Metamodels and Design Patterns in CSL4

Stephen Travis Pope, Xavier Amatriain, Lance Putnam, Jorge Castellanos, and Ryan Avery
Center for Research in Electronic Art Technology (CREATE)
University of California, Santa Barbara
{stp,xavier}@create.ucsb.edu, {ljputnam,jcastellanos,ravery}@umail.ucsb.edu

Abstract

The task of building a description language for audio synthesis and processing consists of balancing a variety of conflicting demands and constraints such as easy learning curve, usability, flexibility, extensibility, and run-time performance. There are many alternatives as to what a modern language for describing signal processing patches should look like. This paper describes the object-oriented models and design patterns used in version 4 of the CREATE Signal Library (CSL), a full rewrite that included an effort to use concepts from the "4MS" metamodel for multimedia systems, and to integrate a set of design patterns for signal processing. We refer the reader to other publications for an introduction to CSL, and will concentrate on design and implementation choices in CSL4 that simplify the kernel classes, improve their performance, and ease their extension while using best-practice software engineering techniques.

1 Introduction

The literature of MusicN-style programming languages for building signal synthesis "instrument" programs and executing them using note list "score" files goes back over 50 years (Mathews 1969; Pope 1993), and is still being actively developed in modern languages such as SuperCollider (McCartney 2002), CommonLISPMusic (Schottstaedt 2000), Cmix (Lanski 1990), J-Syn (Burk 1998), Siren (Pope 2003), Kyma (Scaletti 1991), and CLAM (Amatriain and Arumi 2005).

The CREATE Signal Library (CSL, pron. "sizzle") (Pope and Ramakrishnan 2003) is a C++ class library for (audio) signal synthesis, processing, and analysis. It is similar to a MusicN-style language in that it provides the user with (1) a set of unit generators that implement most common audio synthesis and processing techniques, (2) a framework for constructing "instruments" that are signal processing graphs made up of these unit generators, and (3) supports executing "scores" with a scheduler, or responding to real-time control inputs (MIDI, OpenSoundControl, etc.) using an instrument library or orchestra.

In contrast to the traditional MusicN stand-alone sound compiler, CSL is packaged as a class library in a general-purpose programming language (C++). The simplest CSL program is a 5-line *main()* function in a simple C program, and it is intended that CSL can be used in several ways, including for the development of stand-alone interactive (MIDI- or OSC-driven) sound synthesis programs, serving as a plug-in library for other applications or plug-in hosts, or as a back-end DSP library for programs written in scripting languages. CSL is designed from the ground up to be used in distributed systems, where networks of CSL programs run as servers on a local-area network, streaming control commands (MIDI and OSC) and sample buffers (RTP) between them (Pope, Engberg, Holm, and Wolf 2001).

CSL has evolved through several re-writes since 1998; version 4 (see http://create.ucsb.edu/CSL) was implemented by graduate students at UCSB in the 2005-6 academic year.

1.1 The 4MS Metamodel

CSL's design is an instance of the MetaModel for Multi-Media Systems (4MS) (Amatriain 2005; Amatriain and Pope 2006). 4MS is a metamodel for multimedia systems that can be instantiated to describe any multimedia processing design, and that combines the advantages of the object-oriented (OO) paradigm with system engineering techniques and graphical models of computation. The 4MS metamodel is based on a classification of signal processing objects into two primary categories: *Processing* objects that operate on data and control, and *Processing Data* objects that passively hold media content. Data input to and output from Processing objects is done through *Ports*, and control data is handled through the *Control* mechanism.

1.2 Challenges and Goals

Since its first beeps eight years ago, CSL has been driven by the strongly conflicting design goals of *simplicity* (it has been used in teaching audio programming courses since day one), *flexibility* (students have wanted to build servers, plug-ins, and to interface CSL programs with various other APIs), *comprehensiveness* (supporting easy extension to any domain of digital audio signal analysis, synthesis, and processing), and *scalability* (enabling multi-server distributed processing and inter-server control and sample streaming).

For the version 4 rewrite, we were required by the applications we were building at the time to also address issues such as processing graphs with varying numbers of channels (and different spatial models) in the branches, modeling of spatial data control streams and spatialized sources, multi-rate and variable-rate control input and processing, and distributed sample stream incorporating multiple transport protocols and semi-reliable wireless I/O devices.

