Extensible Computer Music Systems

Steven Yi

A thesis presented in fulfilment of the requirements for the degree of Doctor of Philosophy

Supervisor: Dr. Victor Lazzarini

Head of Department: Prof. Christopher Morris

Department of Music

National University of Ireland Maynooth

Maynooth, Co.Kildare, Ireland

May 2016

Contents

List of Figures

List of Tables

Listings

Acknowledgments

I would like to thank my supervisor, Dr. Victor Lazzarini, for his guidance, wisdom, and support. It was an honour to work with Victor, and I will always be grateful. I would also like to thank Dr. Gordon Delap for his thought provoking discussions and insights into my work.

I would like to thank the people in the Music Department and An Foras Feasa for making my time here a memorable one. Also, this research would not have been possible without the professional and financial support from the DAH Programme, HEA, and PRTLI5.

Outside of Maynooth, I would like to thank Dr. Roger Dannenberg for all of his inspiring work and our wonderful conversations. I would also like to thank Carnegie Mellon University for hosting me as a visiting researcher. Thanks also to Dr. John ffitch, Dr. Richard Boulanger, and the entire Csound community for all of their kinds thoughts and support of my work.

I would also acknowledge that the icons used for diagrams within this thesis were designed by Freepik [8], retrieved from www.flaticon.com [7], and are licensed using the Creative Commons BY 3.0 license [4].

Finally, I would like to thank my family and friends for all of their love and support. A special thanks my wife, Lisa: I am so grateful to have you in my life, and I dedicate this work to you.

Abstract

This thesis explores different aspects of extensibility in computer music software. Extensibility refers to how core developers, third-party developers, and users can extend software. It is a primary factor in determining a software's range of use cases and capacity to grow over time. This has a direct impact on the robustness of both the software and the user's work.

This thesis discusses four main areas of research: extensibility in programming languages, platform extensibility, run-time modular software development, and music systems as libraries. It also explores these areas through the development of four open-source software projects: Csound, Blue, Pink, and Score. Csound and Blue are existing programs that have been modified to provide new means of extension. Pink and Score are new software libraries designed for extension from the start.

The goal of examining extensibility is to help create long-living computer music software and – by extension – enduring musical works. These in turn will hopefully provide future developers, users, and curious students with the means not only to interact with the past through documentation, but also to actively explore, experience, and use these programs and works.

Chapter 1

Introduction

Computer music is the product of users employing software to create and render projects in the context of a computing platform. Users create works by developing projects and rendering them with software, potentially using multiple programs to achieve their musical goals. The programs used must have the features to meet the needs of the user. If the program does not satisfy the user's requirements, the program may be extended by its developers, or other programs may be necessary. These other programs may augment currently used software or replace them completely. For a work to function, all of the software used to perform it must be available and in working order. When programs die, users' works are in jeopardy of never being usable again.

Extensibility – the ability for a system to be extended – is an important facet of software. It refers to a number of different ways that a program grows. One aspect is how new features are introduced to a system. This includes programs where users extend the programs themselves using features provided by the program. It also includes programs where third-party developers provide features through plugins. In each of these scenarios, the core program

remains stable, yet the total system can grow to further accommodate users' needs.

Another aspect of extensibility is how a software grows to adapt to changes in platforms. New hardware becomes available and old hardware becomes obsolete. New versions of operating systems are released and features that programs depend upon may disappear. New platforms may also emerge. In the face of change, a software's platform extensibility, or its readiness to move on to new platforms or new versions of existing ones, is a factor in determining whether a program – and the works created with it – will endure.

The quality of extensibility is one that affects computer software in general. However, when applied to computer music, strong parallels are found with another music system: Western music notation. The comparison of computer music systems and projects as the digital counterpart to traditionally notated music scores is explored below.

1.1 Parallels with Western Music Notation

Richard Taruskin, in his Oxford History of Western Music, develops a theoretical framework of *literate music* based upon the traditional notated score. Within this framework, he traces the development of the written score, as well as the history of Western Music, from the beginnings of notated music to the present. Regarding what the development of notation enables, he writes:

> The beginning of music writing gives us access through actual musical documents to the repertories of the past and suddenly raises the curtain, so to speak, on developments that had been going on for centuries. All at once we are witnesses of a sort, able

> to trace the evolution of music with our own eyes and ears. The development of musical literacy also made possible all kinds of new ideas about music. Music became visual as well as aural. It could occupy space as well as time. All of this had a decisive impact on the styles and forms music would later assume. It would be hard for us to imagine a greater watershed in musical development. [172]

By the end of the work, Taruskin describes the rising popularity in using computers as signaling the decline of the literate tradition and the beginning of a postliterate era. In commenting on the use of computers to create works, he writes:

> When a majority of composers work that way, the postliterate age will have arrived. That will happen when – or if – reading music becomes a rare specialized skill, of practical value only for reproducing "early music" (meaning all composed music performed live). There has already been much movement in this direction. Very few, especially in America, now learn musical notation as part of their general education. The lowered cultural prestige of literate musical genres has accompanied the marginalization of musical literacy and abetted it; the availability of technologies that can circumvent notation in the production of complex composed music may eventually render musical literacy, like knowledge of ancient scripts, superfluous to all but scholars. [173]

While Taruskin sees technology as something different than the literate world of notated music, one that is pushing it aside, I see two models of music

that share much in their form and function. The concerns over entering a postliterate age could then just as easily be applied to computer music.

Western music notation is an open-ended format used to encode musical ideas in a written form called the *score*. It is made up of symbols endowed with specific meaning that are to be interpreted for performance. To understand the score, one must be literate in the meanings of the symbols used. The score is written onto a medium, that provides the context where the score will be created, read, and interpreted.

The symbols used for writing scores may be a part of a common practice, which embodies a well-known baseline of symbols that readers of scores are expected to know. However, notation is also extensible. Composers can define new symbols that they then employ in writing scores. These symbols are not a part of the common practice, and thus readers are expected to first learn the meanings of these symbols before interpreting the score. Composers may augment the existing knowledge of notation with their own definitions, though they may also define completely new forms of scores.

