusses the use of C++ and Tcl in OSW and introduces the Externalizer; section 4 explores timing and scheduling issues; section 5 describes the use of OSW in networked environments and section 6 concludes the paper.

## 2 OSW Basics

OSW is a "dataflow programming language." In dataflow programming, users connect primitive components together to produce networks. Each component accepts data from incoming connections and sends processed results via outgoing connections. OSW is also an "object-oriented" language, in that components are *instances* of *classes* that specify their structure and behavior.

OSW employs a visual programming environment that allows users to instantiate and connect graphical representations of the components. Examples of other visual dataflow languages include Ptolemy for signal processing [4], and Max/MSP and Pd for computer music applications. The look and feel of OSW is familiar to anyone who has programmed in these languages.

This section is not intended as a tutorial for OSW, but rather a presentation of its features. Detailed documentation and tutorials for OSW can be obtained elsewhere [5].

### 2.1 Transforms and Patches

The primitive components in OSW are called *transforms.* Transforms accept data via *inlets*, and produce results in *outlets.* Figure 1 illustrates a transform that generates a sine wave (i.e., pure tone) as a function of time and frequency. This transform has two inlets, timeIn and frequency, for accepting new time and frequency values, and one outlet, samplesOut, to which newly generated samples are sent. A transform can range in complexity from adding two numbers together to modeling a bank of hundreds of resonant filters. Because processing is more efficient within a transform than between transforms, more complex transforms are often favored.

Users instantiate new transforms by specifying the name of a transform class (e.g., Sinewave) followed by a name of the instance and arguments for initializing inlets or other attributes of the transform. An argument is the attribute name prefixed with a dash "-" character and followed by the desired value. For example, if we wanted to instantiate the Sinewave in figure 1 with a default pitch of Concert A, we might say "Sinewave sinewave1 -frequency 440.0".

Transforms can be connected to form larger networks called *patches*, as illustrated in figure 2. Patches are themselves transforms, and can be instantiated and incorporated into other patches.

Connections in OSW are "strongly typed," meaning that an outlet can only be connected to an inlet that accepts data of its type. Moreover, each outlet is connected to at most one inlet. In order to connect an outlet to multiple inlets, an explicit FanOut transform must be used. Likewise, inlets accept only one connection. A FanIn transform is included to allow an inlet to accept data from any connected outlet. Incoming data from multiple outlets can also be combined using other operators, such as addition, by including an appropriate transform.

By default, the visual appearance of a transform is a box containing the text used to instantiate it. As exemplified by figure 2, many transforms override this default appearance, displaying an icon or providing GUI controls for user input.

Transforms are intended to be self-documenting. Each transform includes a brief description of itself, a link to full documentation in HTML or XML format [6], and (like Max/MSP) a reference to an example patch illustrating the use of the transform. For Sinewave, the example patch might look like figure 2. Each inlet and outlet also includes a description of its type and use in its transform. This description is automatically displayed in the visual programming environment when the user moves the mouse over an inlet or outlet.

OSW includes a large set of standard transforms for basic event and signal processing. A current list of the standard transforms included in OSW can be found at `http://www.cnmat.berkeley.edu/OSW`. The set of available transforms can be easily extended to include more advanced operations.

## 2.2 Flow of Execution in Patches

The work of transforms is done in *activation expressions*, code that is executed in response to changes in one or more of the transform's inlets. The result of an activation expression is usually assigned to one or more outlets of the transform. Whenever an outlet is assigned a new value, the value is sent to a connected inlet. If the receiving inlet is *active*, the transform will execute an activation expression, possibly changing the value of its outlets. If the receiving inlet is *passive*, it is assigned the new value but no further processing occurs.

Consider the multiplication operator shown in figure 3. The left inlet inlet is active, so changing its value will trigger an activation expression that outputs the product of the two inlets. The right inlet is passive, so changing its value will not result in any output, but its new value will be used to compute products in subsequent activations.

## 2.3 Data Types

OSW provides a set of primitive data types, such as integers, floating-point numbers, strings, boolean values and lists, as well as several useful data types for music and signal-processing applications, including samples (as floating-point numbers or integers), frequency-domain spectra, notes, MIDI events and the Sound Description Interchange Format (SDIF) [7].

Since the data types used by transforms are C++ types (i.e., classes or primitive scalars), it is relatively straightforward to add new data types to OSW. Of course, new data types require new transforms to handle them. These issues are discussed in section 3.

## 2.4 The Name Space

The hierarchy of nested patches has a natural corresponding hierarchical name space. For example, if the patch illustrated in figure 2 is named sinewaveplayer1, then the full *path name* of our sine wave transform is /sinewaveplayer1/sinewave1, and its frequency inlet is /sinewaveplayer1/sinewave1/frequency. The hierarchical name space is modeled after the directory structure found in most file systems. Names beginning with a slash ("/") are treated as absolute path names, while names that do not begin with a slash are referenced relative to the current patch. Names beginning with two periods and a slash ("../") are referenced relative to the patch that contains the current patch. If there is no such container patch exists, addresses beginning with ("../") are undefined.

For example, suppose a user wanted to use a wavetable to generate a sine wave instead of directly computing the samples, as Sinewave does. The transform WaveTable generates samples by looking up amplitudes in a table according to phase. WaveTable requires an argument specifying the name of the table to use. In figure 4, we specify the table /tables/sine as an absolute path to a default table supplied by OSW. However, suppose the user has grown weary of listening only to sine waves, and wishes to supply his or her own table. He or she can add a new Table to the patch, as shown in figure 5, and replace /tables/sine with the relative path mytable, the name of the newly-created table.

### 2.4.1 Get and Set

The transforms Get and Set can be used to query or modify variables by path name. The variable can be part of a transform (e.g., an inlet or outlet), or a *free variable* (i.e., a variable that is not part of a transform) in a patch, as illustrated in figure 6. In this example, one patch uses Set to assign a value to free variable, and a second patch receives the value via a Get transform with an absolute path name to the variable in the first patch. Note in figure 6 that the free variable is not global, but defined as part of the patch /source. It can be accessed by a relative path name within /source and an absolute pathname elsewhere. In order to make a free variable myvariable global (i.e., not part of any patch), one would specify the absolute path /myvariable.

Note the special syntax of Get and Set. There is no instance name, and the name of the variable being accessed must be the first argument. Get also takes an optional argument, "-order *n*" that assigns a number to the transform. If more than one Get accesses the same variable, they output in ascending order according to this number. (Two such transforms with the same order number output in an indeterminate order.)

The use of Get and Set in patches is analogous to the use of goto in structured programming [8][9]. Their expressive power comes from the fact that they break the dataflow model, allowing the flow of execution to jump between transforms that are not connected or even in the same patch. Abusing this feature can make programs very difficult to understand and debug.

## 2.5 Packages

The previous section described a hierarchical name space for instances of transforms and patches. Similarly, transform class names are grouped into name spaces called *packages*. A transform that is part of a particular package is specified as *packageName*::*transformName*. If no package name is specified, a default list of packages is searched to find the transform class.

Packages are useful for organizing transforms, particularly in environments where new transforms are being developed or installed. A developer can modify an existing transform, and place it in a separate package to keep the class name but not interfere with patches that use the old transform. Likewise, a user can install new sets of transforms in separate packages according to the individuals or organizations that developed them, allowing two or more transforms with the same name from different sources to be used without confusion.

All the standard transforms are in the package osw, but explicit use of the package name is not necessary, as this package is always on the default list. Transforms that implement additive synthesis or resonance modeling techniques developed at CNMAT, for example, are part of the package cnmat, and the transform class for a bank of resonant filters would be cnmat::Resonators. If a user frequently uses transforms from the package, the package name can be added to the default list, and the cnmat package name need not be specified.

## 2.6 Accessing Hardware Devices

Most users will need to access various hardware devices for audio I/O and controller input. OSW provides abstractions of the devices themselves, as well as transforms for communicating with them. For example, an audio output device is required to realize the sound in each of the examples presented. The DAC transform is the interface at the patching level to the audio output. DAC requires a parameter specifying the name of the desired device. In this case, the first channel of the first device, /dac/0/0 was used. If we wanted to use all the available audio channels for the device, we would have used /dac/0 and each channel would have been available as a separate inlet to the transform.

OSW supports a variety of input and output devices, including audio hardware, MIDI ports, Ethernet and serial ports. It can be extended to support additional devices for expressive control, such as graphics tablets [10].

## 2.7 Type any and Dynamically-Typed Inlets

Some transforms include an outlet of type any. Such outlets can be connected to any inlet. Every time a new data value is sent from the outlet to the connected inlet, the type of the data must be checked to ensure that it is compatible with the inlet. If it not, an error occurs and the data is not processed by the receiving transform. The use of any is inefficient, and undermines type checking, so it is not widely used. The most common examples of any are free variables used with Get and Set.

Some transforms include *dynamically-typed inlets*. Dynamic-typed inlets are assigned a type at connection time. For example, a binary arithmetic operator, such as the multiplication operator shown in figure 7, has two dynamically-typed inlets. When the inlets are connected to outlets of valid arithmetic types (e.g., two integers, or samples and a floating-point number), the inlets are assigned those types, and the outlet of the operator is assigned a type according to the operator and the inlet types (e.g., multiplying two integers produces an integer, while multiplying samples by a floating-point number produces samples). If the outlet was already connected to another transform, the connection is broken. Like any, dynamically-typed inlets allow a single transform to be used with many different types of data. However, because a dynamically-typed inlet has the same type as the outlet to which it is connected, no dynamic type-checking is necessary. Thus, they do not incur the performance penalty associated with any.

## 2.8 FanOut and FanIn

As stated earlier, an explicit FanOut transform is needed to connect one outlet to many inlets. Like Get and Set, FanOut does not take an explicit instance name (one is created automatically), but does take an argument -outputs for the number of outlets. If -outputs is not specified, a fanout transform with two outputs is created. The single inlet of FanOut is dynamically-typed. When the inlet is connected, all the outlets of the FanOut are assigned the new inlet type. The syntax of FanIn is equivalent to FanOut, except that it takes an argument -inputs to determine the number of inlets. Any input received by any inlet of FanIn will be output.

## 2.9 Bundles and Transform Arrays

Users of dataflow languages for signal processing often find themselves making several copies of the same group of connected transforms. This frequently happens when dealing with multiple audio channels or multiple voices in a synthesizer. Of course, the copied transforms can be placed in separate patch and the patch can be instantiated multiple times. OSW provides an additional abstraction for multiple copies: transform arrays. The Array transform takes a transform class name, an integer $n$, an instance name and arguments for the transform class, and creates a single object containing $n$ copies of the transform, named $name/0 \ldots name/(n-1)$. The array has inlets and outlets which have the same names as the original transform but each now must be connected using a *bundle*. The bundle data type is an abstraction of busses in audio engineering. It is used to transmit $n$ data objects of another type over a single connection.