It is the challenge of facilitating this advanced level of development with the earlier constraint that the basic description language be as simple as a MusicN-style scripting language as possible that motivated the group in the work described here. In this paper, we discuss the design issues in DSP software, and present the motivations for the decisions behind CSL4's core model classes, relating them to the 4MS metamodel and common-practice OO design patterns.

1.3 The CSL Design Aesthetic

Due to the goals and constraints outlined above (especially with respect to learning curve and flexibility), and to the fact that this is the fourth re-write of CSL, we have striven for minimalism in both the design and the implementation of the system. Our goal is Smalltalk-like simplicity, uniformity, and comprehensiveness, so we have avoided some design options that might have made the system faster or more flexible for advanced users. A secondary goal (also learned from Smalltalk) is to have all of the system's operational mechanisms exposed to the user and expressed within the same model.

2 Data and Processing Object Models

A language or API for audio will need to define the basic data structures for sampled sound and for many kinds of operations on sound. The choice of the model of sampled sound buffers or streams will determine the nature of the language's operational semantics or the flavor of the API that manipulates these sound objects. The standard "procedural-object" model for audio signal processing evolved form MusicN languages, where flow charts are often used to visualize signal processing graphs (implying a data-centric object-oriented model), but description languages often have a distinct procedural flavor (the "unit generator as subroutine" perspective).

The kernel of CSL is a group of abstract classes that map quite directly onto the 4MS metamodel of Processing, Processing Data, and signal or control I/O Ports. In 4MS, applications consist of networks of processing objects interconnected via control and signal flows. 4MS also introduces a categorization of multimedia software components; CSL's framework hierarchies fall squarely in the Infrastructure and Platform Abstraction groups, with out-board components for Visualization and Serialization tasks.

The impact of the metamodel entails not just the class hierarchies, but also the object life-cycles and signal processing network composite models. In addition to the metamodel's architecture, CSL uses a variety of common design patterns such as Observer, Composite, Adaptor, Singleton, Factory, Template Method, Visitor, Builder, Proxy, Faade, Decorator, Strategy, Interpreter, Chain of Responsibility, and Command. These will be the topic of this paper.

2.1 Model Paradigm and System Architecture

The initial requirements prescribe many aspects of the overall system architecture (language vs. API vs. app.), and thus determine much about the basic modeling paradigm used in the data and processing classes (Arumi, Garcia, and Amatriain 2006). How (and by whom) a language or API is to be used, and into what environment it is to be embedded, are the design context for synthesis language designers.

Using the CSL classes means writing C++ code that creates and configures one or more instances of the subclasses of *UnitGenerator*, and connects them to some control or scheduling functionality, and to an output object. The signal processing unit generator graph built in a CSL program is usually activated by regular call-backs from its output object, which propagate up the graph's tree according to the prescribed signal flow.

2.2 Processing Data: Buffers

The requirements for data object models in a digital audio system specify that we provide some object to represent sampled sound data buffers with a give number of frames and number of channels, and some place to store sample data (which could be as simple as a float**). In addition, we would like the model to support a basic life-cycle (so that we have the option of implementing buffer and storage management). According to the 4MS metamodel, Processing Data objects offer a homogeneous interface to media data, and support for meta object facilities such as reflection and serialization.

The concrete C++ class *csl::Buffer* is CSL's model of multi-channel sample frame data storage; its instances are passed between generators and IO objects. Buffer is a "record" class (or a struct) in that its members are all public and it has no accessor functions or algorithmic methods. It handles data buffer allocation and channel maps, and has Boolean members that are used in its life-cycle implementation to determine what its pointer state is. The actual storage vector is usually of type *SampleBufferVector* – a vector of (sample *) buffers, typically floats – and the buffer also holds a timestamp and sequence number. Buffer's design reflects the metamodel's ProcessingData class, and it can be extended so that Buffers can act as Observables (AKA Models or Subjects). Buffers are used by UnitGenerators and Ports via the Visitor design pattern.

None of the above discussion limits Buffers to holding only audio sample data, or fixes what the "channels" of data in a Buffer correspond to; in fact Buffers can be used as control data caches, for 3-D positional data, or as general-purpose N-by-M matrices as in SDIF (Wright 1999).