If the meanings of notation were lost, the score would be meaningless, and the work would fail to be interpretable by readers. If notation was a fixed format, musicians wanting to explore new means of sound production would be handcuffed by the available symbols. A new format would be necessary.

However, looking at the function of extensibility in Western music notation, we see a model that has successfully supported a community for hundreds of years. The system of notation has endured, being able to grow over time to encompass new features. Older and newer works exist together within this system, allowing the knowledge and experiences from past generations to be compared to and understood within the practice to today.

Comparisons can be made between Western music notation and computer music. In place of the score is the project, the data that encodes the work of the user. In place of the system of notation is the software, used to create and interpret the project. The program may act as the performer, or it may be used together with other programs for performance. In this case, the model of a performer would also map to software, just of a different kind.

In addition to this comparison is the mapping of the medium of notation and the computing context. If a score is written on paper – whether hand-written, printed, or computer-published – or presented digitally on a screen, the literate reader can still decipher the symbols and interpret the score. The knowledge of symbols and meanings functions across these mediums. In computer music, software is used to interpret projects. If a project is moved from one platform to another, as long as the program functions on the target platform, the program is capable of reading the project and interpreting it.

Western music notation has provided a community the means by which to create works that has endured over time. Extensibility has played a large part in this tradition to allow notation to expand beyond its original features to include new symbols and ways of music making. These qualities of extensibility and notation can be similarly projected upon extensibility and music software, for the same benefits of long-lived systems and enduring works.

1.2 Goals and Methodology

This thesis explores qualities of extensibility in computer music software. The goal in this research is to better understand and find ways to design software that can:

- Make music systems that are easier to preserve.
- Make music systems that can grow over time.
- Make musical works that are easier to preserve.
- Protect the investment of time in systems by developers and users.

This thesis explores four different areas of extensibility. Each area provides context through the analysis of existing systems and discusses new work implemented in software projects that engages with the particular area of extensibility that is under investigation. This will be applied both to existing systems, exploring new ways to make programs further extensible, as well as new systems, implemented as music libraries designed for extensibility.

1.3 Thesis Overview

Chapter 2 examines user- and developer-extensibility through the evolution of the Csound Orchestra programming language. It will present an overview of the original language and trace its developments through Csound 4 to Csound 5. Focus will be drawn to historical changes that introduced new ways for users to express ideas and extend the system in their own code.

Chapter 3 presents original work to further extend the language design and implementation of the Csound Orchestra language. It will discuss changes that were released as a part of Csound 6 and developed for release in Csound 7. These developments further contribute to the evolution of the language for extensbility by users and developers.

Chapter 4 explores platform-extensibility: the ability to use software across platforms. This chapter will analyse the various qualities of software

that affect platform-extensibility. These qualities will be discussed through the work of porting Csound to new plaforms – Android, iOS, and the Web. Discussion will include the design and development of the cross-platform, yet platform-specific, CsoundObj API. This chapter will also present various case studies of software that have been written using Csound on these new platforms.

Chapter 5 examines run-time modular software development and its applications to computer music software. It presents an analysis of various software architecture archetypes according to extensibility and explores how modular software uniquely builds upon the strengths found in other systems to offer the foundation for software ecosystems. Modular software techniques will be further explored through the development of Blue's modular Score timeline, which provides a unique way for developers to extend the Score interface through plugins.

Chapter 6 explores music systems as libraries in the context of general-purpose programming languages. The chapter begins with an analysis of language-based computer music systems. The analysis will compare and contrast systems that employ domain-specific languages, and those that are designed for general-purpose languages, and look at their impact on extensibility. Next, two new music libraries – Pink and Score – will be presented. These libraries will be used to explore music systems that are designed from the start for extensibility.

Finally, Chapter 7 summarises the thesis and draws conclusions based on the research as it has been presented in this document. The chapter also provides a listing of original work for the thesis and considerations for future work.

Chapter 2

The Csound Orchestra Language

This chapter will look at the history of the Csound [35] Orchestra language and its evolution from its beginning to Csound 5. It will begin with a look at Early Csound to see where the language began. Next, it will cover developments in Csound 4 that provided new ways for users to express their ideas as well as new kinds of things to express. Finally, Csound 5's new parser design will be discussed and how it helped developer extensibility of the language.

2.1 Overview

Csound's Orchestra (ORC) language is used primarily for defining *instruments*. These are used most often used to generate and process sounds. Instruments may also serve non-audio purposes to generate events and handle control input. The ORC language of Csound has largely been stable since its inception through early Csound 4. New developments in Csound were made within the confines of the original syntax. This included abstractions of instruments and opcodes as well as a limited set of variable types. To extend the system, a

developer would have to write code in C to create new opcodes or add new variable types.

It was not until later Csound 4 that significant new language changes were added. During the development of Csound 5, a new parser was implemented that would lay the groundwork for future language extension. Through this time, new versions of Csound were backwards compatible because extensions of the langauge were always made as additions.

The following will present an analysis of the Csound Orchestra language. It will begin by looking at the earliest form of the language that existed until Csound 4. Next, it will discuss changes that occurred in Csound 4 and Csound 5 that led to greater extensibility for users and developers. The history presented here will provide the foundation that the original work for this thesis (presented in Chapter 3) is built upon.

Note: Four different versions of Csound code will be discussed here and in Chapter 3. The first is the 1988 version of Csound, available on the CD-ROM provided with *The Audio Programming Book* [34], which will be called *Csound1988* in this text. The next three versions used in this text – Csound 5, 6, and 7 – will be called here *Csound5*, *Csound6*, and *Csound7*. Each of these versions is available online within the Csound project site [180] on GitHub.[1] The term *Early Csound* will be used to denote versions earlier than 5.00.

[1]The repository at GitHub contains the history of Csound since version 5.00. Each release has been tagged using `git` tags. At the time of this writing, Csound 6.04.0 is the most recent release and is available in the `master` branch. New development work for Csound 6 is available in the `develop` branch, and new work for Csound 7 is available in the `feature/parser3` branch.

2.2 Early Csound

Csound's original Orchestra language existed through Csound 4. It was a statically, strongly-typed programming language made up of a set of basic concepts: instruments, opcodes, variables, expressions, functions, and labels. It bore a resemblance to other Music-N languages of the past, which in turn had similarities to various assembly programming languages. The following describes each of the various parts of the language.