Standard connections are converted to and from bundles using the transforms BundleOne, BundleMany and Unbundle. BundleOne has one inlet and copies incoming values to each connection in the bundle, while BundleMany includes a separate inlet for each connection in the bundle, allowing different values to be sent to different transforms in an array. Unbundle converts a bundle of $n$ connections into $n$ separate standard connections.

Figures 8 and 9 illustrate how transform arrays and bundles can reduce clutter in a patch that includes three sine waves with independent frequency and amplitude controls.

## 3 Transform Implementation

Transforms as well as the entire OSW system are implemented using standard C++. By doing so, we present a unified, object-oriented approach to transforms, patches, devices, the scheduler, etc. Moreover, compilers can take advantage of C++ optimizations like inlining across different components of the system. (This would be exceedingly difficult in a system based on C functions called from separate modules.)

Tcl/Tk [11] is a scripting language and user-interface toolkit that runs on virtually every modern platform. As such, it is an ideal choice for implementing the visual programming environment and user-interface transforms (e.g., buttons, sliders, etc.). Tcl scripts are also used to implement patches.

In addition to describing implementation issues, we hope to motivate the use of these implementation languages as options for developing high-performance signal-processing applications in OSW.

### 3.1 The Externalizer

In most component-based systems, the primitive components are completely opaque to users. The internals of a component cannot be viewed or modified except by experienced developers armed with extensive knowledge of a low-level language (such as C), the host operating system and a specialized toolkit.

OSW provides a graphical tool called the Externalizer that allows users to "peer under the hood" of a transform and extend its behavior without a deep knowledge of C++ or low-level efficiency concerns.

The Externalizer presents a transform implementation as a collection of inlets, outlets, *state variables* and *activation expressions* that a user can view or modify. A state variable is a transform parameter that does not appear as an inlet. Like inlets and outlets, state variables are part of the hierarchical namespace and can be queried or modified using Get and Set transforms.

As described earlier, an activation expression is a piece of C++ code that is executed when certain inlets or state variables are modified. If an activation depends on more than one variable, then all the variables must be modified before it will be executed. An activation can occur immediately, or be delayed for a specified amount of time before running. An activation expression is specified by the subset of inlets or state variables that trigger this activation, whether this activation should occur immediately after the variables are changed or be delayed by a certain amount of time, and the code that should be executed. An activation expression can also be assigned an optional integer that determines its relative order among all expressions triggered by a particular variable. The ordering rules are the same as those described for Get transforms.

Consider the following specification of Sinewave:

Sinewave
Generates a pure tone (i.e., sine wave) signal.

| | Name | Type | Default |
|---|---|---|---|
| Inlets | timeIn | Time | |
| | frequency | float | 440.0 |
| Outlets | samplesOut | Samples | |
| State Variables | prevTime | Time | 0.0 |
| Inherited | SampleRate | float | 44100.0 |
| | NumberOfSamples | int | 128 |

| | |
|---|---|
| Activation Expression | activation1 |
| Depends on | timeIn |
| Delay | none |

```
samplesOut = sin(TWOPI * frequency
                 * Range(prevTime,timeIn,NumberOfSamples));
prevTime = timeIn;
```

Whenever a new time value is received via timeIn, the sine function is computed with frequency frequency over a range of NumberOfSamples values from prevTime and timeIn[*]. More specifically, the Range function creates a vector of interpolated values between prevTime and timeIn. Computing the sine function over the product of $2\pi$ frequency and this vector yields another vector containing the desired output samples, which are then sent to samplesOut. The ending time value (i.e., timeIn) will be the starting time value the next time this expression is evaluated.

The state variables NumberOfSamples and SampleRate are *inherited* from a more general class of transforms that share a set of common features. In this case, the more general class is *time-domain transforms* that manipulate time-domain samples. Inherited variables can be used in activation expressions just like other state variables. In most activation expressions, a new value is assigned to an outlet. Such an assignment is called an *effect*. Effects allow users to understand the flow of execution in a transform and the behavior of patches that incorporate it.

### 3.1.1 Using the Externalizer

Users can invoke the Externalizer by selecting a transform to examine from a patch. The specification of the transform class is then displayed. This specification includes the name of the transform class, its package, a brief description, references to its doc-

[*]In practice, the formula used in this activation expression should be replaced with one that does not cause phase discontinuities when the frequency changes

umentation and example patch, any special base classes (e.g., time-domain transform) and lists of its inlets, outlets, state variables, inherited variables and activation expressions. The user can then select any of these items to call up for detailed information and make modifications.

Suppose, for example, that a user wishes to make a version of Gain that uses a decibel (i.e., logarithmic) scale instead of a linear scale. Figure 10 shows the specification of Gain as presented by the Externalizer, and figures 11 and 12 show the activation expression before and after the modification, respectively.

When the user finishes modifying a transform specification, it is automatically converted to a C++ class that implements the transform. The C++ code is then compiled into a dynamic library which can be loaded into OSW and used to instantiate transforms of the new class.

In order to protect the integrity of the OSW environment, users cannot directly modify the standard library of transforms. However, most of the standard transforms include Externalizer specifications that can be copied and then modified (as a new transform class).

The default policy of the Externalizer is to bundle a copy of the transform specification along with the dynamic library so that it may be examined and extended by other users. However, developers of transforms with proprietary code or algorithms can block this feature. Transforms without accompanying specifications cannot be viewed or modified except by the original developer.

### 3.1.2 Adding Data Types

The Externalizer also allows users to define new data types for transform variables. A new data type consists of a number of named fields, each of which must be a preexisting OSW data type. A conversion to and from the string data type must be included for human-readable input and output. The data type specification is then converted to a C++ `struct` definition for use in transforms.

### 3.1.3 More Externalizer Features

Users can provide C++ code or Tcl scripts that override the default appearance and behavior of the transform in the visual programming environment. Users can also specify code to be executed when an instance of the transform is created or destroyed (e.g., opening and closing files). Additional "private" C++ variables and functions can be specified and used in activation expressions.

## 3.2 Transforms in C++

Experienced programmers can bypass the Externalizer and write C++ transform classes directly. Each transform class is derived from one of the C++ base transform classes (at present, either `Transform` or `TimeDomainTransform`). Each derived class includes members for its inlets, outlets, state variables and activations, a member function for each activation and a constructor function that properly instantiates each member.

```
class Sinewave : public TimeDomainTransform {

public:

   Inlet<Time>      timeIn;
   Inlet<float>     frequency;
   Outlet<Samples>  samplesOut;
   State<Time>      prevTime;

   Sinewave (const string &aname, Patch *acontainer,
            int argc, char *argv[]) :
       TimeDomainTransform(aname,acontainer,argc,argv),
       timeIn("timeIn",this,"Evaluate the sine function up to this time"),
       frequency("frequency",this,"Frequency of the sine wave"),
       samplesOut("samplesOut",this,"Output signal"),
       prevTime("prevTime",this,"Previous time value",0.0),
       activation1(&samplesIn,TIME_NOW,this,&Sinewave::activation1Expr) {
   }

private:

   Activation<Sinewave>  activation1;

   void activation1Expr () {
       samplesOut = sin(TWOPI * frequency
                              * Range(prevTime,timeIn,NumberOfSamples));
       prevTime = timeIn;
   }

};
```

Inlets, outlets and state variables are specified using C++ *templates*. Templates allow developers to specify a generic version of a class or function that can be used with different types. In this case, the generic `Inlet` and `Outlet` templates implement the underlying mechanisms for type-safe connections, triggering activation expressions and self documentation. The developer can then use these mechanisms with any C++ data type.

The `Activation` template is an example of a *functor*, or "functional object" [12]. The functor is created by combining a piece of code (`activation1Expr` in the example) with a context (i.e., depends on `timeIn`, no delay). Because calls to the functor can be inlined, the code in `activation1Expr` will be called without the overhead of subroutine calls.

### 3.2.1 The `osw::vector<T>` Class

Both Externalizer specifications and C++ transform classes allow users to specify activation expressions using intuitive, familiar mathematical definitions instead of hand-optimized computer code [13]. This is achieved through the use of function and operator overloading [14] in expressions that use OSW's vector template class, `osw::vector<T>`[†]. The `Samples` type is actually a synonym for `osw::vector<float>`, an optimized vector of single-precision floating-point numbers.

When a standard arithmetic operator or math function is called with a vector argument, a *composition closure object* [12] is created that evaluates the entire expression for each element in the vector, eliminating the need for temporary storage in complex arithmetic expressions. Like functors, calls to closure objects are inlined. Consider the activation expression from Sinewave:

```
samplesOut = sin(TWOPI*frequency*Range(prevTime,timeIn,NumberOfSamples));
```

A naive implementation would first create a vector for the range with `NumberOfSamples` elements, a temporary vector for the multiplication by `TWOPI` and `frequency` and a final vector containing the sine of each element from the temporary. The intermediate allocation wastes time and space, and also breaks standard loop optimizations, such as unrolling. The composition closure object defines a function that creates only the final vector. Each element of the final vector is assigned the sine of the multiplication of `TWOPI`, `Frequency` and the corresponding element of the range. The temporary allocation has been eliminated, and the resulting simple loop can now be further optimized by the compiler.

The `osw::vector<T>` class also overrides the standard C++ allocators to implement a deferred reference-counted memory manager. In this scheme, further allocations are saved when a vector is simply passed without modification from one variable to another; its reference count is increased whenever it is assigned to a new variable, and decreased when it unassigned or a variable goes out of scope. If the reference count decreases to zero, the vector is deallocated and its memory is returned to a shared pool for use by other vectors. In order to reduce the overhead of deallocation during sequences of signal-processing transforms, this task is deferred until a suitable time in the future, at which point all allocations since the last deferred deallocation are scanned, and all vectors with a reference count of zero are returned to the share pool. The scheduling of deferred deallocations is discussed in section 4.4.2.

## 3.3 Writing Patches and Transforms in Tcl

An OSW patch is implemented as a Tcl script. The script instantiates the transforms, establishes connections between them, and creates the standard graphical interface using Tk. A programmer can edit this script directly.

For example, the following lines of Tcl create the transforms and connections for the patch from figure 2.

