Extending Aura with Csound Opcodes

Steven Yi, Victor Lazzarini
National University of Ireland, Maynooth
Department of Music
stevenyi@gmail.com
victor.lazzarini@nuim.ie

Roger Dannenberg
Carnegie Mellon University
School of Computer Science
rbd@cs.cmu.edu

John ffitch
University of Bath
Department of Computer Science
jpff@codemist.co.uk

ABSTRACT

Languages for music audio processing typically offer a large assortment of unit generators. There is great duplication among different language implementations, as each language must implement many of the same (or nearly the same) unit generators. Csound has a large library of unit generators and could be a useful source of reusable unit generators for other languages or for direct use in applications. In this study, we consider how Csound unit generators can be exposed to direct access by other audio processing languages. Using Aura as an example, we modified Csound to allow efficient, dynamic allocation of individual unit generators without using the Csound compiler or writing Csound instruments. We then extended Aura using automatic code generation so that Csound unit generators can be accessed in the normal way from within Aura. In this scheme, Csound details are completely hidden from Aura users. We suggest that these techniques might eliminate most of the effort of building unit generator libraries and could help with the implementation of embedded audio systems where unit generators are needed but a full embedded Csound engine is not required.

1. INTRODUCTION

Csound [1, 2] is a Music-N-based computer music system with a long history. Over time, it has been recognized that the Csound functionality could be valuable in forms other than the monolithic Csound command-line application. An embeddable engine evolved that can be used by desktop, mobile, and web-based applications. Especially with the continuing growth of Csound opcodes, the equivalent of Music-N unit generators, Csound offers a large library of signal processing elements. While these are available by using Csound as a whole or through an embedded Csound engine, there are cases where one might like to use individual opcodes or access the opcode library through alternative audio frameworks.

This paper will discuss research into the use of Csound opcodes within the distributed, realtime object and music system, Aura [3]. We will analyze how opcodes work within Csound, see what is necessary to use them outside of Csound, and show steps taken to recontextualize opcodes to function within Aura. Finally, we will explore future directions for this work and how it can be useful for research and music systems design. The main result of this work is a new interface that exposes direct access to Csound opcodes and the wealth of signal processing resources they represent. [1] We also offer a detailed description of the Csound opcode and instrument architecture.

2. RELATED WORK

Previous research has taken a different approach to the problem of unit generator code reuse. Several efforts have been made to create abstract representations of the signal processing within unit generators, allowing code generators to convert these high-level descriptions into implementations. The description can be as simple as a set of parameters and state variables and an inner loop written in C. For example, the RATL system [4] can generate unit generators for at least 4 different systems. Faust [5] is a functional programming language for signal processing that can be compiled into C++ implementations for a dozen or more systems. Finally, plug-in standards such as Steinberg's VST and LADSPA [6] provide a standard API for dynamically loadable audio signal processing modules. However, these modules typically have higher overhead than unit generators and may have graphical interfaces, so they usually contain larger building blocks such as entire virtual instruments.

3. ANALYSIS OF CSOUND OPCODES

Csound's system design is based on two key abstractions: Instruments, which represent a time-schedulable series of unit-generators, and Opcodes, the unit-generators that operate to generate or process values. These abstractions have a number of facets that must be considered in order to understand how opcodes can be used either inside or outside of the Csound framework. These facets include *context*, *definition*, *allocation*, *initialization*, *performance*, and *destruction*.

[1] Csound 6.02.0 and Aura 4 were used for this research. Their project pages are available at http://www.github.com/csound/csound and http://sourceforge.net/projects/aurart/, respectively.

3.1 Context

When a user compiles and runs Csound orchestra language code, a series of steps take place that contextualizes each architectural layer. First, a CSOUND structure is allocated. This structure contains the complete state for a Csound engine instance. This includes current definitions of instruments and opcodes, live instances of instruments and opcodes, current run-time state, and management of resources such as function tables. Certain properties, such as the current sampling rate and block size (called ksmps in Csound), are set in the CSOUND structure and referenced globally.

The CSOUND structure also contains function pointers for a number of functions that are used by opcodes as well as by host programs. These include such things as allocating memory and other resources, querying state, processing FFT data, and so on. It is important to note that an opcode's initialization and performance functions can and do use the data and function pointers within the CSOUND structure.