It is important to note that the buffer object knows its (expected) number of channels and frames-per-buffer. This is used in CSL to tell a unit generator what we expect it to do, since the argument of the *nextBuffer* message is a buffer configured by the generators output "client."

2.3 Processing: Unit Generator Classes

The notion of unit generator as a subroutine comes from the earliest MusicV documents (Mathews 1969), which included an extended discussion of different mechanisms for block-oriented sample processing. Given a model for the sample buffer object or stream paradigm, Processing classes are defined to perform operations on data under the guidance of control values; these may be driven by a scheduler, by their own threads, or may be passively executed by the streams themselves, according to the operational semantics of the language/API orientation. In 4MS, Processing objects encapsulate a method or algorithm; they include support for synchronous data processing and asynchronous event-driven control as well as a configuration mechanism and an explicit life cycle model.

The core of CSL's functionality is implemented in the class hierarchy derived from class *csl::UnitGenerator*, akin to the metamodel's Processing class. Instances of UnitGenerator subclasses have members for their sample rate and number of channels, and know their (0, 1, or several) outputs. Most concrete UnitGenerator subclasses also inherit from some subclass of *csl::Controllable*, which adds the notion of control or data input ports (see Figure 1 and the section on Ports below).

The main UnitGenerator behavior is the *nextBuffer* method, which is overridden in the subclasses to fill or alter the buffer object passed as the argument. This method normally polls the unit generator's control or signal input ports (possibly sending them the *nextBuffer* message), and then uses a sample

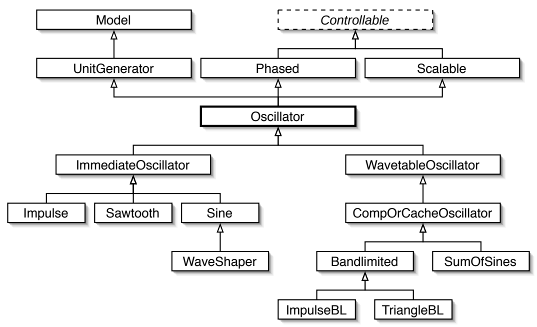

Figure 1: The (partial) class hierarchy of CSL Oscillator UnitGenerators showing multiple inheritance from UnitGenerator, Phased, and Scalable (the last 2 of which are both subclasses of Controllable). The concrete subclasses each implement (at least) a constructor and a *nextBuffer* method.

computation loop to write data into the buffer object's storage vector. Since a buffer knows the number of sample frames it holds (its *X* extent), the number of channels it supports (its *Y* extent), and both has a time-stamp and a sequence number, one can easily build CSL graphs where different components or sub-graphs run at different frame or block rates, numbers of channels, or with different amounts of delay or latency.

If more than 1 output port is connected, UnitGenerators automatically handle fan-out–synchronous (as in loops in a graph) or asynchronous (as in separate call-back threads or observers)–with differing buffer sizes or callback rates. UnitGenerator inherits from Model, meaning that *nextBuffer* methods in subclasses are required to send themselves the message *changed(aDataBuffer)* so that dependent objects (like signal views) get a notification when their models compute samples. This mechanism can also be used for signal flow, or for handling multiple output streams in a graph (e.g., writing an intermediate control value or "direct out" to a sound file).

Figure 1 shows a subtree of the CSL UnitGenerator framework centered on the class *csl::Oscillator*; the multiple superclasses provide processing template methods and observability (*UnitGenerator*), and special interpretation of several of the control ports (*Scalable* and *Phased*). The Oscillator subclasses each implement *nextBuffer* themselves for their primary behavioral refinement (i.e., how each subclass oscillates differently).

2.4 Ports and Control

In simple MusicN languages, connections between unit generators are handled by variables that are buffers or signals. In a more comprehensive model, a connection port or outlet is generally reified to allow unit generators and patch graphs

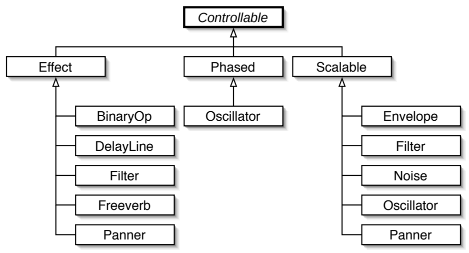

Figure 2: The csl::Controllable hierarchy with its families of Effects, Phaseds, and Scalables; classes like Panner are both Effects and Scalables, while Oscillator is both Phased and Scalable.

themselves to be more easily manipulated.