[†]The explicit namespace `osw::` is used to avoid confusion with the Standard Template Library class `vector<T>`.

```
AddTransform $patch TimeMachine tm1
AddTransform $patch Sinewave sinewave1 -frequency 440.0
AddTransform $patch Gain gain1 -amp_scale 0.5
AddTransform $patch VSlider freqslider -from 100 -to 800 -step 0.1
AddTransform $patch VSlider ampslider -from 0 -to 1 -step 0.01
AddTransform $patch DAC dac1 -channel /dac/0/0

Connect $patch/tm1/timeOut $patch/sinewave1/timeIn
Connect $patch/sinewave1/samplesOut $patch/gain1/samplesIn
Connect $patch/gain1/samplesOut $patch/dac1/in0
Connect $patch/freqslider/out $patch/sinewave1/frequency
Connect $patch/ampslider/out $patch/gain/amp_scale
```

In this script, `$patch` is variable containing the path name of patch being instantiated. So far, this script does not display anything. (Although two VSlider transforms are instantiated, the corresponding GUI slider objects have not yet been created). The remainder of the script contains commands that build the default OSW visual programming interface, displaying the transforms and their connections. This default visualization can be replaced with a panel that contains only the appropriate slider controls using the following Tcl script:

```
toplevel .sinewavepanel
scale .sinewavepanel.frequency -from 100 -to 800 -resolution 0.1 \
     -orient horizontal \
     -command \"oswSet /$patch/sinewave1/frequency\"
scale .sinewavepanel.amplitude -from 0 -to 1 -resolution 0.01 \
     -orient horizontal \
     -command \"oswSet /$patch/gain1/amp_scale\"
pack .sinewavepanel.frequency .sinewavepanel.amplitude -side top
MakePatchWindow .sinewavepanel
```

The `MakePatchWindow` command replaces any default window created for this patch with the custom window `.sinewavepanel`. When a user opens this patch, the customized interface (shown in figure 13) will replace the original interface of figure 2.

Customizing the user interface of patches is particularly useful for developing standalone applications based on OSW [15].

## 4 Scheduling

We now turn our attention from the language issues associated with writing programs in OSW to the scheduling issues associated with running them. This section discusses the real-time constraints imposed on the OSW runtime system as well as the time and dataflow models in OSW programs before describing the scheduler itself.

## 4.1 Reactive Real-Time Constraints

OSW is designed for implementing *reactive real-time* audio and music applications [16]. Reactive real-time involves maintaining output quality while minimizing *latency*, the delay between input and output of the system, and *jitter*, the change in latency over time [17].

First, we want to maintain continuous audio output quality. If the system falls behind, it will produce unpleasant clicks and gaps in the audio output. This situation can be avoided by using large buffers that regulate the output of samples into the device. However, buffers introduce latency into system. For audio and music applications, we would like to keep the input-to-output latency under 10ms [18]. We therefore bound the size of the buffered output to be less than 10ms worth of sound. However, if we allow the delay be any size less than 10ms, we may introduce jitter into the system. Humans are very sensitive to jitter [19], especially if it involves the response of sound output to their own gestures. We therefore process audio in very small chunks (e.g., only 1ms of sound), and queue no more than 10ms of these small chunks in order to minimize overall latency while still preventing audio glitches. To minimize jitter, we stipulate that as soon as there is less than 10ms of sound in the queue, a new small chunk should be added. Of course, if the particular system can handle a bounded delay of less then 10ms, the lower value should be used.

These real-time *distance constraints* [20] of bounded delay and minimal jitter are managed by OSW's timing model and scheduler.

## 4.2 Time in OSW

The flow of time in sound and music can be interpreted differently in different situations. *Real time* is a quantity that always increases at a fixed rate, as measured by a clock. *Virtual time* [21] is a variable-rate quantity that can be scaled or translated. Familiar examples of virtual time include the changing tempo of music, fast-forward or rewind functions on a VCR, and changing the speed of a record player.

OSW includes state variables, called *clocks*, that measure real time in seconds using double-precision floating-point numbers. Clocks are associated with the time sources in the computing environment, including the computer clock, audio devices (i.e., sound cards) and network time sources [22]. A clock is updated at a regular interval, called a *period*. Virtual time is handled by transforms called *time machines*. Time machines can be synchronized to clocks or other time machines, as illustrated in figure 14. Transforms that implement functions of time (e.g., Sinewave) are typically connected to the output of time machines. Like Get, time machines have an optional argument -order that determines their relative order when synchronized to the same clock.

OSW includes a master clock, which can be assigned by the user to any time source. By default, the master clock is assigned to the primary audio output device. The master clock is used for scheduling delayed activations and as the default clock for time machines.

Clocks and time machines play a crucial role in maintaining real-time constraints. Consider the default configuration in which all time machines are synchronized to the audio output device via the main clock. An *audio output device* is a transform that has only state variables but no inlets or outlets and is not part of any patch. However, its state variables can be accessed using Get or Set. An audio output device accepts samples from DAC transforms and sends those samples to the actual device. (If more than one DAC transform sends samples to a device during a period, the samples are mixed.)

Recall from section 4.1 that we bound the latency of our system by bounding the size of the queued output while at the same time ensuring that the queue is always full enough to guarantee that we never have discontinuities in the audio ouput. It is the job of the audio output device to satisfy these opposing constraints. The audio output device has two state variables that represent these constraints, SampleBufferSize and TargetLatency. SampleBufferSize is the number of samples that are sent to the device at once, and TargetLatency is the total number of samples that are allowed to be placed in the output queue awaiting realization by the sound hardware. Because TargetLatency controls overall latency and SampleBufferSize controls jitter, we want both of these quantities to be as low as allowed by the device and operating system.

In order to fulfill real-time requirements, the audio output device has to be able to control when the signal processing that produces the samples it will output is performed. This is accomplished by controlling virtual time sources via the device clock (or the main clock if it is assigned to this device). The audio output abstraction includes an activation expression that depends on the clock:

```
FlushSamples();
while(SamplesInQueue() > TargetLatency) {
   Wait();
}
clock = clock + SampleBufferSize / SampleRate;
```

The `Wait` operation is system dependent, and may include deferring to another thread or process to perform other events such as MIDI input. `FlushSamples` outputs the samples for this period. (The number of samples output is the sample-buffer size.)

When the clock is updated, it triggers several activation expressions. The queue of deferred activations has order *minint*, or the smallest integer allowed, thus assuring that it will be activated first. Any deferred activations that are scheduled to occur during this period are then evaluated. The clock then triggers the audio input device, whose activation has order $minint + 1$. The audio input device then reads a period of samples into a buffer and activates any ADC transforms in the program. All the time machines are then activated according to their relative ordering (as set by the -order option). Finally, the clock re-triggers the activation expression of the audio output device, thus completing the loop. The audio output activation is assigned an order of *maxint* (the largest available integer) and will be the last activation triggered by the clock during this period. By ordering the activations in this manner, we guarantee that in each period we first receive audio input, then perform audio processing that may depend on the input, and finally send the results to the audio output.

This timing model can be generalized to multiple audio output devices with independent clocks, as well as other clock sources.

## 4.3 Synchronous and Asynchronous Execution

A common type of patch is one in which a transform that produces samples as a function of virtual time (from a connected time machine) is followed by a succession of transforms that modify the samples and finally send them to an audio output device. Such a set of transforms constitute a *synchronous chain*. Activation expressions in transforms of a synchronous chain are guaranteed to occur exactly once each period of the clock to which they are synchronized. Moreover, the order in which they are executed is fixed within a period. This is true of more general *synchronous dataflow graphs*, which can include multiple dataflow paths (e.g., FanOuts) as long as they are synchronized to the same clock. Most signal processing in OSW occurs within synchronous dataflows.

Each time-domain transform (i.e., a transform that produces samples as a function of time or input samples, as described in section 3.1) has state variables for its sample rate and the number of samples that are processed during one period. The sample rate and buffer size of all transforms in a synchronous dataflow graph must be equal, unless the graph includes an explicit sample-rate conversion transform [23], in which case the values must be equal for all transforms before the conversion and for all transforms after the conversion. The sample rate or vector size can be explicitly set in an audio input (ADC) or output (DAC) transform. The values are then propagated to all connected time-domain transforms.

Because OSW allows multiple audio devices and clock sources, several synchronous dataflow graphs that run at different sample rates and buffer sizes are supported. For example we may want to use an audio input running at a low sample rate (e.g., 11025Hz) for pitch detection, and send the resulting pitch values (i.e., as synchronous signals or asynchronous events) to a synthesizer that generates samples at a higher rate (e.g, 44.1Khz). More generally, some audio channels in a signal-processing application may be used for transmitting continuous control information instead of sound [24]. Although the guarantees of synchronicity and fixed order of execution remain true within the synchronous graphs of individual channels, interaction between graphs requires explicit synchronization with one clock, usually the master clock. Otherwise, the interaction is asynchronous and non-deterministic.

Other examples of asynchronous events include input from the user interface or a MIDI device. Unlike transforms in synchronous chains, the relative execution times of asynchronous transforms cannot be predicted. If any of the coefficients of a filter are updated asynchronously while the filter is running, the filter may become unstable. Thus, it is necessary to protect inlets and state variables that are sensitive to asynchronous updates. Such protection is included in the more general parallel scheduler described in the next section.

## 4.4 Parallelism

Asynchronous events are a special case of parallelism. In parallel environments, two or more tasks are executed simultaneously, or at an indeterminate time relative to one other.

Parallel processing offers greater throughput for computationally intensive tasks (like signal processing) at the expense of greater hardware cost and software complexity. However, as processing hardware becomes cheaper and more powerful, computers with two or more processors are becoming more widely available. OSW is designed to take advantage of such multi-processor capabilities when they are present. We describe the parallel scheduling scheme used by OSW, including how to protect sensitive variables against asynchronous updates and non-determinism in parallel environments.

### 4.4.1 A Formal Approach

Given enough processors, each transform could run on its own processor, executing activation expressions whenever its inlets or state variables change. Processor utilization would be poor. If there are two transforms $T_1$ and $T_2$ such that an outlet of $T_1$ is connected to an inlet of $T_2$, then $T_1$ must be executed before $T_2$.

Consider all finite sequences of transforms $X = T_1, T_2, \ldots, T_n$ such that $T_1$ is connected to $T_2$, etc. If there exists no transform $T$ such that an outlet of $T_1, \ldots, T_{n-1}$ connects to an inlet of $T$ or an outlet of $T$ connects to an inlet of $T_2, \ldots, T_n$, then $X$ is a *chain*. More informally, a chain is a sequence of connected transforms in which there are no branches. We say that $X$ is a *maximal chain* if there is no chain $Y$ such that $X \subset Y$ (i.e., if we add any more transforms to $X$, $X$ will no longer be a chain). Each maximal chain must be scheduled sequentially, while separate maximal chains can run in parallel. Thus, maximal chains are considered the formal unit of parallel computation in OSW.

Examples of maximal chains that often occur in real programs include processing several channels of audio, or synthesizers with multiple voices (i.e., polyphony). A synchronous chain, as described in the previous section, is simply a chain in which $T_1$ is a time machine or an ADC transform and $T_2, \ldots, T_n$ are time-domain transforms.

A more general and rigorous analysis of dataflow and related models used in signal processing is provided by Lee and Parks [25].