After the CSOUND structure is initialized, Csound Orchestra code is then compiled. This reads in definitions of instruments and user-defined opcodes, as well as global resources and opcodes to run once at the start of Csound's performance. At this point, the CSOUND structure contains definitions of instruments and user-defined opcodes, but does not yet contain any instances of those definitions.

Next, Csound score code may be read in and processed. This information will be used to trigger events at runtime, including instantiation or forced destruction of instrument instances, creation of function table resources, and ending the score (and thus stopping the Csound engine).

After all compilation is done, runtime begins. Before the initial run, opcodes found in the global code space (commonly called instrument 0) are executed. Next, Csound runs one audio block at a time. In that time, instrument instances may be scheduled to be instantiated or deactivated, and active instances will be run. Csound does not instantiate, deactivate, or run opcodes by themselves, but rather only as part of an instrument instance.

In addition to the CSOUND structure, opcodes may also read in information from the instrument instance they are a part of. This may include information such as if the instance of the instrument was initialized by MIDI, whether the instrument is in a held or releasing state, duration of note, and so on. More importantly, the value that is most often used from the instrument instance context is the local ksmps (buffer size) for the instrument instance. As Csound allows for setting local ksmps per instrument instance, all opcodes that work with audio-rate signals use the local ksmps value when calculating how much audio to render or process.

3.2 Definition

Csound opcodes are defined using the OENTRY data structure, as seen in Figure 1.

The data structure is made up of:

```
typedef struct oentry {
  char    *opname;
  uint16  dsblksiz;
  uint16  flags;
  uint8_t thread;
  char    *outypes;
  char    *intypes;
  int     (*iopadr)(CSOUND *, void *p);
  int     (*kopadr)(CSOUND *, void *p);
  int     (*aopadr)(CSOUND *, void *p);
  void    *useropinfo;    /* user opcode
    parameters */
} OENTRY;
```

Figure 1. Definition of OENTRY struct.

opname the name of the opcode as used in Csound orchestra code

dsblksize the size in bytes of the data structure to use with the opcode

flags bit flag that describes resource reading/writing dependencies, used by Csound's automatic parallelization algorithm

thread bit flag that describes if the opcode has init, k-rate, and a-rate performance functions

outypes a string description of the types used for the output arguments of the opcode

intypes a string description of the types used for the input arguments of the opcode

iopadr, kopadr, aopadr function pointers to use for initialization and performance of the opcode

useropinfo additional data used for user-defined opcodes

An OENTRY describes an opcode, but is not the instance of an opcode used at run-time. Instead, the information from an OENTRY is used to create, initialize, and perform an OPDS data structure, which is the active instance of an opcode. This is similar to the difference between a class definition and and object instance in Object-Oriented Programming.

Figure 2 shows the OENTRY definition for the oscils opcode.

```
{ "oscils",   S(OSCILS), 0, 5, "a", "iiio",
  (SUBR)oscils_set, NULL, (SUBR)oscils  },
```

Figure 2. OENTRY definition for the oscils opcode.

3.3 Allocation

The data structure for an opcode is allocated with a size equal to the OENTRY's dsblksize. The value for a dsblksize is set using `sizeof()` with a struct that will be passed into the opcode's initialization and performance functions. Note that it is the convention in Csound that the struct always starts with its first member being an instance of OPDS. This allows all opcode instances to be cast to OPDS and handled generically within the engine. Following the

OPDS are a set of pointers, one for each of the output and input arguments. These argument pointers are set by Csound at runtime, using the information defined in the intypes and outypes fields of the OENTRY. After the pointers for arguments to the opcode come any internal state data that the opcode will use between calls to its performance function. This layout of data is shown in Figure 3.

```
/* oscils opcode struct */

typedef struct {
  OPDS    h;
  /* opcode args */
  MYFLT   *ar, *iamp, *icps, *iphs, *iflg;
  /* internal variables */
  int     use_double;
  double  xd, cd, vd;
  MYFLT   x, c, v;
} OSCILS;
```

Figure 3. Definition for OSCILS struct, used for the oscils opcode.