A *Port* is used to represent constant, control-rate or audio signal inputs and outputs to unit generators; the *Controllable* mix-in classes (see below) add naming and semantics to the multiple ports of a unit generator. Each Port holds either a UnitGenerator and its buffer, or a single floating-point value. The *nextValue()* message is used to get the dynamic or static value. The *nextValue()* method has no branching or complex logic and can be compiled in-line for efficiency (since it will be called several times per sample in a complex UnitGenerator).

UnitGenerators represent their inputs and outputs as named maps (or multimaps) of port objects, and these can be plugged and un-plugged at will (within reason). Port uses a mix of the well-known Adaptor and Proxy design patterns for all data values, and is used for the Chain-of-command pattern among UnitGenerators. Holding ports in named maps in UnitGenerators, provides us with good object reflection properties in CSL graphs.

Asynchronous control inputs to CSL patches can also be handled through Ports, and there are several special subclasses of class *Port* to handle interpolating or lag-timed controls. These are inserted into graphs via the setter functions of CSL instrument classes (see below), and handle the de-zippering of stepped dynamic controls such as low-rate sampled continuous MIDI or OSC control value messages.

2.5 Mix-ins: Controllable, Scalable, Phased

The final component of the model core is the mechanism and policies through which ports are actually assigned and used within unit generators. Many of the most important trade-offs in audio language design are to be found among the alternative designs to port maps, unit generator inheritance schemes, and multiple inheritance and sharing of behavior among unit generators.

Class *csl::Controllable* is the base of the collection of classes that add control or signal inputs to UnitGenerators. (We use Controllable as a virtual superclass so that we can mix it in more than once.) Controllable defines the map of port objects, and manages the naming and processing flow for dynamic inputs. A typical UnitGenerator will have several input and/or control ports, e.g., for frequency, scale, and offset in the case of an oscillator that supports AM and FM. The *pullInput* message is used within a UnitGenerator's *nextBuffer* method to send *nextBuffer* to a given port.

Controllable has three abstract subclasses that are widely reused by refinement. Class *csl::Scalable* defines scale and offset control ports (which may be constant or dynamic) as special input map keys; most concrete unit generators inherit this as well as UnitGenerator (akin to SuperCollider's optional *mul* and *add* unit generator constructor arguments (McCartney 2002)). The class *csl::Effect* assigns one or more of a unit generator's ports the special semantics of signal inputs, as in signal filters or panners. *csl::Phased* adds a default interpretation of a phase accumulator cache and a frequency input port.

Although they are used as mix-ins (i.e., they are used in multiple inheritance and don't provide virtual methods that are refined in their subclasses), the Controllable classes implement a version of the Policy and/or Decorator design patterns, and provide template methods as C++ macros; for example, Scalable defines macros to declare, load, and update the scale/offset control ports in the *nextBuffer* methods of their subclasses.

Figure 2 shows a subtree of the class *csl::Controllable*, with several of its children multiply inheriting from (e.g.,) Scalable and Effect. Note that several of the leaves constitute their own rich class hierarchies, as in Oscillators, Filters, or Envelopes.

2.6 IO and Scopes

We have so far ignored the activation mechanism for graphs, but it is obviously a major determinant of the framework that will support processing in any design. As in the above sections, our design is based on simplicity and visibility.

In the CSL model, graphs are activated (triggered or driven) by an IO object, which is normally an interface to a sound output driver that receives call-backs from the operating system at a regular rate (the frame rate divided by the output buffer size), and forwards them to the root of its DSP graph, sending that UnitGenerator the *nextBuffer* message. Other IO classes are available that write (control or audio) samples to sound files, or send their data packets to a network socket. Related to this are classes that implement graphical output such as oscilloscopes or control-rate monitors; to date, we have im-

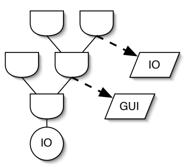

Figure 3: CSL processing graphs are activated by IO objects. Other IO objects or GUI displays then use the Observer/Observable relationship (shown as dotted arrows in the Figure) to attach themselves to any unit generator in the graph.

plemented CSL data views using both the Qt and OpenGL graphics APIs.