### 4.4.2 The OSW Parallel Scheduler

We now present an algorithm for scheduling maximal chains. Given $N$ processors, we instantiate $N$ threads for processing the scheduled chains. Each chain-processing thread executes the following loop *ad infinitum*:

```
loop
  pop a chain off the queue and execute it.
end loop
```

The following greedy decision rule is used by a transform after an outlet has just been assigned to determine whether or not its connections are part of the current maximal chain:

```
if this transform has more than one outlet then
   The transform connected to this outlet begins a new maximal chain.
   Add it to the queue and return
else
   The next transform is part of the current chain.
   Assign the outlet value to the connected inlet and evaluate its activation
   expressions...
end if
```

Recall from section 4.2 that audio devices have activation expressions that both trigger and are trigged by a clock when an output device needs more samples. Thus, these tasks are already scheduled as part of maximal chains. However, there remain necessary tasks to perform, including scheduling asynchronous events and deferred deallocation of reference-counted memory. We can insert deferred deallocation in between the execution of chains:

```
loop
   pop a chain off the queue and execute it.
   Do deferred deallocations if no other thread is doing this.
end loop
```

Only one thread is allowed to perform deallocations at any given time, thus protecting the common memory pool and also preventing additional processors from being wasted on memory management.

Alternatively, deferred deallocation can be performed by a separate thread that runs at lower priority (i.e., is given less processing time) than the chain-processing threads. Additional threads are needed for handling the user interface as well as some asynchronous input devices, such as MIDI input ports[‡]. Although this means there will be more threads than processors, the asynchronous input threads will likely be idle much of the time and therefore should not be assigned dedicated processors.

### 4.4.3 Thread-safety and Synchronization

Consider the example of a 2-pole resonant filter transform, with inlets for input samplesIn and filter coefficients a, b1 and b2. It might contain the following activation expression:

```
Samples newsamples(NumberOfSamples);
for (int i = 0; i < NumberOfSamples; ++i) {
   newsamples[i] = x * samplesIn[i] + b1 * y1 + b2 * y2;
   y2 = y1;
   y1 = samplesIn[i];
}
samplesOut = newsamples;
```

[‡]The need for separate threads for devices depends on the device and the operating system.

where `y1` and `y2` always contain the previous two samples. A transform that is running in parallel or asynchronously may update the values of `a`, `b1` or `b2` while the activation expression is running. This will cause unpredictable and often disastrous results in the output sound (including loud clicks or saturation if the filter becomes unstable). We want to be able to guarantee that the values of the filter coefficients remain constant during the evaluation of the expression. More generally, we want activation expressions to be *thread-safe*, i.e., the evaluation of an expression on one thread will not be corrupted by the asynchronous actions of other threads [26].

To this end, OSW provides methods for locking sensitive variables [27] [28]. When a variable is locked, any attempts to assign it a new value will be deferred by placing the new value in a buffer and then copying the value from the buffer to the variable as soon as it is unlocked. This technique is called *double-buffering.* The filter expression can now be protected as follows:

```
Samples newsamples(NumberOfSamples);
a.Lock();
b1.Lock();
b2.Lock();
for (int i = 0; i < NumberOfSamples; ++i) {
   newsamples[i] = x * samplesIn[i] + b1 * y1 + b2 * y2;
   y2 = y1;
   y1 = samplesIn[i];
}
a.Unlock();
b1.Unlock();
b2.Unlock();
samplesOut = newsamples;
```

Any attempts to assign new values to the filter coefficients during the loop will be double-buffered and assigned to the variables when they are unlocked. If multiple assignments to the variables occur while they are locked, the most recent values are assigned and other assignments are discarded.

It should be noted that because vectors (like `samplesIn`) are stored as referenced-counted pointers, the copying operation requires only the pointer to be copied, not the entire vector. Thus, vector variables can also be efficiently protected using double-buffering.

## 5 OSW in Networked Environments

Most computers can operate in networked environments using such standards as Internet protocols, Ethernet, USB and 1394 FireWire. Computer networks are now used regularly for audio broadcast and have been used to control multiple computers in live interactive musical performances [29] [30].

Since OSW is being developed in the era of ubiquitous networking, its design incorporates features for networking technologies and environments

## 5.1 OpenSound Control

OpenSound Control (OSC) is a new protocol for high-level, expressive control of sound synthesis and other multimedia applications [31]. OSC divides the world into clients that generate control messages and servers that produce sound in response to these messages. Because OSC messages use a hierarchical addressing scheme that is similar to the OSW hierarchical name space, OSW is a "natural" server for OSC. OSW continuously monitors network connections selected by the user (e.g., a particular UDP port) and waits for incoming messages. If the address of an incoming message corresponds to an inlet, outlet or state variable in the OSW name space, the message is sent to that variable. If the message is a query for information about the variable, OSW returns the type, description and value of the variable. If it is a control message, then the data associated with the message is assigned to the variable. For example, an OSC client could remotely set the frequency of the sinewaveplayer patch to middle C by sending the message:

```
/sinewaveplayer1/sinewave1/frequency 261.626
```

If an incoming OSC message does not match any variable in the OSW namespace, a string for the address and the raw binary representation of the data are sent to any OSCReceive transforms in the program. If there are no running OSCReceive transforms, then the message is discarded.

## 5.2 Distributed Namespace

If a reference to a named OSW object (e.g., a variable name in a Get or Set transform) begins with two slashes ("//"), then it is considered to be an IP address on a network. For example, the path name //myserver.cnmat.berkeley.edu/mypatch/sinewave1/frequency refers to the frequency inlet of a Sinewave transform in a patch running on the machine `myserver`. Over IP networks, the remote machine specification also includes a port number after the machine name separated with a colon (e.g., //myserver:7777/...). If no port number is included, a default port is assumed. Requests to query or assign values to variables on other machines are performed using OSC messages. The network-aware name space can be used to build distributed OSW applications on local networks.

## 5.3 SDIF

The Sound Description Interchange Format (SDIF) is a new standard for storing and distributing sounds in different representations, including time-domain samples, frequency-domain spectra and other higher-level models. SDIF arranges data into sequences of data, called *streams*. A stream is time-ordered sequence of data structures, called *frames* that describe a sound using a particular representation. Each frame is associated with a frame type indicating the type of sound representation being used. The SDIF standard includes an extensible library of frame types. Interested readers are encouraged to find more about the SDIF specification, the currently supported sound representations and the applications that use SDIF [7].

OSW includes transforms and data types for manipulating SDIF streams and frames, including separate data types for the standard SDIF frame types, translating between SDIF frames and OSW vector and list types, and reading and writing of SDIF streams on local disks or networks.

### 5.4 Downloading Patches and Transforms

As described in section 3.3, OSW patches are Tcl scripts. Popular web browsers can download and execute Tcl scripts via a plug-in [32]. These scripts are also allowed to load dynamic libraries, provided that the user has deemed them "safe" and granted permission to run them on his or her computer.

We propose the use of OSW patches as platform for generalized "downloadable instruments" on the World Wide Web. For example, a user could select the sinewaveplayer patch from a server. The patch is downloaded and interpreted by the Tcl plug-in, which dynamically loads the OSW run-time environment and necessary transforms, and renders the custom user interface in the web browser.

Downloadable patches and SDIF can be used together to create customizable streaming audio players for different sound representations [33].

## 6 Discussion

We have implemented and tested OSW on Intel-based PC's running the Windows NT and 98 operating systems. We are currently porting OSW to the Linux operating system, Macintosh computers running MacOS and SGI workstations.

Most transforms and patches that do not access platform-specific functions (e.g., accessing hardware) can run on each platform without modification. Of course, C++ source code for each transform must be recompiled. Further work will evaluate different compilers as "back-ends" for the Externalizer. In particular, we require ANSI/ISO C++ compatibility [34] and agressive optimization.

In addition to the language and scheduling issues described in this paper, an important part of this project is its use by sound designers and musicians. OSW has been used successfully in live musical performance situations [35]. We hope that its continued use in creative applications will drive continued work on fundamental system issues and also stimulate the development of high-level transforms that realize new ideas for audio processing and expressive control of sound.

## 7 Acknowledgements

We gratefully acknowledge the NSF Graduate Research Fellowship Program for their support of this research. We would also like to thank David Wessel and Lawrence A. Rowe, directors of CNMAT and the Berkeley Multimedia Research Center, respectively, for their support; and Roger Dannenberg from the School of Computer Science at Carnegie Mellon University for his constructive feedback.

## References

[1] D. Zicarelli. “An Extensible Real-Time Signal Processing Environment for Max”. *International Computer Music Conference*, pages 463–466, Ann Arbor, MI, 1998. ICMA.

[2] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition.* PTR Prentice Hall, Englewood Cliffs, NJ, 1988.

[3] M. Puckette. “Pure Data: Another Integrated Computer Music Environment”. *Second Intercollege Computer Music Concerts*, pages 37–41, Tachikawa, Japan, 1996.

[4] J. Davis et al. “Heterogeneous Concurrent Modeling and Design in Java”. Technical Report UCB/ERL M98/72, EECS, University of California, November 23, 1998 1998. `http://ptolemy.eecs.berkeley.edu`.

[5] `http://www.cnmat.berkeley.edu/OSW`.

[6] `http://www.w3.org/XML/`.

[7] M. Wright, A. Chaudhary, A. Freed, S. Khoury, and D. Wessel. “Audio Applications of the Sound Description Interchange Format Standard”. *107th AES Convention*, New York, 1999. `http://www.cnmat.berkeley.edu/SDIF`.

[8] E. W. Dijkstra. “Go To Statement Considered Harmful”. *Communications of the ACM*, 11(8):538, 1968.

[9] D. Knuth. “Structured Programming with Go To Statements”. *Computing Surveys*, 6:261–301, 1974.

[10] M. Wright, D. Wessel, and A. Freed. “New Musical Control Structures from Standard Gestural Controllers”. *ICMC*, Thessaloniki, Greece, 1997. `http://cnmat.cnmat.berkeley.edu/ICMC97/GesturalControl.html`.

[11] B. B. Welch. *Practical programming in Tcl & Tk.* Prentice Hall PTR, Upper Saddle River, NJ, 2nd edition, 1997.

[12] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[13] A. Freed and A. Chaudhary. “Music Programming with the new Features of Standard C++”. *International Computer Music Conference*, pages 244–247, Ann Arbor, MI, 1998.

[14] T. Veldhuizen. “Scientific Computing: C++ Versus Fortran”. *Dr. Dobb’s Journal*, 22(11):34, 36–8, 91, 1997. Article Miller Freeman.

[15] A. Chaudhary. “Band-limited Simulation of Analog Synthesizer Modules by Additive Synthesis”. *105th AES Convention*, San Francisco, CA, 1998.