Csound does not allocate memory for an opcode individually, but rather allocates a single large memory block for an entire instrument instance. The compiler tracks the total amount of memory required for an instance of an instrument. The total is a sum of the size of an INSDS struct, the dsblksize's of opcodes used within the instrument, and the sizes of types for the variables defined for the instrument. Upon allocation of the total memory block, the memory is then divided up using pointers to addresses within the block. As shown in Figure 4, the initial part of the memory is used as an instance of INSDS (the data structure for an instrument instance), the second part of the memory is used as variables, and the last part is used as opcode instances.

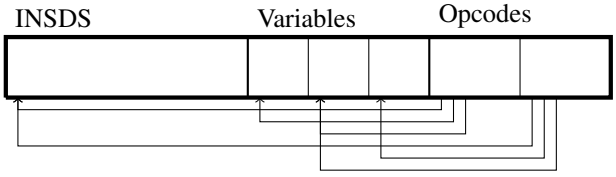

Figure 4. Memory block diagram for a Csound instrument instance.

The information for what opcodes and what variables are used in the instrument instance, as well as how to wire up the memory are all gathered up during the compilation phase. That information is stored with the instrument definition (the INSTRTXT data structure). Csound will allocate, then wire up the memory before any initialization of the instrument instance occurs.

3.4 Initialization

Once the memory is allocated for an instrument and wired together by setting pointers, Csound runs through the list of opcodes and calls initialization functions (if the opcode has an init-function). As shown in Figure 1, the iopadr has a function signature where it takes in a pointer to a CSOUND struct, as well as a void*. In general, the function used for the opcodes will have their second argument already cast to the type of the opcode's data structure. Figure 5 shows the initialization function of oscils with a second argument of OSCILS*, not void*.

```
int oscils_set(CSOUND *csound, OSCILS *p);
```

Figure 5. Function prototype for oscils opcode's initialization function.

This step in the opcode's lifecycle is generally used to pre-compute values that can be reused at run-time, as well as allocate any further resources that the opcode may need. The opcode will use values set in the input-argument pointers, as well as write values out to the output-argument pointers.

3.5 Performance

Csound's kperf() function is used to perform one buffer's worth of audio. In this time, active instances of an instrument are performed by running through each opcode for that instrument calling their performance function. This will map to the opcode's kopadr or aopadr function pointer, depending on what pointer was set for use during initialization. [2] The function is called with the same set of arguments as discussed in Section 3.4.

3.6 Destruction

For opcodes, there are two aspects to destruction. The first may be considered a form of deinitialization when an instance of an instrument completes (for example, when a note stops). In this scenario, any opcode that has registered a deinitialization callback will have that callback executed. The callback may be used to perform cleanup of resources that might be valid only for that instance.

The other aspect to destruction is when the memory for an instance of an instrument is being freed. Within a score section, Csound does not destroy instances of instruments when they become inactive and deinitialized. Rather, the inactive instance is left in a pool and made available for reuse and reinitialization. The memory for an instance is actually freed only at the end of a score section or at the very end of score rendering. When it is freed, all opcode instances for the instrument are included as they are subparts of the larger instrument instance memory, as shown earlier in Figure 4.

4. RECONTEXTUALIZING THE OPCODE

By analyzing how Csound uses opcodes in Section 3, the following points were understood to be necessary for using opcodes outside of the Csound engine:

1. Opcodes are defined in OENTRYs. We will need to reference the OENTRY to be able to allocate, instantiate, and perform an opcode.

[2] Csound has the ability to change what performance function is used by an opcode. This is done to optimize runtime code performance.

2. The Csound engine does not allocate an opcode's data structure on its own, but rather as part of a larger block of memory for an instance of an entire instrument. However, we should be able to allocate memory to use for the data structure on its own, using the dsblcksize field from the OENTRY.

3. Besides the opcode's data structure, opcodes may also rely on three other data structures for operation. These include the CSOUND, INSDS, and OPDS data structures. As OPDS is already part of the opcode data structure, we will not have to handle allocation specifically outside of allocation of the opcode data structure. On the other hand, we will need to allocate an instance of CSOUND and INSDS to use the opcode.

4. The CSOUND structure is used as an argument to opcode's functions, as is the opcode's data structure. The INSDS will have to be wired to the OPDS data structure in the opcode. Additionally, opcode input and output arguments are allocated outside of the opcode data structure, and pointers are set within the data structure to make the values from the arguments available for use by the opcode's processing functions.