In CSL4, we separate the roles of trigger and data sink, using the Observer pattern to allow us to attach outputs or monitors to any unit generator in a graph as shown in Figure 3. Each CSL graph is expected to have exactly one trigger IO, but that may in fact be a *csl::NullIO* object that uses a timer thread to schedule graph activation and discards the IO. In that case, other IO objects are connected to the graph as observers of its unit generators.

2.7 Positioned Sound and Spatial Sources

The Spatial audio engine in CSL follows a layered API complying with the design goals of simplicity and flexibility. The core of the engine consists of Spatial Sources and Processors (UnitGenerators). Panners are among the most important type of these ”Processors”. Subsequently, higher API layers take these processors and sound sources providing a simpler, more intuitive interface. This layering can be interpreted as a direct implementation of the Faade pattern.

A Spatial Source is a UnitGenerator that adds spatial information to a UnitGenerator (Sound Object). Currently a Spatial Source positions a sound in a 3D space. In the future, it will also handle source directivity, object radiation, and possibly size and shape. Spatial Sources are implemented as Decorators, which wrap around UnitGenerators, adding the spatial information. This information can be later used by a Spatial processor that make use of the position of the sound.

Class *csl::Panner* is the base of a collection of classes whose main objective is to position input signals in a 3D space. A Panner is a UnitGenerator that can handle multiple input spatial sources producing a multichannel buffer as output. A simple way to think of a panner is as an N-input to M-output mixer. Example panners are VBAP (Vector Base Amplitude Panner), AmbisonicPanner, BinauralPanner, or special cases like a 5.1 panner or a stereo panner. The use of *csl::Panner* as a superclass is to allow for dynamic choice of the panner in use.

One layer above, the Spatializer hides the low-level engine, and presents a simple interface. Spatializers manage the graph for the different components needed, like a Panner and Distance Cues. Spatializers implement the Strategy Design Pattern, where the Panners are the strategies. A Spatializer can then choose at runtime what panner to use.

2.8 Analysis and Feature Extraction

Any general-purpose API or language for sound signal processing must also enable and support flexible signal analysis and metadata feature extraction functionality. In the model domain, this means the addition of windowed signal features in the time and frequency domains, and higher-level derived features such as spectral tracks and their statistics.

In our case, the FASTLab Music Analysis Kernel (FMAK) (Pope, Holm, and Kouznetsov 2004) is a sibling of CSL and uses many of the same models, with an *Analyzer* class that closely resembles CSL's UnitGenerator.

2.9 Putting it all Together

A unit generator class can be described in terms of its *nextBuffer* method, which has the general form shown in the pseudo-code below.

```
void nextBuffer (Buffer& outBuffer)
get local pointers to output arrays
    and to controllable ports
    (scale, offset, phase-inc, ...)
set up sample loop vars
  (send nextBuffer() to ports)
sample loop
    calculate a sample
    store sample to output buffer
    update input and control values
      (e.g., (send nextValue() to ports)
update output pointers
update stored members (e.g., inst. phase)
broadcast change message
  (Observers get notified)
```

In this method, the unit generator uses its prescribed input and control ports to get buffers of data, and then to get pointers to that data for use in its sample computation loop. The messages *void nextBuffer()* and *sample * nextValue()* provide the glue between unit generators, mediated by ports.

It is left up to the CSL user whether to package a CSL program in a single *main()* function or to write a new instrument class that encapsulates a DSP graph and a set of accessors in its constructor. Users can also write their own unit generator classes, providing a *nextBuffer()* method that applies some semantics to one or more named control and/or input ports.

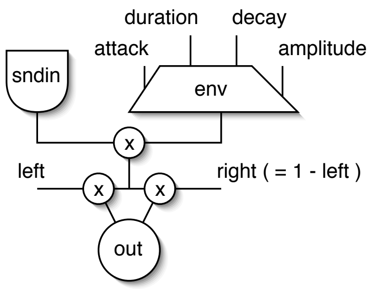

Figure 4: Sound file player instrument graph from the canonical software sound synthesis paper; a sample reader/player object is scaled by an envelope and panner to a stereo output. In CSL (and other modern languages) the graph would use four unit generators.