[16] R. Dannenberg and D. Jameson. "Real-Time Issues in Computer Music". *Proceedings of the Real-Time Systems Symposium*, pages 258–261. IEEE Computer Society Press, 1993.

[17] E. Brandt and R. Dannenberg. "Low-Latency Music Software Using Off-the-Shelf Operating Systems". *International Computer Music Conference*, pages 137–140, Ann Arbor, MI, 1998. ICMA.

[18] E. Clarke. "Rhythm and Timing in Music"'. Diana Deutsch, editor, *The Psychology of Music*, pages 473–500. Academic Press, San Diego, 1999.

[19] M. Tsuzaki and R. D. Patterson. "Jitter Detection: A Brief Review and Some New Experiments". A. Palmer, R. Summerfield, R. Meddis, and A. Rees, editors, *Proceedings of the Symposium on Hearing*, Grantham, UK, 1997.

[20] C. Han, K. Lin, and C. Hou. "Distance-Constrained Scheduling and Its Applications to Real-Time Systems". *Communications of the ACM*, 45(7):814–826, 1996.

[21] R. Dannenberg. "Real-Time Scheduling and Computer Accompaniment"'. Max Matthews and John Pierce, editors, *Current Research in Computer Music*. MIT Press, Cambridge, MA, 1989.

[22] D. Mills. "Simple Network Time Protocol (SNTP) Version 4 for Ipv4, Ipv6 and OSI". Technical Report Internet RFC 2030, 1996. `http://sunsite.auc.dk/RFC/rfc2030.html`.

[23] Y. Medan and U. Shvadron. "Asynchronous rate conversion". Y. Wang, A. R. Reibman, B. H. Juang, T. Chen, and S. Y. Kung, editors, *Proceedings of First Signal Processing Society Workshop on Multimedia Signal Processing*, pages 107–12, Princeton, NJ, USA, 1997. IEEE.

[24] A. Freed and D. Wessel. "Communication of Musical Gesture using the AES/EBU Digital Audio Standard". *International Computer Music Conference*, Ann Arbor, Michigan, 1998. ICMA. `http://cnmat.cnmat.berkeley.edu/ICMC98/papers-html/AESGesture.html`.

[25] E. A. Lee and T. M. Parks. "Dataflow Process Networks". *Proceedings of the IEEE*, 83(5):773–799, 1995.

[26] S. Ignatchenko. "STL Implementations and Thread Safety". *C++ Report*, 10(7), 1998.

[27] J. Richter. *Advanced Windows*. Microsoft Press, Redmond, WA, 1995.

[28] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[29] D. Wessel, M. Wright, and S. A. Khan. "Preparation for Improvised Performance in Collaboration with a Khyal Singer". *International Computer Music Conference*, Ann Arbor, Michigan, 1998. International Computer Music Association.

[30] K. Makan. “Broken Thoughts”, 1999. Live performance at CNMAT/CCRMA/CARTAH Spring 1999 Concert Exchange. Center for New Music and Audio Technologies, Berkeley, CA.

[31] M. Wright. “Implementation and Performance Issues with OpenSound Control”. *International Computer Music Conference*, Ann Arbor, MI, 1998. ICMA. `http://www.cnmat.berkeley.edu/OpenSoundControl`.

[32] `http://www.scriptics.com`.

[33] M. Wright, S. Khoury, R. Wang, and D. Zicarelli. “Supporting the Sound Description Interchange Format in the Max/MSP Environment”. *ICMC*, Beijing, 1999.

[34] A. Stevens. “A C++Standard At Last”. *Dr. Dobb’s Journal*, 23(2):115–17, 130–1, 1998.

[35] A. Chaudhary. “Spin Cyclce / Control Freak”, 1999. Live performance at CNMAT/CCRMA Spring 1999 Concert Exchange. Center for New Music and Audio Technologies, Berkeley, CA, May 15 1999.

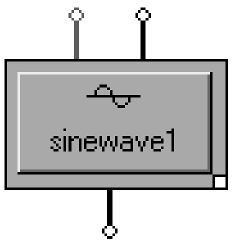


Figure 1: A sine wave transform

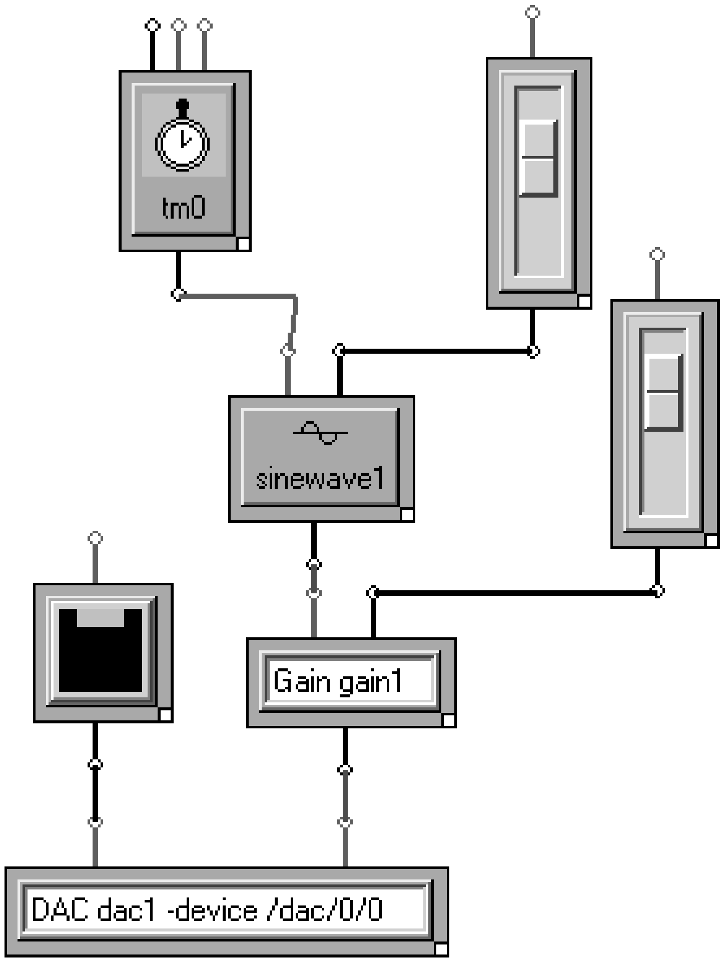


Figure 2: A patch that plays a pure tone with varying frequency and amplitude

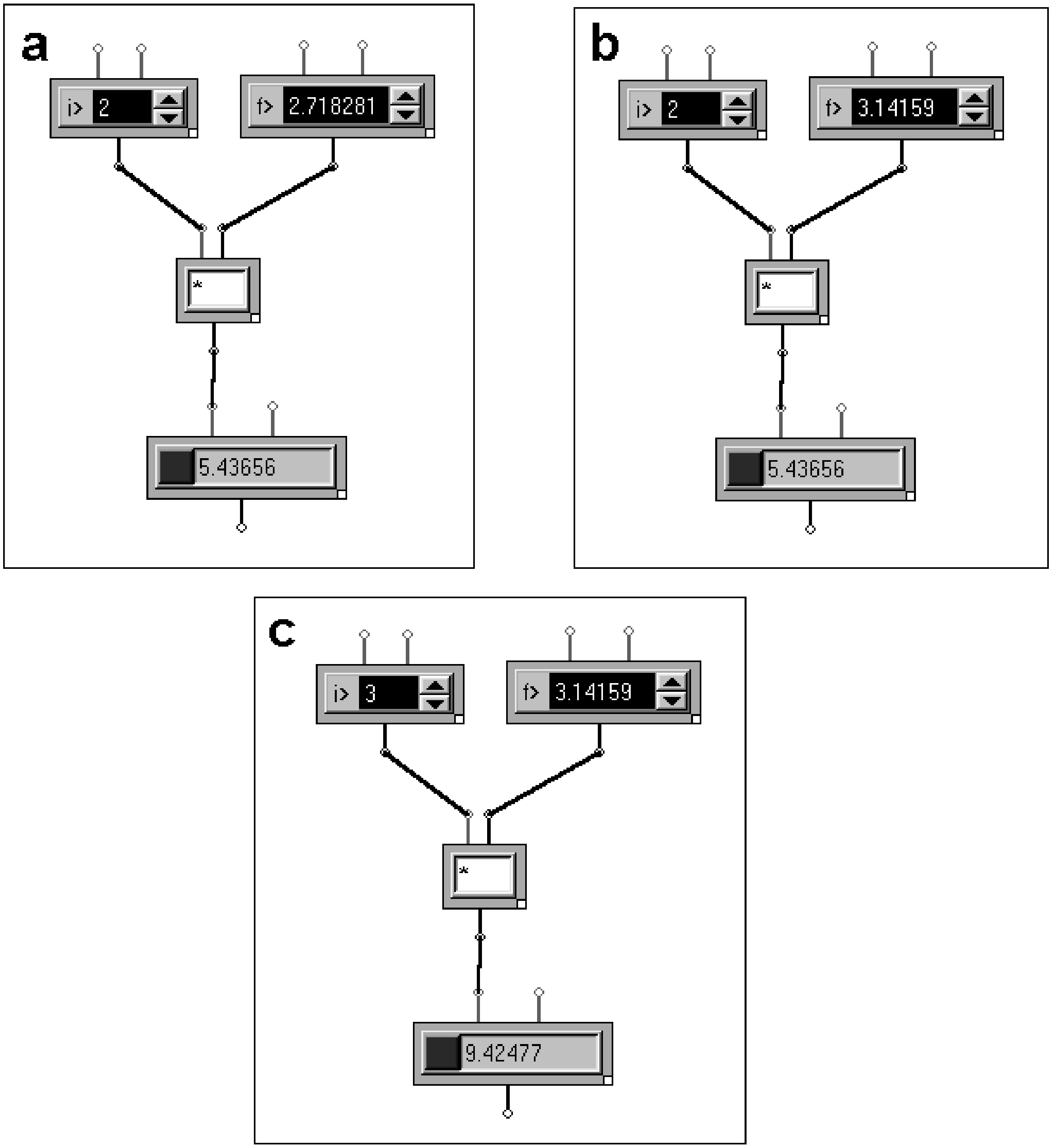


Figure 3: Active and passive inlets. a) A multiplication operator. b) The right inlet is passive, so changing its value does not cause any output. c) The left inlet is active, so the operator outputs the product of the two inlets.