2.2.1 Instruments

Instruments are the primary abstraction that users employ to define units of computation run by the engine. Users define instruments by using the `instr` keyword, followed by a numeric identifier, then writing their processing code, and ending with the `endin` keyword. The processing code is written using *statements*, which are single lines of text that declare what *opcode* to use, together with their inputs and outputs.

```
      instr 1
iamp  =     0.5
ifreq =     440
aout  vco2  iamp, ifreq
      out   aout
      endin
```

Listing 2.1: Example Csound instrument

Listing 2.1 shows an example instrument definition. It defines an instrument with an identifier of 1 and includes 4 statements. The first two lines use the `=` opcode to assign the values 0.5 and 440 to `iamp` and `ifreq` respectively. Next, the `vco2` opcode is used with `iamp` and `ifreq` values as inputs and

generates the `aout` output audio signal. Finally, the `out` opcode is used to write the `aout` signal to the target audio file or sound card.

Users generally use Csound instruments to generate, analyse, and/or process sound. However, instruments are not limited only to sound related work but also may be used to process control values and execute non-audio tasks. This provides the user with the flexibility to employ instruments for any kind of processing to be done within the engine.

2.2.2 Opcodes

Csound *opcodes* are objects that can operate on zero or more input values and return zero or more output values. Csound opcodes are equivalent to the concept of unit generators in earlier Music-N systems. Listing 2.2 shows the general syntax for opcodes.

```
[output arguments] opcode_name [input arguments]
```

Listing 2.2: Csound opcode syntax

Opcodes in early Csound have a similar syntax to opcodes one would find as instructions in an assembly language. An opcode in assembly code would represent a primitive machine code instruction, such as fetching memory from a memory address, adding values from memory addresses, and storing results in another memory address. However, unlike assembly, where an opcode would have all memory locations – whether input or output – to the right-hand side of the opcode, Csound's opcode syntax typically uses outputs to the left of the opcode name and inputs to the right.

2.2.3 Variables

Variables are named locations in memory that store values. The meaning and size of the memory used for a variable is determined by its *type*. Variables are given names by the user that conform to the format specified in Listing 2.3. Here, a type-specifier is a single character that denotes the type of the variable, and the name-specifier is the unique name for the variable. The type-specifier in early Csound could be one of a few known types (i.e., `i-`, `k-`, `a-`, etc.). The name-specifier could be made up of one or more characters, numbers, or underscores.

```
[type-specifier][name-specifier]
```

Listing 2.3: Csound variable syntax

When parsing code in early Csound, a `NAME` entry would be registered for each variable identified in code. The `NAME` entry would record the variable's type as well as an index saying what number of a type it was (i.e. the 3rd `k` variable found). The total count of each variable type, together with the size of each type, was used at runtime to determine how much memory would be required as part of an instrument instance. The index that was stored in the NAME structure was later used to setup pointers into an instrument instance's memory.

Variables are used in the system to store values. Opcodes in turn read and write values from and to variables. The use of variables and opcodes together determine how data flows during instrument processing at runtime.

2.2.4 Constants

Constants are hard-coded values that do not change. In Csound, there are two types of constants: numbers and strings. Listing 2.4 shows an example of both constant types: `440` is an example of a numeric constant and ``A String Value'' is an example of a string constant.

```
;; numeric constant
i1 = 440
i2 = 440

;; String constant
Sval1 = "A String Value"
```

Listing 2.4: Example of constants

When the compiler processes constants, it stores a copy of the values in constant pools, with numbers stored in the `MYFLT` pool and strings in the string pool. For each constant, only one unique copy of the value is stored in each pool. For example, in Listing 2.4, the `i1` and `i2` variables are both assigned the value 440. The value 440 is stored once into the `MYFLT` pool, and the location of that value is shared with both assignment calls.

Note that the term *pool* has multiple meanings for computer science. In the preceding paragraph, pools refer to *lookup tables* and the term is used in the same manner that the Java Language Specification [81] describes Java's use of constant pools and String pools. Another use of the term is described by Kircher and Jain [100] in the context of a design pattern that "describes how expensive acquisition and release of resources can be avoided by recycling the resources no longer needed." The use of pool as a lookup table will be the assumed meaning for this thesis; if resource pooling is the intended meaning, it will be clearly marked as such.

2.2.5 Expressions

In addition to variables and constants, inputs to opcodes could also be expressions. Expressions allowed the inlining of operations to occur within the same line as opcode call statements. Listing 2.5 shows an instrument that uses code with expressions. The second `oscil` statement uses the value of "ipch * 2", and the statement that follows assigns the result of "a1 + a2" to a3.

```
instr 1
i1 = 440
a1 oscil 0.5, ipch, 1
a2 oscil 0.5, ipch * 2, 1  ;; EXPRESSION
a3 = a1 + a2 ;; EXPRESSION
out a3
endin
```

Listing 2.5: Example of expressions

Note that while the Orchestra language allowed for expressions to occur inline as part of the arguments to an opcode, the actual compiled result would be a single list of opcode statements. Each operation within an expression would be converted into an opcode call with its results assigned to a synthesised variable. (Synthesised variables have names starting with the # symbol.) Listing 2.6 shows what the compiled result for Listing 2.5 would look like if it was decompiled back to Csound Orchestra code.

```
instr 1
i1  = 440
a1  oscil 0.5, ipch, 1
#i0 mul ipch, 2          ;; ipch * 2
a2  oscil 0.5, #i0, 1
#a0 add a1, a2           ;; a1 + a2
```

```
a3  = #a0
out a3
endin
```

Listing 2.6: Example of expressions after compilation

Function Calls

Function calls allow for the use of secondary opcodes in-line within opcode statements. The results of the function call are used as arguments to the top-level opcode. They are a form of expression that may be combined with other mathematical operations.

```
a1  oscil 0.5, cpspch(8.00), 1
```

Listing 2.7: Example of function call as argument to oscil

Listing 2.7 shows the use of `cpspch` with an argument of 8.00. The result of calling `cpspch` is further used as an argument to the oscil opcode. The compiled result in Listing 2.8 shows how function calls are handled similarly to other mathematical expressions. Function calls have been available as early as Csound1988[2] and were processed as part of the expression handling code.[3]

```
#i0 cpspch 8.00
a1  oscil 0.5, #i0, 1
```

Listing 2.8: Compiled result of function call

Early Csound implementations of function calls were limited to a single argument as input. They also required that the type of the output match the

[2]For Vercoe's systems, function calls were also available as early as MUSIC 360 using "Sublist Notation." [177]

[3]This code is found in express.c in Csound1988 and is handled in Engine csound_orc_expressions.c in modern Csound.

type of the input. With these restrictions, only a certain limited number of opcodes would be allowed as a candidate for use as a function call. (This limit on arguments and type restrictions was later removed as part of Csound 6, discussed in Section 3.2.3).