Understanding the above, we set out to create a basic set of C++ classes that could encapsulate a single opcode for use outside of Csound. To do this, we have to support the entire lifecycle of opcodes–allocation, initialization, performance, and destruction. We also have to honor the aspects of Csound's internal design to allow the opcode to perform *as if it were running within Csound*. Additionally, we want the design to be flexible enough to function within any desired music system context, and in particular, within Aura.

From here, we designed two layers of classes. The first layer is a generic Opcode layer capable of creating opcode instances that can be used on their own. The second layer builds upon the first to use those opcodes within Aura. While both layers were developed within the Aura 4 code base, the first layer was developed with the intention that it could be used within other applications, and could even be moved into Csound's code base as part of its public API.

4.1 OpcodeFactory and CSOpcode

The generic Opcode layer uses two classes, OpcodeFactory and CSOpcode. OpcodeFactory is a utility class that handles allocation and pre-setup of CSOpcodes. In its constructor, it allocates and initializes a single CSOUND and INSDS that will be shared by all CSOpcodes. The CSOUND and INSDS within OpcodeFactory uses a ksmps block size of 32 samples, matching the default value of Aura. [3] By creating a single instance of CSOUND and INSDS, all opcode instances share the same world-view as if they were part of a single Csound instrument instance. This was determined to be enough to allow the target set of opcodes to function properly when run on their own.

Outside of the constructor and destructor, the OpcodeFactory class has one public method, shown in Figure 6.

```
CSOpcode* createCsOpcode(char* opName, char*
    outArgTypes, char* inArgTypes);
```

Figure 6. Public methods for OpcodeFactory class.

The createCSOpcode() method requires that the calling code pass in the exact name, intypes, and outypes strings that matches those of the OENTRY to use for the opcode. This design places the responsibilty for choosing what version of an opcode (in the case of using a polymorphic opcode) on the caller. We chose this design as it worked best for the Serpent code generation system discussed further below in Section 5.4.

With the given arguments, the OpcodeFactory will search the list of opcodes in the CSOUND structure that matches those parameters. If a valid OENTRY is found, createCSOpcode() calls the CSOpcode constructor (shown in Figure 7) to create a CSOpcode instance, using the shared CSOUND and INSDS structures, as well as the found OENTRY. The factory will then return the CSOpcode to the factory's calling code. If a valid OENTRY is not found, the factory will instead return NULL.

```
CSOpcode(CSOUND* csound, INSDS* insds, OENTRY*
    oentry);
```

Figure 7. Constructor for CSOpcode class.

The CSOpcode constructor allocates and sets up an instance of a Csound opcode. It stores a reference to the CSOUND structure to later pass in as an argument for the opcode's initialization and performance functions. It also allocates the opcode data structure and wires it up to the shared INSDS instance. Afterwards, using the the OENTRY's input and output argument type string, it determines the storage requirements in terms of Csound MYFLT's [4]. Once the storage requirements are calculated, a block of memory is allocated for the total size of the input and output arguments (this is held in the MYFLT* data member of the CSOpcode class). The argument pointers for the opcode are then configured to point to various addresses within the data block.

Note that the input and output argument types defined in an OENTRY describe allowable types. These types may be concrete types (i.e. i-, k-, or a-rate variables), optional argument of type x (i.e. the type specifier "o" means an optional i-rate variable that defaults to 0), or var-arg of type x (i.e. the type specifier "z" means an indefinite list of k-rate arguments). [5]

[3] For the purpose of research this was adequate to continue development, though this should be made configurable for general use.

[4] In Csound, MYFLT is a macro defined to be either a float or double.

[5] For more information about Csound's type specifications, please see Engine/entry1.c and Engine/csound_standard_types.c files, found within the Csound source code.

As some of the type specifiers may indicate types which have different storage requirements (i.e may be of type k or type a, the first being a single scalar value, and the latter being a vector value), the size of the possible types with the largest value is used. This ensures that there will be enough memory for the type that is actually used, regardless of which type is chosen.