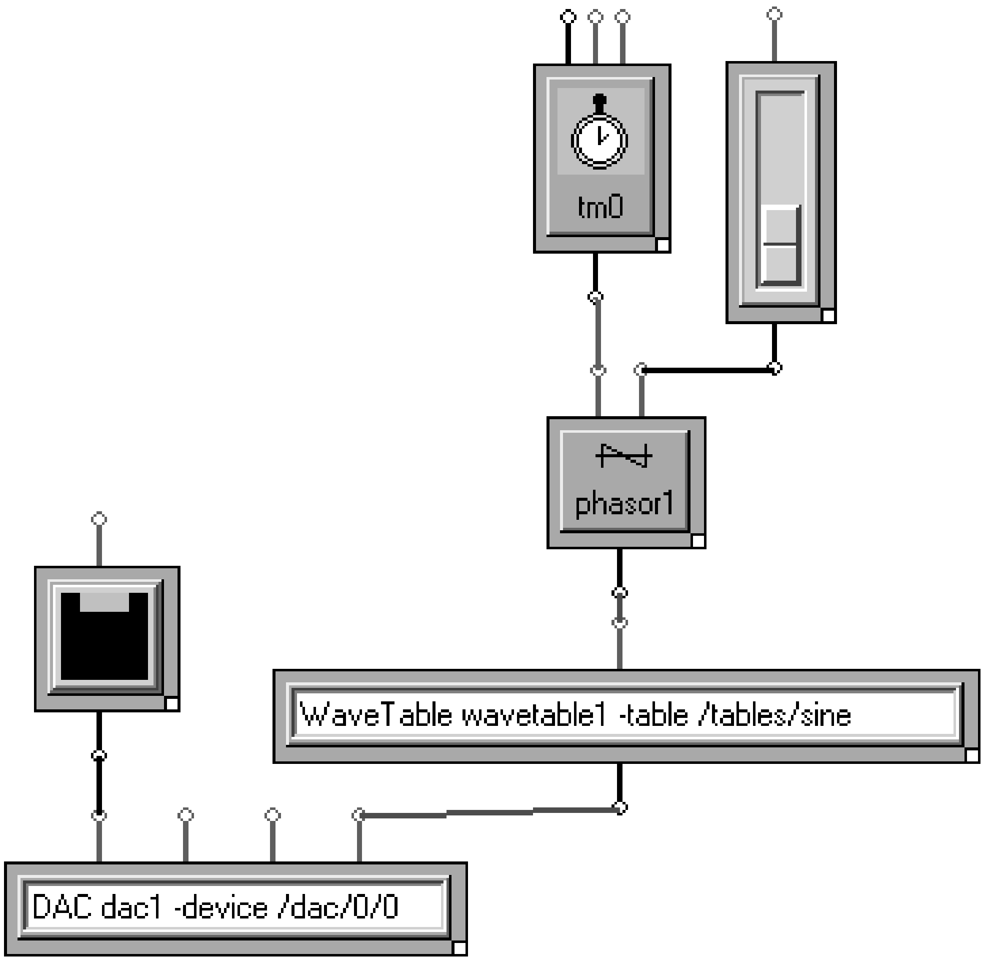
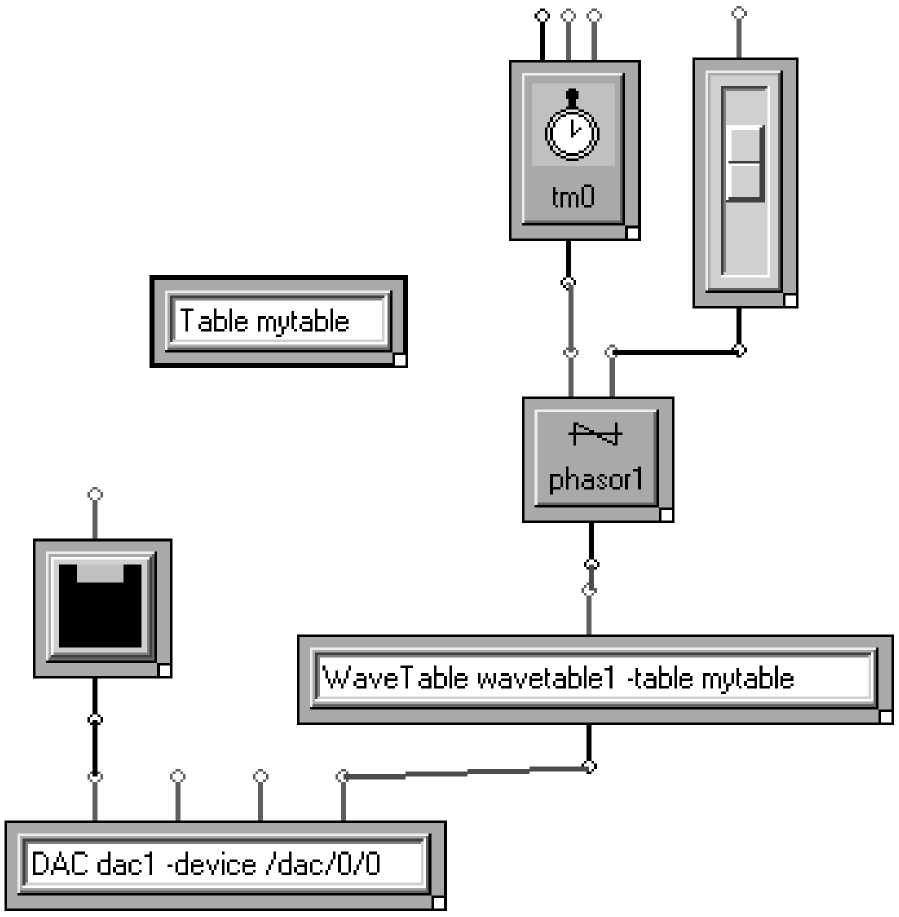

tm0
phasor1
WaveTable wavetable1 -table /tables/sine
DAC dac1 -device /dac/0/0

Figure 4: A patch that plays a pure tone using a wave table



tm0
Table mytable
phasor1
WaveTable wavetable1 -table mytable
DAC dac1 -device /dac/0/0

Figure 5: A patch that uses a custom wavetable stored locally in the patch. Tables can be loaded from files or hand-drawn.

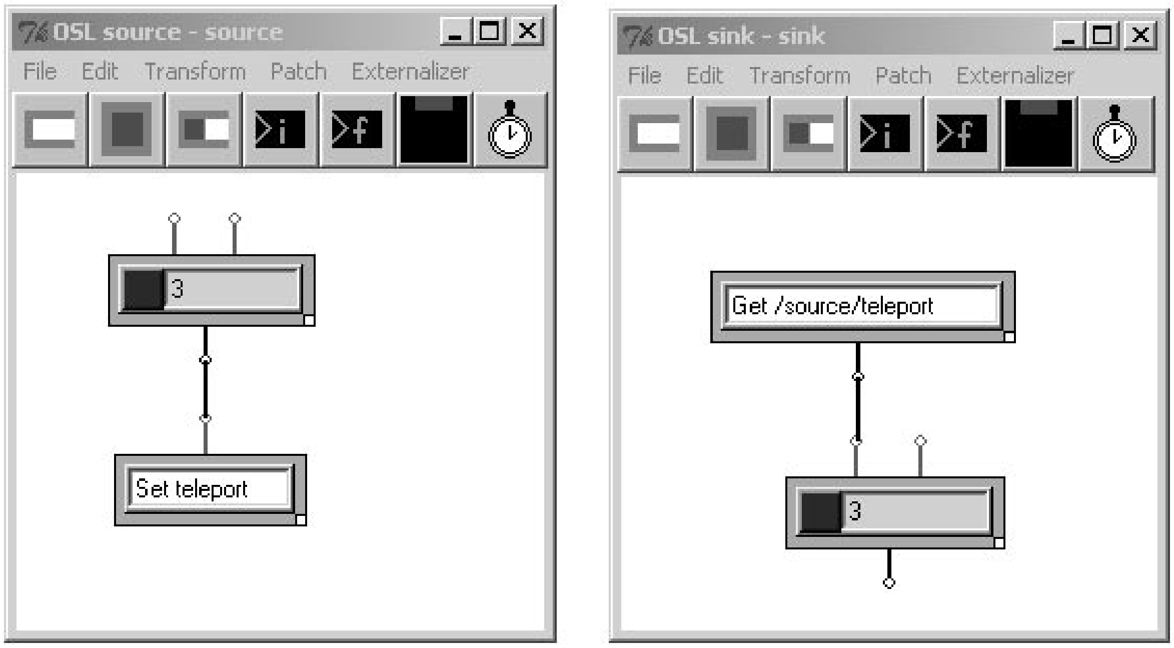

OSL source - source
File Edit Transform Patch Externalizer
3
Set teleport
OSL sink - sink
File Edit Transform Patch Externalizer
Get /source/teleport
3

Figure 6: Using Get and Set to transfer data between patches.

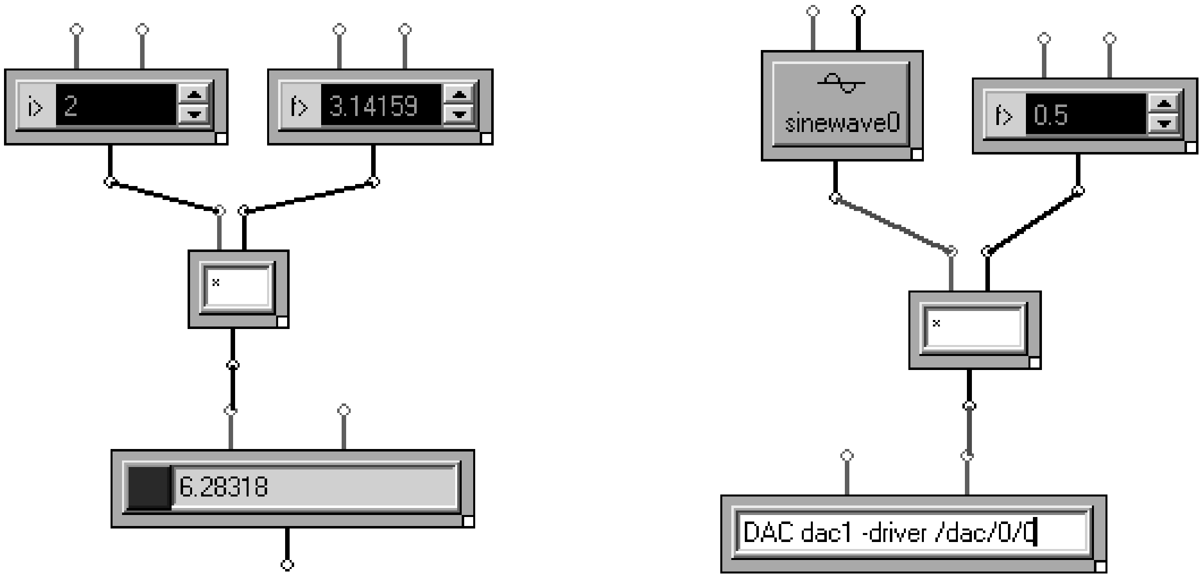


Figure 7: The multiplication operator on left returns the product of an integer and a floating-point number as a floating-point number. The operator on the right returns the product of samples and a floating-point number as samples.