2.2.6 Labels

Labels name places in code that can be the targets of jumps initiated by goto statements. In early Csound Orchestra, labels and gotos were the only option for controlling program flow. Labels are defined with an alpha-numeric identifier, followed by a colon and either whitespace or a new line. Listing 2.9 shows an example of defining a label(`loopStart:`) and its usage as a target of program flow by an if-goto statement. When executed, the example code increments `kval` once per loop and repeats this 32 times.

```
k0 = 0
loopStart:                    ;; label definition
k0 = k0 + 1
if (k0 < 32) goto loopStart   ;; label use
```

Listing 2.9: Example of label and goto

Edsger Dijkstra famously criticised gotos and labels as “just too primitive" and an “invitation to make a mess of one’s program”. [61] The proposition to move towards higher-level programming constructs from Dijkstra were further developed by Knuth [101] and others and manifested itself under the umbrella of *Structured Programming*. Since then, many programming languages (i.e., Java, Python, Ruby) abandoned exposing gotos and labels for programmer use, instead employing them only in compiled code generated by their compilers.

2.2.7 Discussion

The early Csound Orchestra programming language provided a simple syntax for expressing musical processing code. The definitions of instruments and use of opcode statements had similarities with Vercoe's earlier systems, MUSIC 360 [177] and MUSIC 11 [178], as well as earlier Music-N systems, such as Music V [121]. Users coming to Csound at that time from other systems – especially Csound's direct ancestor MUSIC 11 – would have found the language familiar and easy to learn.

However, the Orchestra language at that time did have drawbacks. Firstly, the language did not provide structured programming constructs, such as if-else branching or while-loops. Using labels and gotos made for code that could be difficult for users to write as well as read. Next, the requirement that function calls in expressions allow only a single input and output of the same type limited expressive possibilities for programming. Relaxing those requirements would open up more of the available opcodes for use with function calls. Finally, the language and system provided users with variable types and opcodes to use, but provided no mechanism for users to create their own. The only way for new types and opcodes to enter into Csound was by modification of the core program itself.

2.3 Csound 4

Near the end of the Csound 4 series of releases, the Orchestra language was extended, adding new control-flow syntax as well introducing a new concept of User-Defined Opcodes (UDOs). The primary additions were if-then clauses, subinstruments, and UDOs. These contributions to the language moved

the Orchestra language further away from its roots, providing the user with programming facilities commonly found in other programming languages.

2.3.1 Branching (if-then)

In Csound 4.21, Matt Ingalls introduced an alternative to if-goto style statements, called if-then. With if-then, rather than having an opcode like abstraction to determine control flow, the user was able to have a higher-level way to organise control-flow branches. Listing 2.10 shows an example of branching code using early Csound if-goto style as well as Csound 5 if-then style. Note, these two examples are functionally identical.

```
;; Early Csound if-goto
if (p4 == 0) goto branch1
if (p4 == 1) goto branch2
if (p4 == 2) goto branch3
goto branchDefault

branch0:
  ... code ...
goto end

branch1:
  ... code ...
goto end

branch2:
  ... code ...
goto end

branchDefault:
```

```
  ... code ...
end:

;; Csound 5 if-then
if (p4 == 0) then
  ... code ...
elseif (p4 == 1) then
  ... code ...
elseif (p4 == 2) then
  ... code ...
else
  ... code ...
endif
```

Listing 2.10: if-then statements in Csound 5

With if-then, one defines branches of code that would optionally be executed. Which branch of code is executed is determined by the test conditions supplied to `if` or `elseif` statements. A default branch may also be supplied using an `else` statement that will execute when no other conditions pass.

Using if-then code is arguably more concise and expressive than if-goto code as well as easier to read. Comparing the two code examples in Listing 2.10, the if-then style uses roughly half the amount of lines to express the same program as the corresponding if-goto style code. Also, the if-then code is organised into a set of conditions and consequences, which may more closely align with the user's thought process, than if-goto, which aligns well with the computer's execution process.

The introduction of if-then represents a large shift in Csound's history. Introducing new opcodes to Csound, such as new signal processing routines,

extends *what* users could say in the language. Introducing changes to the language, such as with if-then, changes *how* users express themselves altogether.

2.3.2 Subinstruments

In addition to if-then, Matt Ingalls also introduced subinstruments in Csound 4.21. Subinstruments was a feature that allowed users to call one instrument from another. This was a particularly powerful feature as it allowed the user to define instruments as normal, but use them as one would use an opcode.[4]

The implementation of subinstruments required no new ORC language syntax. The system was modified in two ways: first, a named instrument could be called directly as if it was an opcode. To enable this, when the compiler read in named instrument definitions, it would not record the instrument definition, but also create an `OENTRY` in the global opcode table to define a new opcode. Second, one could use the `subinstr` opcode to call a named or numbered instrument. When the `subinstr` opcode was initialised, it would create an instance of the subinstrument and delegate further initialisation and performance calls to the subinstrument. Note that when the system created new `OENTRYs`, those entries would delegate to `subinstr` internally.

```
instr 1
aout vco2 p4, p5
out aout
endin

instr MyOscil
aout vco2 p4, p5
out aout
```

[4]For further information, see the documentation for subinstr. [184]

```
endin


instr 2
iamp = 0.5
ifreq = 440
asig subinstr 1, iamp, ifreq ;; subinstr call
asig2 MyOscil iamp, ifreq ;; named instrument call
out asig + asig2
endin
```

Listing 2.11: Example Csound 4 subinstrument usage

Listing 2.11 shows an example use of subinstruments. Instrument 2 defines an amplitude and frequency, then calls `subinstr` to execute an instance of instrument 1 using those values. When that instance of instrument 1 is run, it will share the same p2 and p3 values from the parent instrument (instr 2), but it will receive `iamp` and `ifreq` as p4 and p5 values. The resulting audio signal output from instrument 1 is then returned to the subinstr call in instrument 2, and the value is assigned to `asig`. The line following the `subinstr` call shows the equivalent processing code, using the MyOscil named instrument directly as an opcode.