3 Building and Using Graphs

The discussion above concentrated on the core data and processing objects; we should now address the issues of what mechanisms are to be used for patch construction, management, optimization, and scheduling. This will be determined largely by the model's unit generator properties and interconnections. Along with this, there will be requirements as to how we handle modules and libraries, and how DSP graphs are connected to real-time control and triggering.

To use CSL, one programs in C++ or a scripting language (or uses a GUI) to connect UnitGenerator ports into a (probably, though not necessarily, acyclic) directed graph, which can be called an instrument or patch. Multiple patches in a thread can be connected to a mixer for output. According to the metamodel, CSL UnitGenerator/Port graphs use the Chain of Responsibility design pattern as the activation framework, i.e., the graph's root object starts its *nextBuffer* method and sends the same request to its input and control ports as specified by its configuration.

3.1 An Example: A SoundFile Player

As an example of CSL, let's take the 3rd example from the CMJ synthesis languages comparison paper (Pope 1993), a soundfile player with a panner control as shown in Figure 4. In CSL, this can be written as a compact *main()* function in a procedural style that consists of calls to the constructor methods of 4 objects: the envelope, the sound file player, the stereo panner, and the output. The default constructors to the unit generator classes allow the inputs to be plugged in at instantiation time, or one can patch generators together with setter methods such as *setScale* or *addInput*. Our main function would then be,

```
     // amplitude env = std ADSR
ADSR amplEnv(1, 0.1, 0.1, 0.4, 0.3);
    // sample file player
SoundFile filePlayer("name.snd", amplEnv);
    // stereo panner/processor
Panner thePanner(filePlayer, 0.2);
    // PortAudio IO object
PAIO theIO(thePanner);
```

The constructor calls in this example connect the unit generators into the minimal DSP patch for the stereo panned sound file player. To play it now, we need (in main) to trigger the envelope and start the output object's call-back thread.

To extend this example some, we could (1) add another output to write the output to a file while playing; (2) add a control-rate oscilloscope to the envelope and a signal-rate oscilloscope to the output; (3) create an interface instrument object (see below) to encapsulate our player for MIDI and OSC; we'll discuss these extensions in the sections below.

CSL supports several mechanisms for processing with multiple output objects in one or several threads. The simplest technique is to use the Observer pattern (or dependency mechanism) to add an output to a unit generator independent of the root IO object of the graph in which the unit generator gets triggered. We can easily instantiate another object of some IO subclass and say,

```
    // create a file writer
FileIO fileIO("sndFileName");
    // add it as a panner dependent
thePanner.attachObserver(fileIO);
```

Because the panner object sends the *changed* message at the end of its nextBuffer method, the second output will get a chance to copy out the final buffer at the end of each callback.

For graphics output, interfaces between CSL and the Qt GUI library and an OpenGL-based display and interaction framework (http://glv.mat.ucsb.edu) can be used. To add Qt widgets that display control and audio signals, one encapsulates the contructor statements we wrote into the constructor of a subclass of QMainWindow. Within a Qt or GLV application, one deals with instrument collections that have GUI widgets as observers. The GUI code interfaces to the DSP graph via the instrument's named map of unit generators, and GUI buttons can send instrument messages.

3.2 Instruments and Networks

There are several utility classes to make it easier to manage DSP graphs. An *Instrument* object has a DSP graph, a set of reflective accessors, and a list of envelopes. The DSP graph is the instrument patch, the accessors describe what the control parameters of the patch are (i.e., their names, types, and setter functions), and the envelope list is the collection of envelopes that need to be triggered to start a new note. With