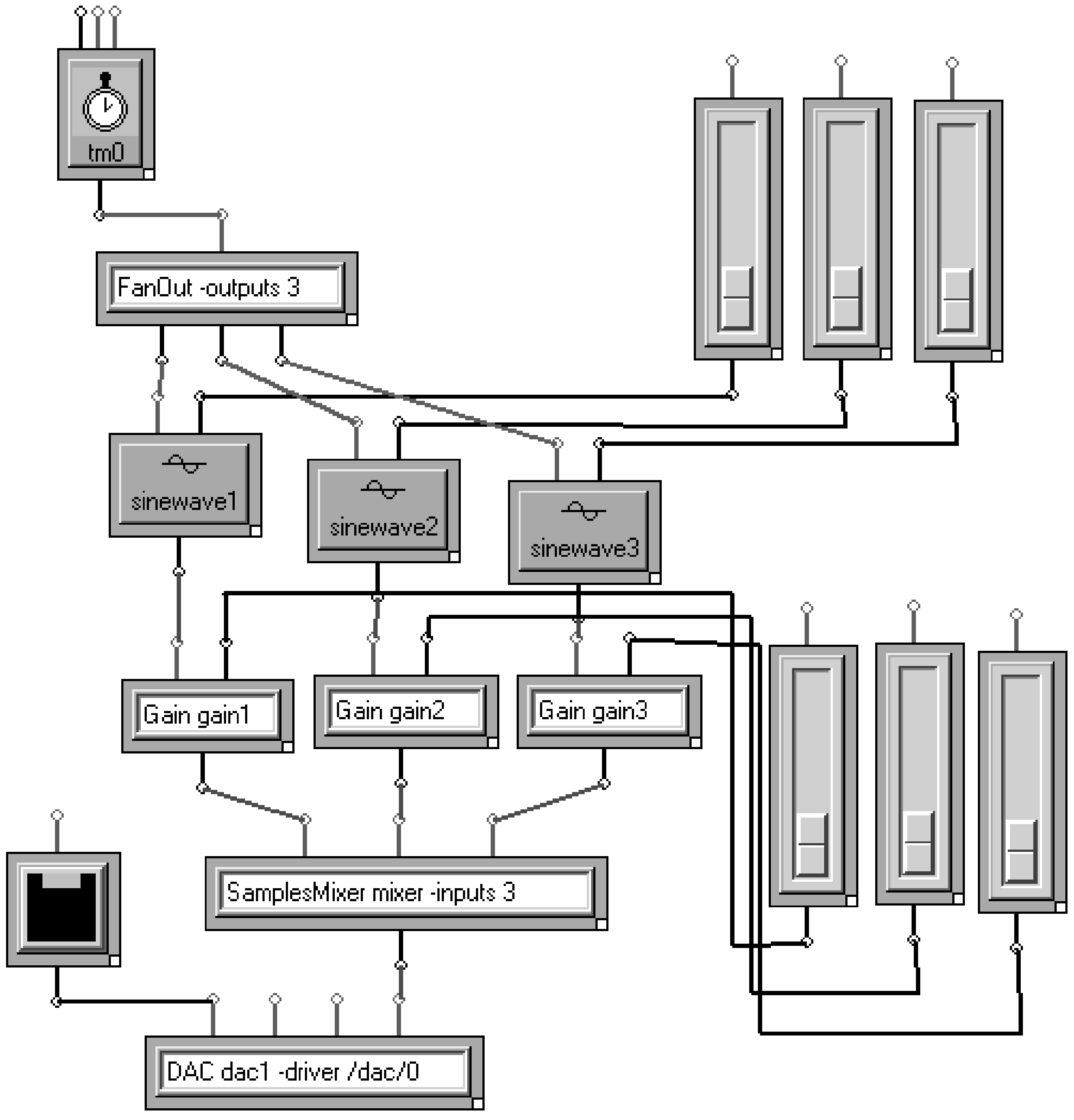


Figure 8: Mixing three sine-wave oscillators.

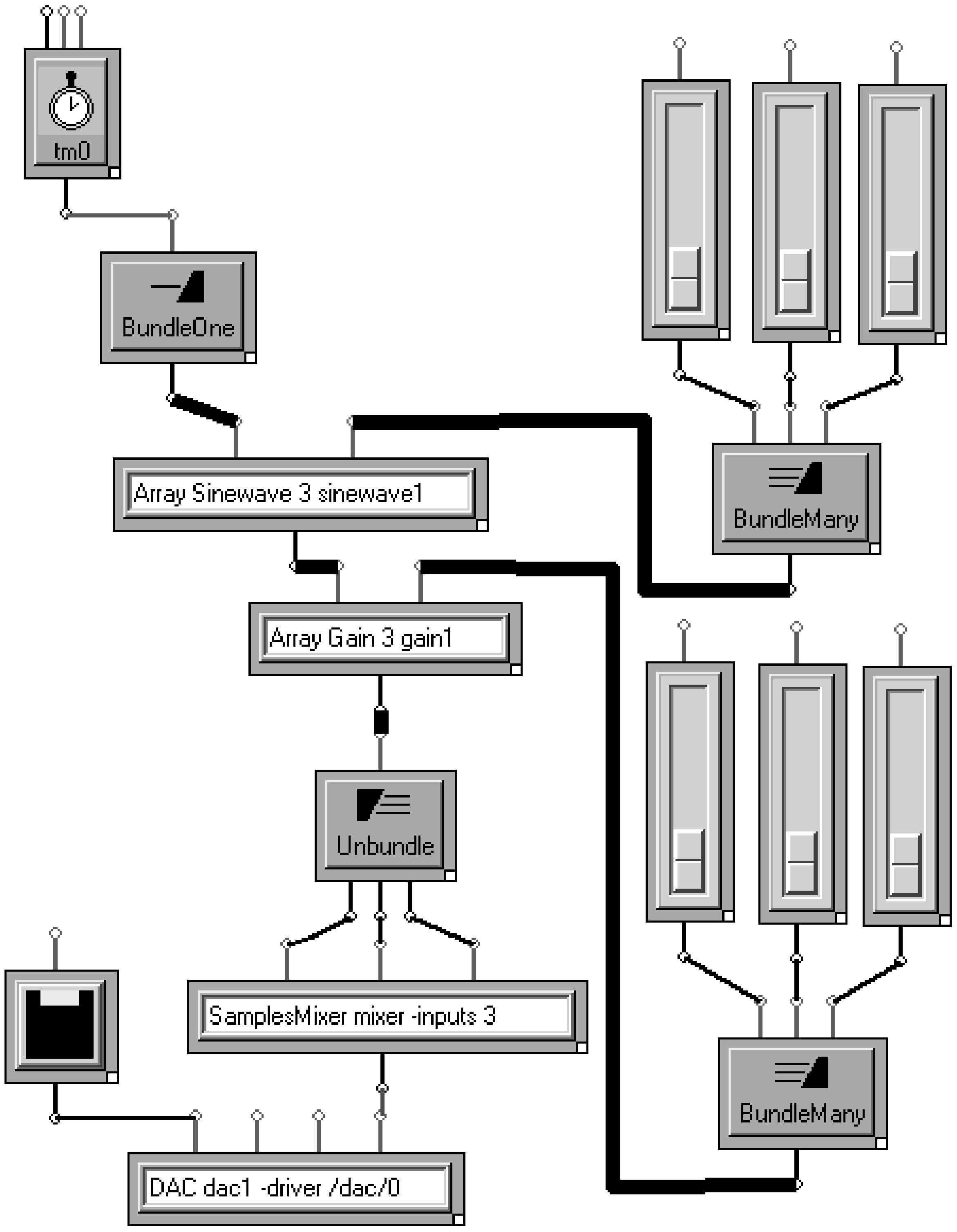
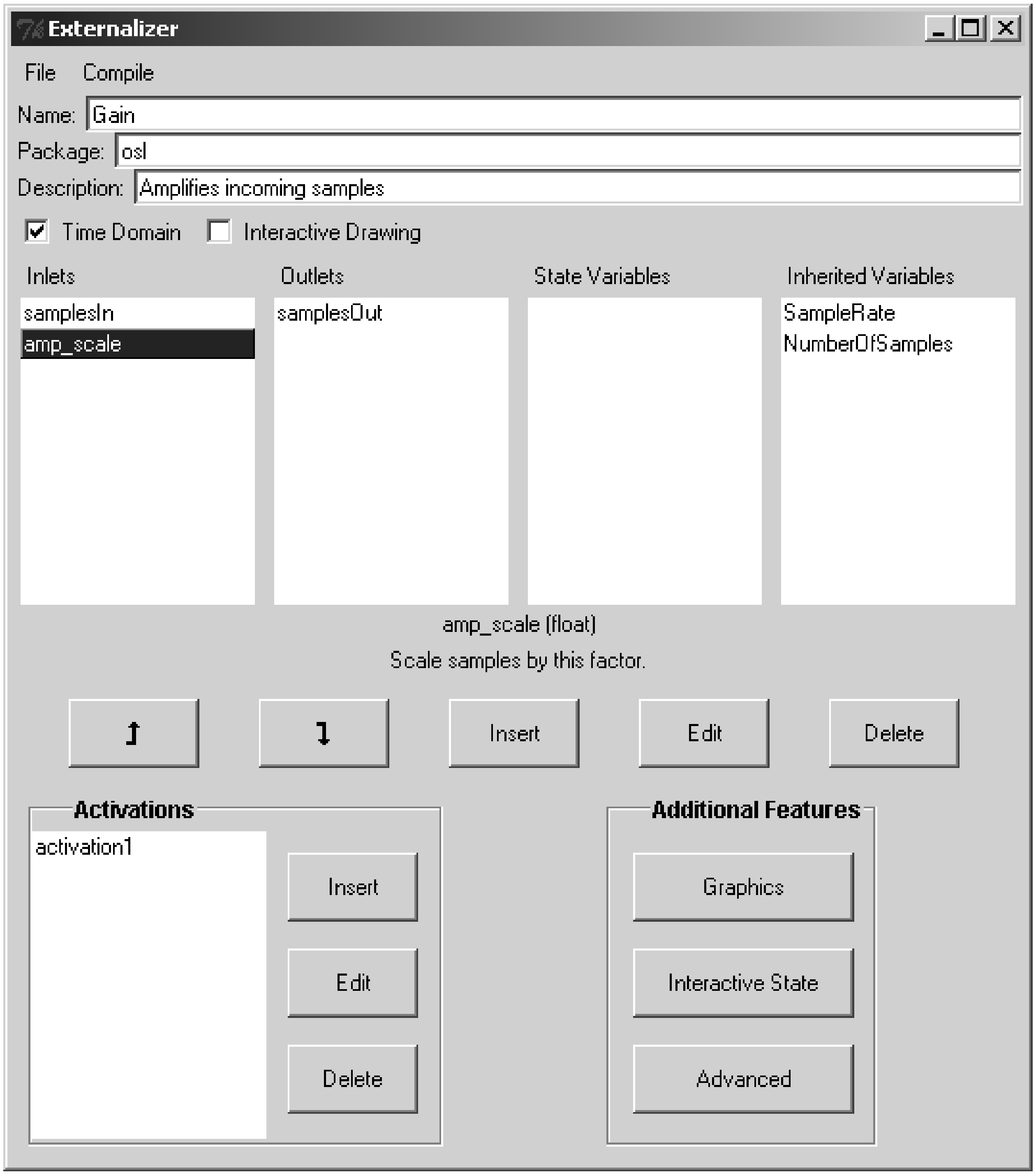


Figure 9: Same as figure 8, but using bundles and transform arrays.