While subinstruments did not change the Orchestra language, the feature itself enabled users to work in new ways. For the first time, users could use the Orchestra language itself to define reusable bodies of code, as if they were writing their own opcodes. This introduced a new form of user-extensibility to Csound. However, subinstruments had a severe drawback in that they could only return audio signals to their caller. This limited what kinds of code could be reused. In addition, the feature to directly call instruments as opcodes was lost in Csound 5, though the `subinstr` opcode continues to

function today. While this represents a break in compatibility, it did not have a great impact as User-Defined Opcodes, discussed below, provided similar features to subinstruments and was more widely adopted.

2.3.3 User-Defined Opcodes

After Ingalls introduced subinstruments, Istvan Varga introduced User-Defined Opcodes [183] in Csound 4.22. Based on the work by Ingalls, UDOs provided a means for users to define their opcodes using Csound Orchestra code. Unlike subinstruments, UDOs could not be scheduled to run like instruments and only function as opcodes.

```
opcode myOpcode, i,i
  ival xin
  iret = ival + 1
  xout ival
endop
```

Listing 2.12: User-Defined Opcode in Csound 4

Listing 2.12 shows an example of a UDO. The first line defines an opcode with the name myOpcode and declares one output variable of type `i` and one input variable of type `i`. The `xin` and `xout` opcodes are used to assign input to variables and return variables as output. The number and types of input and outputs must match those declared in the initial opcode line. Processing code is written between the `xin` and `xout` lines. Finally, `endop` completes the definition of the UDO.

Unlike subinstruments, which did not extend the language grammar but did extend the semantics of instruments, UDOs brought forth new syntax to define opcodes. In terms of implementation, the internal UDO processing

code was based upon the instrument instance and delegation system written by Ingalls for subinstruments. However, it was extended to allow for types other than audio signals to be returned from the opcode. Also, when a UDO is defined, it compiles to the same data structure as used by instrument definitions. That way, runtime processing for a UDO or instrument can be the same. Finally, when the system compiles UDO, it creates a new `OENTRY` and appends it to the engine-wide list of opcode entries. Once the `OENTRY` is registered, the UDO is treated by the system like any other natively-programmed opcode.

UDOs extended the language of Csound to allow user-extensibility at the level of the opcode. The mechanism to define UDOs acts similarly to subinstruments, but aligns more closely with the native opcode model. However, as the language of Csound continued to develop, the design of UDOs presented drawbacks, particularly in the definition syntax. These issues and solutions will be addressed in Section 3.3.4.

2.3.4 Summary

Csound 4.21 and 4.22 brought major changes to the Csound Orchestra language and system. if-then provided a more structured approach to expressing conditional code. This allowed users to express musical processing in easier to write and understand ways. Subinstruments and UDOs provided user-extensibility at the level of opcodes. This empowered users to create their own opcodes using the Csound Orchestra language itself.

2.4 Csound 5

While significant changes to Csound's language occurred in Csound 4, the internal code changes were implemented by extending the same parser and compiler code that derived from the earliest Csound code.[5] When Csound 5 was first released, the major version was incremented to mark the large redesign of the internals to make Csound re-entrant as well as for the introduction of the Csound API (Application Programming Interface). These changes provided a new approach to using Csound as a library to embed within other programs. However, through the Csound 5 series of releases, another large internal change would occur: the introduction of a new parser and compiler, called *NewParser*. Developed by John ffitch and myself, the NewParser would provide a foundation upon which the Csound Orchestra language would continue to develop. The following will discuss the original parser, the NewParser that was developed in Csound 5, and until-statements, which were developed using the new parser.

2.4.1 The Original Parser

The original Csound parser and compiler (hereafter *OldParser*) was hand-written completely using C code. It was responsible for reading in text, interpreting it as code, verifying the code was valid, and compiling data structures that would be used for performance at runtime. It was here that the Csound Orchestra language was implemented.

[5]For reference, the code files that contained the lexing, parsing, and compilation code were primarily held in rdorch.c, rdscor.c, express.c, and oload.c. These files can be found both in Csound1988 and Csound5.

Looking at the source code to the OldParser from Csound1988, one finds an arguably simple set of functions for processing the language. This is not surprising, as the language was relatively simple at that time. However, by the time of Csound5, as the Orchestra language grew more complex, so did the OldParser. The Csound5 source code reveals that the preexisting functions for the OldParser not only grew in size, but that many new functions were also introduced. The original design and implementation of the OldParser was perfectly suitable for the original language, but became increasingly difficult to understand and modify as the system evolved.

By Csound5, the OldParser's complexity presented two main problems. Firstly, if a problem with the language interpreter was found, it was difficult to debug and find the root cause of the bug. Secondly, as the compilation process had mixed syntactic analysis, semantic analysis, expression processing, optimisation, and compilation all together, it was difficult to figure out how to develop new features. These problems – maintenance and development – were key areas where the Csound language interpreter could benefit from a new design.

2.4.2 NewParser

The *NewParser* was a rewrite of Csound's interpreter using Flex [135] and Bison [62], open-source versions of the classic Lex and Yacc tools [114].[6] Within the domain of compiler construction tools, Flex and Bison are considered *lexer* and *parser* generators, repsectively. With compilers, the lexer is responsible for reading individual characters from a stream of text and

[6] A broader discussion of software development tools and processes is presented in Section 4.2.1.

breaking that down into groups of characters. These groups are called *tokens* and may be considered the "words" of the language. The parser is responsible for pulling tokens from the lexer and applying syntactic analysis according to rules specified in a *grammar*. This determines if the stream of tokens found by the lexer are correctly organised and valid statements. If the lexer forms "words", then the parser organises "words" into "sentences". When the parser identifies groups of tokens that match a rule, an *action* is performed.