4.2 Argument Handling

Once a `CSOpcode` is returned from an `OpcodeFactory`, the memory for the opcode data structure is ready to be used, but arguments for the opcode have not yet been set. Pre-configuring the opcode data structure to point to pre-allocated memory for arguments allows for two different approaches to argument handling (the methods for these approaches are shown in Figure 8). The first approach allows setting of opcode arguments by value. Using these methods will copy values to and from the data member of the `CSOpcode` class. Because the opcode data structure is configured to point to the values held in the `CSOpcode` data member, those values will be used when the opcode initialization and performance functions are executed.

```
void setInArgValue(int index, void *mem, size_t
    size);
size_t getOutArgValue(int index, void* mem);
void setInArgPtr(int index, void* mem);
void setOutArgPtr(int index, void* mem);
```

Figure 8. Methods for argument handling in `CSOpcode`.

The second approach allows for directly setting the argument pointer in the opcode data structure to an address supplied by the `CSOpcode` client. This approach assumes the client has allocated memory and that the size of the memory is equal in size to the space requirement for the argument that the opcode expects. For example, if the opcode expects an a-rate argument, it will expect that argument will point to memory equal to the size of `MYFLT` × `ksmps` block size. This approach removes the need to copy the value if the value is already allocated elsewhere and can lead to more efficient processing. Figure 9 shows a diagram of how the two approaches handle argument pointers.

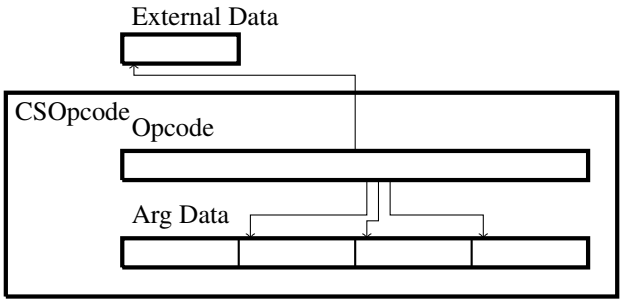

Figure 9. Memory diagram for `CSOpcode` and argument handling.

4.3 Initialization, Performance, and Destruction

Once arguments have been set by value or by reference, the opcode data structure is ready for initialization. `CSOpcode` exposes two public methods for initialization and performance (see Figure 10). opInit() delegates to calling the function pointer set as the `iopadr` in the `OENTRY`, passing in the `CSOUND` structure and opcode data structure. This is the same function as would be called if an opcode was being initialized within Csound's engine. The `opPerform()` function delegates similarly to the `opInit()` function, but instead uses either the `kopadr` or `aopadr` function pointers.

```
int opInit();
int opPerform();
```

Figure 10. Opcode initialization and performance functions in `CSOpcode`.

Once an init and/or performance function is called, the value in the output argument pointers for the opcode may be read with the updated value generated from the opcode. This can be done by either retrieving the value if using the set-by-value argument methods, or reading the memory directly for the pointer set on the opcode data structure.

When it is time to finish using the opcode, the `~CSOpcode()` destructor function will handle releasing memory for the Csound opcode and cleaning up the internal data allocated by `CSOpcode`.

The `OpcodeFactory` and `CSOpcode` class design allows for allocating, initializing, performing, and destroying an opcode instance, separate from its normal usage within a Csound engine. This completes the general usage layer of abstraction. Next we will discuss how this layer is used with Aura's object model and runtime system.

5. USING CSOUND OPCODES IN AURA

To use Csound opcodes in Aura, we must first analyze the differences between the abstractions and designs. Next, we must determine how to map concepts from Csound to Aura. Finally, we must develop a means to bridge the two together.

5.1 Aura Concepts

In Aura, there are two main abstractions for audio related code: `Instr` and `UGen`. These roughly map to Csound instruments and opcodes, but have features unique to Aura. Similar to an opcode, a `UGen` defines a signal generator or processor. Examples include oscillators, signal summers, and filters. Also like an opcode, `UGens` are used as part of an `Instr`. An `Instr` is basically a container for one or more `UGens`, much like Csound instruments contain opcodes.

`Instr` however, differs somewhat from Csound instruments. `Instrs` can use other `Instrs` as inputs and outputs, and the network of `Instrs` can be composed together within the Audio Zone at runtime, often under the

control of programs written in the scripting language Serpent [7].