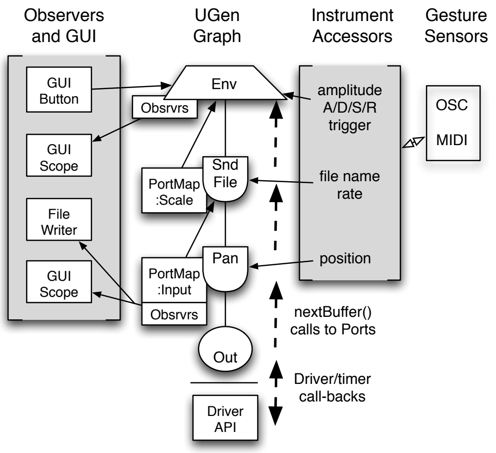

Figure 5: The detailed CSL graph for the sound file player shows the unit generators at the center, the GUI I/O connected via the observer dependency pattern on the left, and the Instrument object's accessors on the right, connected to OSC and MIDI I/O.

this abstraction of a graph, one can easily construct code that automatically creates the mapping glue to control CSL programs from OSC (Freed and Wright 1997), CORBA, XML, or MIDI. The Instrument framework represents a reflective composite metamodel with a Reflective Adaptor or Wrapper design pattern, and is used by tools (such as the OSC and GUI interface classes) in the Builder or Interpreter patterns.

The next logical step in our sound file player example would be to connect the player to external triggers and control. We need to declare an instrument class that will encapsulate our player for MIDI and OSC control. The instrument holds onto the root of the graph, and adds these reflective accessors. In its constructor method, we create a named map of the graph's unit generators, and a list of Accessors that map OSC command names or MIDI control values to flags sent to a general setter function

```
    // add ugens to the map for
    // external reference
mUGens["Pan"] = thePanner;
mUGens["Play"] = fileplayer;
    // add accessors with names and
    // setter flags
mAccessors.push(new Accessor("am", setAmplF));
mAccessors.push(new Accessor("po", setPosF));
```

One could add setter functions to the instrument class to map OSC/MIDI messages for the envelope parameters, stereo position, and file name, and for triggering the envelope, making a flexible enveloped stereo sample player with interactive control. The detailed view of our sound file player instrument

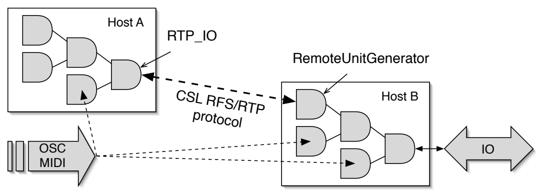

Figure 6: A multi-host CSL graph with UnitGenerator graphs running on 2 computers and a sample streaming mechanism using the RTP protocol between them. Control comes in to either server from MIDI or Open Sound Control messages.

is shown in Figure 6, which shows the four core objects (in the center) with their inherited members (port maps and observer lists) connected to one another and to outside objects.

3.3 OSC and MIDI Control

The design decisions implicit in the object model will determine how DSP networks are run and controlled. In CSL servers, we can use Instrument classes described above for OSC or for MIDI interfaces with code generators to fill in the glue code for mapping. Using CSL/OSC instrument libraries controlled by CSL GestureSensor control mappers is now a simple matter.

3.4 Distributed Graphs

Figure 6 shows a distributed instrument where CSL processes are running on separate host computers; both server (left) and client (right) receive control inputs from MIDI or Open Sound Control, and use CSL *RTP_IO* (server) and *RemoteUnitGenerator* (client) objects to stream samples over a socket connection using the RTP network protocol (Schulzrinne, Casner, Frederick, and Jacobson 1996). Remote sample and control streaming interfaces have been built for a number of protocols ranging from low-level UDP sockets to higher-level managed and monitored RTP connections.

In a recent student project, the CSL framework was used to implement a system for wireless audio transmission, storage, and web hosting. Referring to Figure 6, a connection between a wireless recording client (Host A) and a wired server (Host B) was established using the CSL RemoteUnitGenerator and RTP_IO objects.

RTP packets are streamed from the RTP_IO object of Host A to the RemoteUnitGenerator at Host B. A subclass of the CSL RingBuffer class, RtpRingBuffer, buffers the network connection of RemoteUnitGenerator to guard against network jitter. The RTCP control protocol is used to provide Quality of Service statistics to all hosts, allowing for dynamic bandwidth control or lost packet interpolation to help ensure the reliability of the network connection.