Rather than write the lexer and parser parts of a compiler by hand in C, Flex and Bison provide domain-specific languages (DSLs) for specifying lexical and syntactic analysis rules. When given code written in their respective DSLs, these programs generate the C code for lexers and parsers, respectively. Developers using lexer and parser generators can focus on the higher-level specification of the language and employ the tools to produce efficient and correct language processing code. A developer will still write C code for the actions of the parser to customise processing for their own specific compiler. Some compilers use parser actions to immediately perform some task, such as adding two numbers together and printing the output to screen, while other other compilers, such as Csound, may use actions to assemble an *abstract-syntax tree* data structure to use for further processing by later stages of the compiler.

The following will discuss the differences in design between the OldParser and NewParser. Discussion about the impact of the NewParser will follow.

Design

Csound's OldParser's architecture resembled closely the single-pass compiler design described in [14, Chapter 2: A Simple One-Pass Compiler]. Using the

terminology from [14], Csound's OldParser had two distinct parts: a *lexical analyser* and a *syntax-directed translator*. This is shown in Figure 2.1. Code would be loaded in as text by the system and read by the lexer as a stream of characters. The lexer would then analyse the stream and emit tokens. Next, the translator would read in the tokens, perform syntactic analysis, expression processing, and semantic analysis, and finally generate the run-time data structures (marked as "Output" in the diagram) for use with Csound's engine.

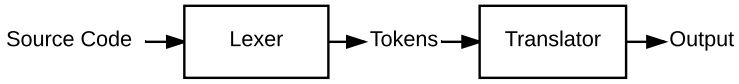

Figure 2.1: Csound OldParser Design

Csound's NewParser provided a separate compilation path than the OldParser, shown in Figure 2.2. With the NewParser, code was sent to the Flex-generated lexer as a stream of characters. The lexer would in turn emit tokens that would be read by the Bison-generated parser. The parser would do syntactic analysis of the tokens and generate a parse tree using Csound's `TREE` data structure. The `TREE` was then passed to the compiler, which performed expression processing, semantic analysis, and generation of the run-time data structures for Csound's engine.

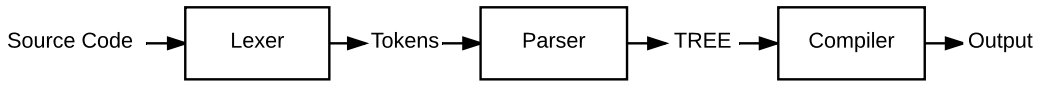

Figure 2.2: Csound NewParser Design

With the new `TREE` data structure, processing was now done using an in-memory representation of the code. This allowed the code for the compiler to be organised into multiple phases, rather than completely intertwined as it was in the OldParser. The NewParser did not completely organise code into

phases at this time, but it did establish a foundation that would be developed further in newer versions of Csound.

2.4.3 Discussion

With the initial development of the NewParser, the Csound Orchestra language design did not change, only its implementation. The OldParser and NewParser generated the same output, and the NewParser was developed to process the same code in the same way as the OldParser, emitting the same correct results and errors. The benefits then at this time were purely for the core Csound developer.

Firstly, the NewParser arguably provided a clearer view of the Orchestra language's design and how it was processed. With Flex and Bison, the code used to specify the tokens and grammar rules more clearly showed the syntactic structure of the language as well as the component "words" of the language. With the `TREE` output from the parser, the syntactic and semantic analysis aspects of the system were clearly separated and thus, more easily understood.

Secondly, the NewParser's clarity led to better extensibility of the language implementation. The phases of compilation with the NewParser more clearly outlined where in the codebase one would introduce new language features and where to introduce code for processing of those features. The benefits of extensibility would first be realised in the implementation of until-loops by ffitch in Csound 5.14, described in Section 2.4.4.

In addition to extending the language, the NewParser would facilitate other interpreter-related system changes. A prime example of this was the introduction of Csound's automated parallel processing facilities by ffitch in [66].

The implementation of the annotation aspect of the parallel implementation was done by extending the NewParser.

However, one drawback to the NewParser's design was that it still mixed semantic analysis within various phases of the compilation process. This had similarities to how semantic information was processed in the OldParser and served well enough for the Orchestra language in Csound5. The design of the NewParser would later change to introduce a new semantic analysis phase to support new language work in Csound6. These changes would undergo further redesign that would lead to the introduction of Parser3 (see Section 3.3.1) in Csound7.

Also, using Flex and Bison required learning new DSLs and tools. This adds a burden to developers to learn these tools if they wanted to work on the language. However, these tools are well-known and well-documented, such that there is a clear way to go about doing so. In comparison to trying to learn the OldParser code, I would argue that the benefits of the tools outweighed the cost of learning them.

2.4.4 until-loops

ffitch introduced until-loops [185] in Csound 5.14 using the NewParser. This provided a more structured programming approach to expressing looping in code. It removed the need for users to create labels and more clearly organised the looping body of code.

```
1 ;; Loop with gotos
2 kndx = 0
3 loopStart:
4   ... code ...
5   kndx = kndx + 1
```

```
6  if (kndx < 8) goto loopStart
7
8  ;; Loop with loop_lt
9  kndx = 0
10 loopStart:
11   ... code ...
12 loop_lt kndx, 1, 8, loopStart
13
14 ;; Loop with until
15 kndx = 0
16 until(kndx >= 8) do
17   ... code ...
18   kndx = kndx + 1
19 od
```

Listing 2.13: Looping examples with Csound 5

Listing 2.13 shows three ways to implement looping in Csound5 code. Line 1 shows the classic use of gotos and labels. Line 9 shows the use of the `loop_lt` [182] opcode, which handles index variable, increment amount, limit, and target label. Finally, line 14 shows the until-loop.

