An Open Architecture for Real-time Music Software
Amar Chaudhary, Adrian Freed and Matthew Wright
Center for New Music and Audio Technologies
University of California
Berkeley, CA 94720
{amar,adrian,matt}@cnmat.berkeley.edu

ABSTRACT
Open Sound World (OSW) is a scalable, extensible object-oriented language that allows sound designers and musicians to process sound in response to expressive real-time control. OSW allows users to develop at different levels, including visual patching, XML editing, scripting and high-level C++. Components called transforms are combined to form programs called patches. The set of included transforms can be extended using the “Externalizer,” a tool for writing high-level specifications of new transforms. The OSW real-time scheduler supports a uniform timing model for all components and symmetric multiprocessing.

INTRODUCTION
We introduce “Open Sound World” (OSW), a scalable, extensible object-oriented language that allows sound designers and musicians to process sound in response to expressive real-time control [Chaudhary, Freed et al. 1999].

Real-time software synthesis packages such as Max/MSP [Zicarelli 1998] and FTSjMax [Déchelle, Birghesi et al. 1998] 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 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 introduced hierarchical, user-definable data structures for components [Puckette 1996]. 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++, XML documents and scripting.

OSW is also highly dynamic and allows users to both edit transforms and manipulate performance controls simultaneously, run audio signals at several rates simultaneously and change patches or the basic configuration even while the audio is running. The real-time scheduler supports symmetric multiprocessing as well as multiple clocks and audio devices.

BASIC OSW FEATURES AND USE
OSW is a “dataflow programming language,” in which primitive components called transforms are connected together to form dataflow networks called patches. Patches are themselves transforms, which can be nested to form hierarchical structures for large projects. OSW includes a large set of standard transforms for basic event and signal processing, including filters, synthesizers, arithmetic operations on numbers and vectors, and functions for MIDI and the Sound Description Interchange Format (SDIF) [Wright, Chaudhary et al. 1999]. The set of available transforms can be easily extended to include more advanced operations. 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 as well.

OSW includes a visual environment for instantiating transforms and building patches, illustrated in Figure 1. Although it is quite similar to other real-time music systems such as Max/MSP, it includes several enhancements that are the product of user testing and usability evaluation, such as an extensible “dock” for frequently used transforms, a single pane window for simultaneous access to several patches in large projects, and editing functions (e.g., selecting or connecting transforms) based on freehand drawing.

XML AND SCRIPTING
OSW patches are represented using XML [DuCharme 1999], which can be directly edited using a text editor. Thus users who wish not to use the visual environment can still use OSW to develop their musical ideas. Additionally, users can embed scripted elements within patch files. Scripting affords advanced users dynamic patch-building options that would be difficult in the visual patching environment, such conditional instantiation or connection of transforms, or custom user interfaces. OSW currently supports Tcl/Tk scripting [Welch 1997].
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, a specialized toolkit and the original source code.

OSW includes 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. A transform is specified as a collection of inlets, outlets, state variables and activation expressions that a user can view or modify. A state variable is a public variable of a transform that can be queried or modified by other transforms in OSW. Inlets and outlets are special cases of state variables used in connections. An activation expression is a piece of C++ code that is executed when inlets or state variables are modified. It is specified by the variables that will trigger this activation, whether it should occur immediately or be delayed by a certain amount of time, and the code that should be executed. Table 1 contains the specification of Sinewave, a transform that implements a simple sinusoid oscillator:

![Figure 1. Editing a patch in the OSW visual environment. Transforms can be selected for editing by drawing a closed figure around them. The area to the left contains “docked” transforms.](image)
The activation expression looks like a continuous function of time. However, it is actually computing a sequence of samples from a discrete time variable, timeIn. The state variables NumberOfSamples and SampleRate are inherited from a more general class of time-domain transforms that manipulate time-domain samples.

An Externalizer transform specification is automatically converted to a C++ class which is then compiled into a dynamic library. The new library will be automatically loaded when the transform is first instantiated. Users can also specify new data types, which are converted to C++ struct definitions for use in transforms.

Externalizer specifications allow users to specify activation expressions using intuitive, familiar mathematical definitions instead of hand-optimized computer code. Efficiency is maintained by judicious application of standard C++ features in the underlying OSW base classes [Freed and Chaudhary 1998]. In addition to techniques such as composition closure, functors and operator overloading that are already well-known in the numeric community [Veldhuizen 1997], we exploit additional optimizations for functions of discrete time that are used extensively in signal-processing applications.

**CONTROL OF TIME IN OSW**

Many real-time computer-music systems employ two different notions of time: “physical time” that is associated with hardware clocks and low-level scheduling, and “virtual time” which can be scaled or otherwise controlled by the user [Dannenberg 1989]. OSW provides a unified user-configurable view of physical and virtual time that encompasses both system functions (such as scheduling and sample clocks) and user control.

All transforms that depend on time, such as the Sinewave transform in the previous section, a sample playback transform or a MIDI sequencer are automatically synchronized to a single clock (by default, the clock of an audio device being used). Users can gain more explicit control over time by optionally connecting such time-dependent components to *time machines*, special transforms that map physical time from clicks to virtual time, or scale virtual time output from other time machines. Time machines include parameters for scaling virtual time or resetting to a specified value (i.e., “go to this time”), thus allowing users to control the flow of time in the connected signal-processing components. The control functions can be arbitrarily complex, and include user input from MIDI or high-resolution gestural input devices.

**OSW SCHEDULING**

OSW uses a greedy scheduling algorithm that executes waiting activation expressions as soon as hardware to process them becomes available. This model exploits symmetric multiprocessors to handle several activation expressions as once, which happens in patches that have several branches or paths of execution [Chaudhary, Freed et al. 1999]. The scheduler cooperates with the timing model to provide reactive real-time performance, i.e.,

---

*Table 1. Externalizer specification for the Sinewave transform.*

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inlets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeIn</td>
<td>Time</td>
<td>440.0</td>
</tr>
<tr>
<td>Frequency</td>
<td>Float</td>
<td>44100.0</td>
</tr>
<tr>
<td>Outlet</td>
<td>SamplesOut</td>
<td>Samples</td>
</tr>
<tr>
<td>Outlet</td>
<td>SamplesOut</td>
<td>Samples</td>
</tr>
<tr>
<td>Inherited</td>
<td>SampleRate</td>
<td>float</td>
</tr>
<tr>
<td>Inherited</td>
<td>NumberOfSamples</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Activation Expression activation1, depends on timeIn

\[\text{samplesOut} = \sin(\text{TWOPI} \times \text{frequency} \times \text{timeIn});\]
10ms total latency between user control and audio output [Clarke 1999], and 1ms of jitter [Tsuzaki and Patterson 1997].

**DISCUSSION**

We are actively developing OSW in Intel-based PC’s running Windows 98/2000 and Linux operating systems, as well as SGI workstations running Irix. We use the GCC compiler for compiling the core system and Externalizer specifications under Linux and SGI, and Microsoft Visual C++ for Windows. The current version of OSW has been used successfully in live performance situations [Chaudhary 2000].

Addition information about OSW can be obtained at http://www.cnmat.berkeley.edu/OSW.

**ACKNOWLEDGEMENTS**

We gratefully acknowledge the NSF Graduate Research Fellowship Research Program for their support of this research. We would also like to thank David Wessel, director of CNMAT and Lawrence A. Rowe, director of the Berkeley Multimedia Research Center, for their continued support.

**REFERENCES**