In regards to the two abstractions, the `CSOpcode` class developed in Section 4 functions much like an Aura `UGen`, and would be used primarily within C++ where input and output data can easily be allocated and managed. However, to allow users to instantiate opcodes dynamically, possibly by writing code in Serpent, we need to wrap each `CSOpcode` within an Aura `Instr`. Rather than write many `Instr`s by hand, or even generate many `Instr` subclasses automatically, we developed a special `Instr` class that uses both `OpcodeFactory` and `CSOpcode` to interact with Csound, handle exchange of values between `CSOpcode` and clients of the `Instr` class, as well as function normally as any other `Instr` class would within Aura. Additionally, we developed the appropriate Serpent code to instantiate and use this new `Instr` class.

Figure 11 shows the design between the Csound, Opcode, and Aura layers.

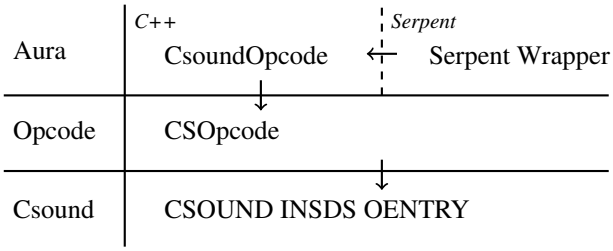

Figure 11. Architecture showing relationship between Csound, Opcode, and Aura layers.

5.2 Code Generation

Aura uses a preprocessing script to aid development. The preprocessor reads comments in .h (header) files and automatically generates C++ code and declarations for some `Instr` methods and for remote method invocation as well as Serpent wrapper code for instantiating the `Instr`. For this project, we designed a special `Instr` class called `CsoundOpcode` that can dynamically create a `CSOpcode` at initialization.

For native Aura `Instr`s, there is a one-to-one mapping of an `Instr` to its Serpent code wrapper. In the case of `CsoundOpcode`, the decision was made to have a one-to-many mapping. This means that the user writing Serpent code would be presented with many Csound opcodes to use, but that all of the Serpent wrappers would use instances of the same `CsoundOpcode` class. To achieve this, initialization steps were added to `CsoundOpcode` not found in other `Instr` classes. Also, a second Serpent generator script was designed to generate the opcode mappings that would reuse the generated `CsoundOpcode` Serpent code. More details of each follow below.

5.3 CsoundOpcode

The `CsoundOpcode` class is a sub-class of Aura's `Instr` class. As mentioned in Section 5.1, the class uses the Opcode layer to create and use `CSOpcode`s to bridge Aura `Instr` usage with Csound's opcode usage. In general, most of the Aura `Instr` lifecycle maps closely to Csound's opcodes, and `CsoundOpcode` simply delegates actions to `CSOpcode`.

The unique aspect of `CsoundOpcode` is its multi-step initialization. For a native Aura `Instr`, when Serpent code sends a message to create an instance of an `Instr`, the `Instr` is first constructed using its constructor, then an `init_io()` function is called as a means to set up argument pointers between `Instr`s, as well as perform other initialization. However, to accommodate the generic design of `CsoundOpcode` to map to multiple Serpent representations, the initialization steps of `CsoundOpcode` were modified.

First, the constructor for `CsoundOpcode` takes no arguments. At construction time, it only allocates the basic data for the class, but as of yet does no initialization. Next, the `init_io()` function just calls the parent class's `init_io()` with zero inputs and outputs. Instead of making the usual connection to other instruments, we will wait to do it at a later time.

Following the standard construction and initialization, a number of special methods were added. First, `set_opcode()` is a method used to set what Csound opcode the `CsoundOpcode` class should use. This passes in the exact opcode name, input arg string, and output arg string that should be matched against in the list of `OENTRY`s available from Csound. This information is then used by `OpcodeFactory` to create an instance of `CSOpcode`. Next, `set_a_input()`, `set_b_input()`, and `set_c_input()` functions are called. Each take in an int index for what argument to set by arg position, and an Aura object that should correspond to the Aura a, b, or c type of the function called. (Aura types are described below.) Once all inputs have been set, a final `init_complete()` method is called. This then performs the operations that a native `Instr` would in its `init_io()` function, setting up argument pointers.

While care must be taken to call these functions in a specific order, the user does not have to particularly worry about it as the generated Serpent code takes care to do all of the operations correctly. To the user, the Serpent code looks very much like any other Serpent class that wraps an Aura `Instr`.