4 Summary and Evaluation

Many systems in the literature address similar basic requirements but have vastly divergent design goals and aesthetics. In our case, we were driven most strongly by our desire to build an open and simple system based on a strict adherence to the 4MS metamodel. Our conscious use of known design patterns wherever possible also had a significant impact on CSL's flavor. Lastly, we deliberated on the various trade-offs, frequently between simplicity, scalability, and performance, often opting to prioritize the former over the latter.

5 Conclusions

Building a class hierarchy of digital audio unit generators for sound synthesis and processing is relatively easy. Building one based on a sophisticated metamodel that combines ease of use, flexibility, efficiency, and mature design principles is somewhat more of a challenge. In the design of CSL4, we added requirements that arose out of our recent projects. Simplicity and short learning curve have always been central design criteria for CSL, a feature that sets it apart from many of its siblings such as CLAM or SuperCollider. It has been in continuous use since 2000 as a platform for teaching graduate courses on digital audio software at UCSB (see http://www.mat.ucsb.edu/240).

This paper described several of the novel design choices and the use of modern design patterns in the most recent version of CSL. Our team continues to look for ways to make CSL a better client of the best results of current software engineering practice, and to drive the evolution of CSL with concrete applications

The CSL code base and documentation are available from the CREATE web site at http://create.ucsb.edu/CSL.

References

Amatriain, X. (2005). *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. Ph. D. thesis, Universitat Pompeu Fabra, Barcelona, Spain.

Amatriain, X. and P. Arumi (2005). Developing cross-platform audio and music applications with the clam framework. In *Proceedings of International Computer Music Conference 2005*.

Amatriain, X. and S. Pope (2006). An object-oriented metamodel for multimedia processing. *ACM Transactions on Multimedia Computing, Communications and Applications*. in press.

Arumi, P., D. Garcia, and X. Amatriain (2006). A data flow pattern language for audio and music computing. In *Proceedings of the 2006 Pattern Languages of Programming Conference*, Portland, Oregon. pending acceptance.

Burk, P. (1998). JSyn- A Real-time Synthesis API for Java. In *Proceedings of the 1998 International Computer Music Conference (ICMC '98)*. Computer Music Associaciation.

Freed, A. and M. Wright (1997). Open sound control: A new protocol for communicating with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference (ICMC '97)*. International Computer Music Association.

Lanski, P. (1990). The architecture and musical logic and cmix. In *Proceedings of the 1990 International Computer Music Conference (ICMC 90)*.

Mathews, M. V. (1969). *The Technology of Computer Music*. MIT Press.

McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal 26*(4), 61–68.

Pope, S., A. Engberg, F. Holm, and A. Wolf (2001). The distributed processing environment for high-performance distributed multimedia applications. In *Proc. 2001 IEEE Multimedia Technology and Applications Conf., U. C. Irvine*.

Pope, S., F. Holm, and A. Kouznetsov (2004). Feature extraction and database design for music software. In *Proceedings of the 2004 International Computer Music Conference (ICMC '04)*. International Computer Music Association.

Pope, S. and C. Ramakrishnan (2003). The create signal library (sizzle): Design, issues, and applications. In *Proceedings of the 2003 International Computer Music Conference (ICMC 2003)*. International Computer Music Association.

Pope, S. T. (1993). Machine Tongues XV: Three packages for Software Sound Synthesis. *Computer Music Journal 17*(2), 23–54.

Pope, S. T. (2003). Recent Developments in Siren: Modeling, Control and Interaction for Large-scale Distributed Music Software. In *Proceedings of the 2003 International Computer Music Conference (ICMC '03)*. Computer Music Association. Also in Journal of Object-Oriented Programming 1(1): 6-14.

Scaletti, C. (1991). *The Well-tempered Object. Musical Applications of Object-Oriented Software Technology*, Chapter The Kyma/Platypus Computer Music Workstation, pp. 119–140. MIT Press.

Schottstaedt, W. (2000). *Common Lisp Music Documentation*. http://www-ccrma-stanford.edu/software/clm: CCRMA-Stanford University.

Schulzrinne, H., S. Casner, R. Frederick, and V. Jacobson (1996). *RTP: A Transport Protocol for Real-Time Applications*. IETF RFC1889.

Wright, M. (1999). Audio applications of the sound description interchange format. In *Proceedings of the 107th AES Convention*.