Name: Gain
Package: osl
Description: Amplifies incoming samples

[✓] Time Domain [ ] Interactive Drawing

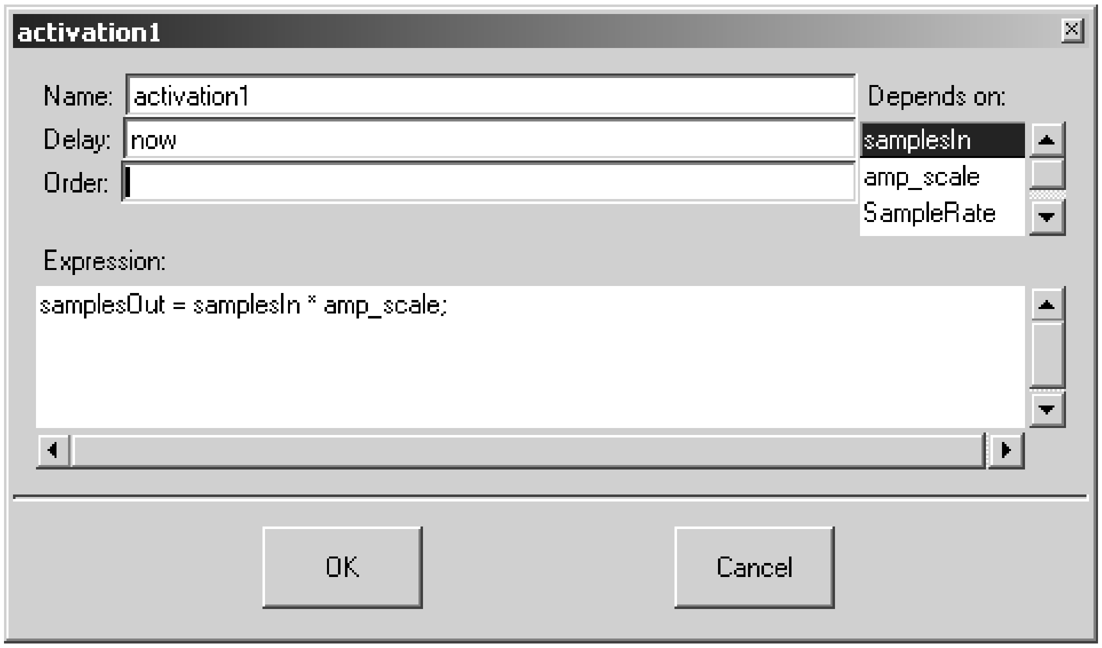
| Inlets | Outlets | State Variables | Inherited Variables |
|---|---|---|---|
| samplesIn | samplesOut | | SampleRate |
| amp_scale | | | NumberOfSamples |

amp_scale (float)

Scale samples by this factor.

Activations: activation1

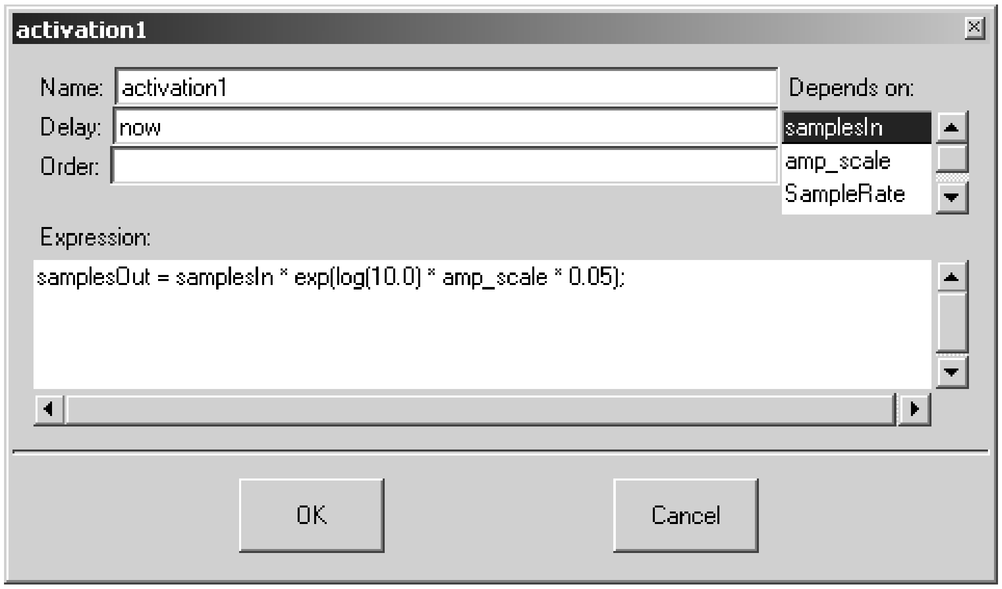
Figure 10: The Externalizer lists the variables and activation expressions for the Gain transform.



Name: activation1
Delay: now
Depends on: samplesIn, amp_scale, SampleRate

```
samplesOut = samplesIn * amp_scale;
```

Figure 11: The activation expression of Gain. Incoming samples are multiplied by amp_scale.



Name: activation1
Delay: now
Depends on: samplesIn, amp_scale, SampleRate

```
samplesOut = samplesIn * exp(log(10.0) * amp_scale * 0.05);
```

Figure 12: The activation expression is modified to measure amp_scale in decibels instead of a linear scale.

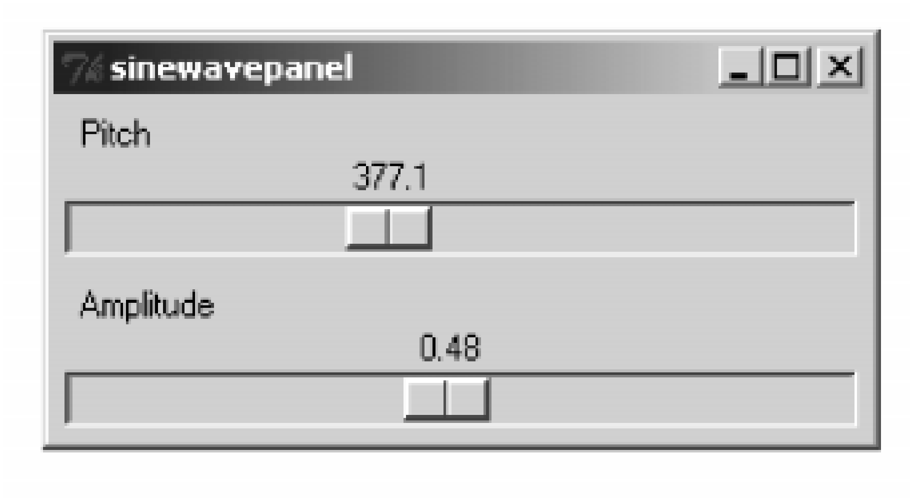


Figure 13: A customized interface for the simple oscillator patch in figure 2.

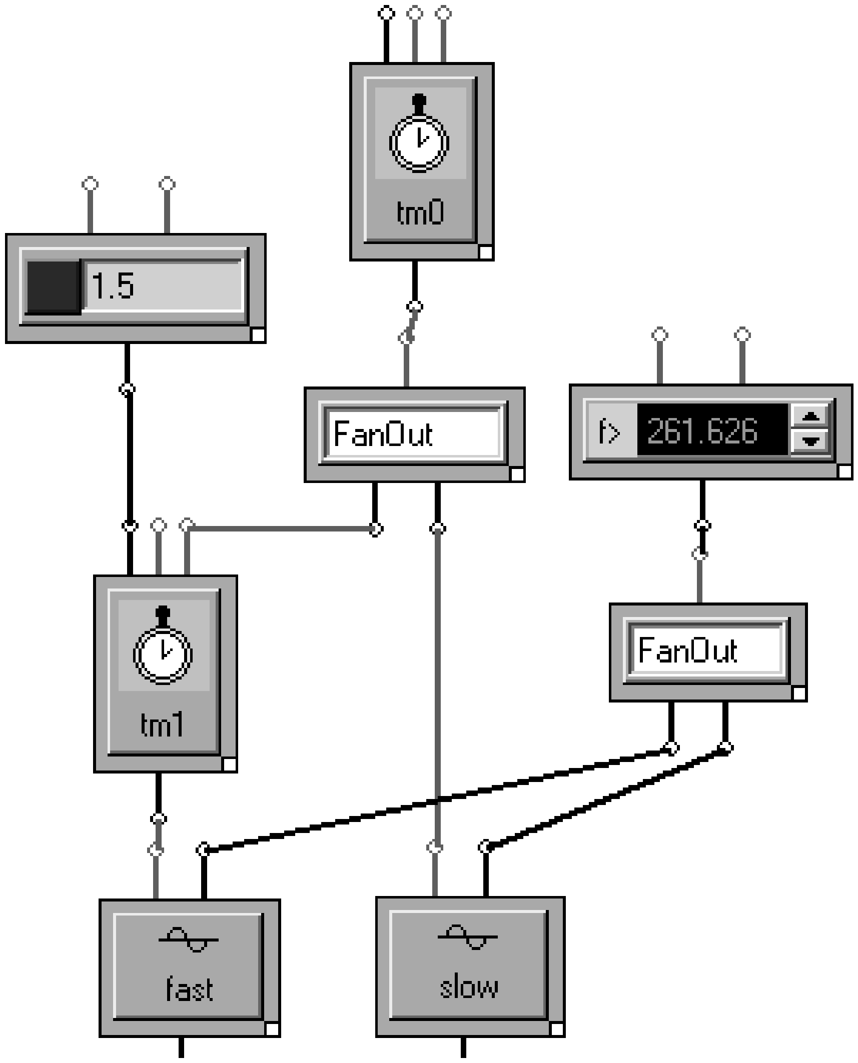


Figure 14: Fun with time machines. Both oscillators are set to the same frequency. However, the time machine driving fast is running 1.5 times faster than the the time machine driving slow. The oscillators will sound a perfect fifth apart.