With until-loops, users did not have to worry about writing labels and setting them as targets correctly with `goto` or `loop_lt` statements. Instead, Csound's compiler would handle rewriting the loop to use labels and gotos for them. Like the introduction of if-then, until-loops provided a more expressive way to write code that was less error-prone than using labels and gotos.

The development of until-loops mark two aspects of Csound's language evolution. Firstly, it continued the trend started by the introduction of if-then to extend Csound's language to include more structured programming constructs. Secondly, it was the first new language feature implemented using

the NewParser. It would prove to be a model for implementing new language features in future versions of Csound.

2.4.5 Summary

The NewParser in Csound5 provided a new implementation of the Csound Orchestra language. Using Flex and Bison to generate the lexer and parser, the NewParser provided a clearer specification of the language that was easier to maintain and develop. The NewParser provided core developers new opportunities to extend the language, as with the implementation of until-loops, as well as to extend the system, as with the parallel implementation of Csound. This work would provide the foundation for new language changes in future versions of Csound.

2.5 Conclusions

In this chapter, I have looked at the evolution of the Csound Orchestra language from its beginning through Csound 5. I started with an analysis of the early Csound language, providing a baseline of language syntax and features.

In Csound 4, the development of if-then marked the introduction of structured programming into the language of Csound. It would change how users could express themselves in the language. The introduction of subinstruments and user-defined opcodes marked the beginnings of growing user-extensibility. With the ability to define UDOs within the Orchestra language itself, users now had new kinds of things they could express in their code.

In Csound 5, the introduction of the NewParser provided a new language interpreter implementation. This laid the groundwork for future language developments and represents better developer extensibility. The introduction of until-loops brought another structured programming feature and was built with the NewParser. This provided users with further ways to express their code. The implementation of until-loops also served as a model of defining new language features using the new infrastructure.

The Csound Orchestra language has grown over time, both in its design and in its implementation. The kinds of changes shown for Csound 4 and 5 demonstrate ways to develop features to enable others to grow the system themselves. These aspects of extensibility found in the historical development of Csound's language serve as the foundation of the original work discussed in Chapter 3.

Chapter 3

Evolving the Language of Csound

This chapter will look at new features of the Csound [35] Orchestra language developed for this thesis. It will begin with an overview, then discuss language design and implementation changes introduced in Csound 6. It will then discuss further changes that are developed and will be released in Csound 7. Finally, a summary will be given for the work on Csound's language and its impact on the system's extensibility.

3.1 Overview

Chapter 2 discussed the origins and history of Csound's Orchestra language. The developments in Csound 4 and Csound 5 introduced new ways for users and developers to extend the system. These changes provide a model for how to grow the language and extend the richness of Csound's language over time.

The work presented here continues to evolve the language of Csound. Like Csound 5's NewParser, changes like the new type system (Section 3.2.1) and Parser3 (Section 3.3.1) will develop the infrastructure of the language. These are changes that provide better developer extensibility, which is realised

in the implementation of new language features like arrays (Section 3.2.2) and user-defined types (Section 3.3.3). Like Csound 4, these new language features provide new ways for users to develop their work and extend the system themselves.

3.2 Csound 6

Csound 6 was a major rewrite of Csound 5. It introduced many new features, such as sample-accurate scheduling, a new type system, transactional recompilation for live coding, a new development test suite, and more. It was also the first version where the OldParser was completely removed from use and the codebase. The full set of goals and features are discussed in [67] and [68].

In this section, I will discuss original work on the Csound Orchestra language design and implementation I have done as part of Csound 6 for this thesis. This includes the new type system, new implementation of opcode polymorphism, extensions to opcode function-call syntax, introduction of array types, and runtime type identification. For each topic, I will discuss motivations, provide analyses, then discuss the design and implementation of the feature. I will also describe how these features enable greater extensibility in Csound for users and developers.

3.2.1 Type System

The new type system (hereafter *TypeSystem*) introduced in Csound6 provides a systematic way to define and work with data types used in the Csound Orchestra language. It isolates all features for a data type to a single location, which then provides a generic way to work with data types. The goal of

this work was to organise and simplify the code related to data types within Csound's codebase as well as make data types extensible for developers.

I will begin with an analysis of types and their usage in Csound. Next, I will discuss the implementation of types prior to Csound6. Afterwards, I will discuss the goals of the new TypeSystem, then discuss the implementation. I will then summarise this work.

Types in Csound

Types in Csound have two main aspects: concrete data types and specification of data types. The former describes data in terms of what the data means and how it is used. The latter defines what kinds of data types are acceptable as arguments to opcodes.

Concrete Data Types In Csound, *variables* are used to name locations in memory that store values. The meaning of that value is determined by the variable's `data type`. For example, when a variable is defined to be of type `i`, the memory allocated for the variable is understood to represent a single floating-point number.

Data types are uniquely named, such that a given type name maps to only one data type. The Csound data type maps to a C data type, which may be a primitive type (i.e. `float`) or a structure. Internally, Csound uses the C data type to determine the size for a Csound type when allocating memory. Variable memory is also cast to its corresponding C data type so that processing code (i.e., opcodes) know how to read from and write to variables.

Csound's data types have also had a concept of *rate* traditionally associated with them. The rate of a data type describes when the value of a variable is

updated. Originally, Csound had three types of variables: `i-`, `k-`, and `a`-types. Each type corresponded to a numeric value updated only at initialisation time (i.e., init-rate), once per-block of audio (i.e., control-rate), or once per-sample of audio (i.e., audio-rate), respectively. Internally, `i-` and `k`-types would map to the same C data type, a single floating-point number, and the `a`-type would map to an array of floating-point numbers with the size of the array equal to the block-size — called `ksmps` in Csound — configured for the engine.

However, the association of specific rates with individual data types has not been maintained with Csound's newer data types. For example, Csound's `S`-type represents a character string that has no specific rate associated with it. Some opcodes like `strcat` will use and process `S`-variables only at initialisation time, while others like `strcatk` will process at initialisation and performance times, once per block computation. The same `S`-type is used to describe variables processed by these opcodes even though they differ in their update rates. If Csound were to differentiate rates for character string types in the same way as it did for floating point numbers, there would be multiple data types defined for string variables, one for each rate used. Instead, the rate of the `S`-type is left ambiguous and dependent upon the opcodes used with the `S`-type variable.