5.3.1 Mapping Csound and Aura types

An important part of allowing `CsoundOpcode` to function within Aura as an `Instr` is mapping of Aura types to Csound types. In Aura, there are three types: a (audio-rate vector), b (control-rate scalar), and c (constant scalar). Fortunately, there is a direct mapping of these types to Csound's a-, k-, and i-type variables, respectively. Not only are they related in purpose, but they also match in storage requirements, if Csound is compiled with `MYFLT` set to float.

In general, Aura `Instr`s share values directly by reference, sharing pointers between `Instr` instances. When an `Instr` goes to process audio, it will first call the processing methods for the `Instr`s it depends on, then use

the values shared through the pointers directly. For flexibility in `CsoundOpcode`, code was written to check the `sizeof(MYFLT)` and compare to the `sizeof(float)`. If these match, then `CsoundOpcode` will use the standard Aura practice and share pointers, using the corresponding `CSOpcode` methods for setting and getting arguments by reference. If these do not match, this will be detected and extra work will be done to read and convert values to and from Csound. In this case, the `CSOpcode` methods for setting and getting arguments by value are used. This gives the flexibility for the Aura user to use the `CsoundOpcode` class with either the double or float version of Csound. [6]

Another important thing to note is that while there are corresponding types in Csound for Aura's types, the opposite is not true. Csound has other types for which Aura does not have a corresponding type. These include things like f-sig (phase vocoder analysis signals) and array data types. These types can be accessed through C++ but they are not automatically available using Serpent. This then restricts what opcodes can be supported by automatically generated code, as described in the following section.

5.4 Generating Serpent Code

The design of the `CsoundOpcode Instr` enables the use of Csound opcodes from Aura. However, to make this convenient and safe to use, we need to generate Serpent code that will create `CsoundOpcode` instances and configure them for the desired opcode. Additionally, we want to make what the user sees look like any other Aura Serpent code, with the Csound opcodes looking and functioning like native Aura `Instrs` in Serpent.

A Python script was developed to generate stubs in Serpent that encapsulate the operations and parameters needed to instantiate Csound opcodes. Python was used because Csound has an API available to Python. We use the API to query the available opcodes in Csound and then use that information to generate Serpent code. The script takes care not to generate Serpent classes for opcodes where argument types are not available in Aura. Also, a whitelist and blacklist system was added for special cases where `OENTRY`'s were marked up differently than what was documented in the manual, as well as for skipping generation for opcodes that really make sense only in the context of Csound instruments (i.e. opcodes for gotos, if-branching).

One other adjustment was required for Csound opcodes that are polymorphic based upon their output argument types. To handle these cases of polymorphism, the actual name of the generated class has the output types appended to them, i.e. "Linseg_a", "Linseg_k". This puts the burden on the user to understand and know what version of the opcode to call, but this was vastly simpler than implementing a type inference system.

The output from the script is a single Serpent file called csound_opcodes.srp. Using this code, end users can now avail themselves of Csound opcodes within their projects.

[6] In principle, one could also define Aura's sample type to be double and do all DSP in double precision.

The following section demonstrates usage of the generated Serpent script.

5.5 Example Code

Figure 12 shows a simple example making use of Csound opcodes within Aura, using the Serpent scripting language. The code begins by loading csoundopcode_rpc.srp, which was generated from the `CsoundOpcode` class. The information in that file is in turn used by the csound_opcodes.srp script, discussed in Section 5.4. This is all that is necessary for Aura Serpent users to begin to use Csound opcodes.

```
load "csoundopcode_rpc"
load "csound_opcodes"

def adsr(a, d, s, r, u)
    [a, 1, a + d, s, u, s, u + r, 0]

tone_bps = adsr(0.01, 0.1, 1.0, 0.5, 1.0)

def csTest(amp, freq):
    tone = Mult(Moogladder(Vco2(1.0,
           Linseg_k(freq, 0.4, freq * 2, 0.4,
                    freq, 0.1, freq)),
                2000, 0.9), Env(tone_bps), t)
    tone.name = "moogladder"
    tone.play()

rtsched.cause(4.0, nil, 'csTest', 0.5, 400)
rtsched.cause(6.0, nil, 'csTest', 0.5, 600)
rtsched.cause(8.0, nil, 'csTest', 0.5, 700)
```

Figure 12. Example Serpent code using Csound opcodes and Aura `Instrs`.

The next block of code defines a utility function that will pack a list with values appropriate for use with the Aura Env `Instr`. Then, `tone_bps` is defined to be used globally by the rest of the script.

Next is the `csTest()` function. Given an amplitude and frequency, it will create an enveloped, filtered, saw-tooth sound with a modulated frequency. It will last the duration of `Env Instr`, using the values from `tone_bps`. After creating the sound generator, it will call `play()` on it to schedule it for playback. Note that `Mult` and `Env` map to native Aura `Instr` classes, while `Moogladder`, `Vco2`, and `Linseg_k` all map to `CsoundOpcode Instrs`. The `CsoundOpcode`-based classes look and act in the exact same manner as the native Aura `Instr`-based classes. (For reference, Figure 13 shows an equivalent Csound ORC code example, written using Csound 6 function-call syntax style.)

The final part of the script uses `rtsched()` to schedule three events. It uses the `csTest()` function to generate and play `Instr` instances at times 4.0, 6.0, and 8.0. These events will play using starting frequencies of 400 hz, 600 hz, and 700 hz.

6. CONCLUSIONS

This paper has analyzed how Csound opcodes are used in Csound. We developed two layers of code to allow using opcodes outside of the Csound engine in general, as well as to use opcodes within the Aura music system. Bridging together two different music systems has shown us that while

```
0dbfs=1
nchnls=1

instr 1

iamp = p4
ifreq = p5

out(moogladder(
      vco2(1.0,
        linseg(ifreq, 0.4, ifreq * 2, 0.4,
          ifreq, 0.1, ifreq)), 2000, 0.9)) *
      adsr(0.01, 0.1, 1.0, 0.5)))

endin
```

Figure 13. Csound ORC example using function-call syntax.

system designs may differ, there are points of commonality that would encourage reuse between systems. The end result is a working example where Csound opcodes are used within Aura in a way that is natural for the Aura user.

For the future, we can see the generic Opcode layer discussed in Section 4 becoming a part of Csound's own public API. For other music systems developers, we see the possibility of Csound becoming a library and resource upon which to build larger systems. Within Csound itself, the ability to instantiate and wire up opcode instances individually invites experimentation with live signal graph modifications. This would allow a number of use cases to be addressed where Csound cannot currently be used, such as patcher applications with live graph modifications. Also, having an alternate compilation method within Csound that allocates opcode instances individually might facilitate the development of debugging facilities such as watches, probing, and logging.

Acknowledgments

This research was partly funded by the Program of Research in Third Level Institutions (PRTLI 5) of the Higher Education Authority (HEA) of Ireland, through the Digital Arts and Humanities programme.

7. REFERENCES

[1] R. J. Boulanger, Ed., *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February 2000.

[2] A. Cabrera, J. ffitch, M. Gogins, V. Lazzarini, and S. Yi, "The New Developments in Csound6," in *Proceedings of the 2013 ICMC Conference*, A. Gardiner and A. Varano, Eds. TURA and ICMA, 2013.

[3] R. B. Dannenberg and E. Brandt, "A flexible real-time software synthesis system," in *Proceedings of the 1996 International Computer Music Conference*, ICMA. Ann Arbor, MI: ICMA and HKUST, August 1996, pp. 270–273.

[4] K. MacMillan, M. Droettboom, and I. Fujinaga, "A System to Port Unit Generators Between Audio DSP Systems," in *Proceedings of the 2001 International Computer Music Conference*. International Computer Music Association, September 2001, pp. 103–106.

[5] Y. Orlarey, D. Fober, and S. Letz, *Faust: an Efficient Functional Approach to DSP Programming*. Edition Delatour, 2009.

[6] R. Furse, "LADSPA SDK Documentation," 2000. [Online]. Available: http://www.ladspa.org/ladspa_sdk/

[7] R. B. Dannenberg, "A language for interactive audio applications," in *Proceedings of the 2002 International Computer Music Conference*, M. Nordahl, Ed., ICMC2002. School of Music and Music Education, Göteborg University: ICMC, September 2002, pp. 509–515.