Another aspect of rates with data types is further illustrated by `f`-type variables. Csound's `f` data type describes the contents of variable memory as holding information for a spectral signal (i.e., FFT data frame) and maps them internally to the `PVSDAT` data structure. The update rate of the `f`-signal is generally determined by the hop size number of samples for the FFT analysis source, which must be defined in relation to both the sample rate and the block rate for proper processing to occur. Multiple `f`-type signals

may exist in the same project and they may each have their own unique update rates (i.e., hop sizes) that are different from Csound's other rates (i.e., Csound's control and audio rates). The requirement for variable rates that are determined at run time for a data type is at odds with Csound's original practice of differentiating rates by using unique data type names at compile time.

The situation where some data types can have a hard-coded rate associated with them and others do not is possible because Csound's engine does not itself have any knowledge of rates for a data type. Internally, Csound's data types do not have a property that describes their rates. Rates are also not expressed through some form of type hierarchy such that variables of different types could be compared to each other to discover if they operate at equivalent rates. The Csound compiler only knows that there are data type names and that they map to data type definitions (i.e., the underlying C data type) that describe the variable's memory layout. The rate at which a variable actually updates is determined entirely by the opcodes that use those variables.

Although Csound's data types do not currently have rate as a first-class property, it may be an interesting area to explore further for Csound's type system. Separating rate from the primary data type itself would allow generically handling of data types by rate and enforcing certain variable usage at the level of the language and engine. This would also prevent a a proliferation of types simply to differentiate rates as was done for the `i`, `k`, and `a` types. If the three types could be recast as a single floating-point numeric type with three different rate attributes, the same process of attributing rates could be applied to other data types. Other programming languages offer a model solution to this kind of problem by using special language syntax and

universal types [138, 23. Universal Types] (i.e., types specified in terms of other types).

Listing 3.1 shows two related examples of how the issue of rates might be interpreted using the C programming language. In the first example, the three variables listed — `isig`, `ksig`, and `asig` — would correspond to Csound's `i`-, `k`-, and `a`-rate signals. The example shows three kinds of types related to the float data type: a `const float` (i.e., an immutable float), a `float`, and a `float array`. The the `const` keyword and braces are a part of the C language as special syntax that denotes that universal types — a const and array type respectively — are being used that are defined in terms of the `float` data type. The special syntax may be applied to any other data type, which allows endowing those qualities of *mutability* and *dimensionality* to the the base type.

In the second example, C's `typedef` feature provides a way to create an alias for the more verbose universal types. Applying the typedef-created types to variables shows a similarity to Csound's method of discerning rates using uniquely named types. The typedef provides a possible path to using a universal type solution to reifying rates in Csound that could support the original design of designating unique rates for certain data types.

```
// Implementation using types and modifiers
const float isig;
float ksig;
float asig[BLOCK_SIZE];

// Implementation using typedefs
typedef const float IRATE;
typedef float KRATE;
typedef float* ARATE;
```

```
IRATE isig;
KRATE ksig;
ARATE asig;
```

Listing 3.1: Possible C implementation of Csound's rates

Csound's existing data types demonstrate that the concept of rates are not a first-class property of types. Language changes and the implementation of universal types could one day be used to reify rates within the system. However, the support for first-class rates is not a requirement for implementing a type system that supports all of Csound's existing features for data types. Further exploration of rates is reserved for future work and is discussed in Section 7.2.

In addition to mapping of data type names to definitions that describe the storage format for a variable, there are *general* operations that happen to all variables regardless of type and *specific* operations that are only relevant to a single type. General operations include allocation, initialisation, and freeing of variables, as well as copying from one variable to another of the same type. These operations are performed by the engine and use information about a variable's type to process the variables. The first three operations occur as part of the standard life cycle of an instrument instance, while the last operation is performed as part of Csound's execution model for calling user-defined opcodes, where all variables are always passed-by-value.[1] Specific operations for variables occur within opcodes and outside of the main Csound

[1]Csound's handling of arguments for native opcodes written in C are always passed-by-reference and contrasts with what is done for user-defined opcodes. This disparity of argument handling will be discussed in Section 7.2.

engine. Opcodes process a variable's memory in ways specific to the domain of the data type's meaning.

-

Type-specifiers Opcodes use *type-specifiers* to describe their input and output arguments. Type-specifiers denote the permissible types allowed for an argument, whether the argument is optional (and if so, what is its default value when not present), and the number of arguments covered by that type-specifier (i.e., cardinality). Like data type names, type-specifiers are single characters.

All of the type names for concrete data types may be used as type-specifiers. If a data type name is used, it specifies that the type-specifier refers to a single argument, it allows only arguments that match the given data type, and it is a required value. Additional single-character type-specifiers are defined in Csound. For example, the type specifier `T` denotes a required, single argument that may be of type `S` or `i`. The `o` type specifier denotes an optional `i` type argument with a default value of 0 (`p`, `q`, `v`, `j`, and `h` denote the same but with default values of 1, 10, 0.5, -1, and 127 respectively). The `z` type specifier is used to denote an indefinite number of `k`-type arguments.

Type-specifiers intertwine a number of aspects about opcode arguments all into a single-character name. This includes allowable types, optionality and default value, and cardinality. Because these qualities are all encoded within the single-character, the variations in qualities leads to the situation where many different type-specifiers may be used for a single type, differing by only one quality or another. The process of adding new type-specifiers and variations can occur for each new type added to Csound. It is not difficult to

imagine the available single-characters for type-specifiers becoming quickly exhausted as the number of data types grow.

In other programming languages, these qualities of argument are usually separated out from the type given for an argument. This may be done with special syntax or qualifiers added to the argument. Listing 3.2 shows an example in the Python language of defining a function with default (i.e., optional) arguments as well as variadic arguments. Here, `arg1` is an optional argument that defaults to the value 2.0, and `otherargs` denotes zero to many other arguments may be used. Note that use of equals and asterisk are additional modifiers to the standard argument specification.

```
# Definition of my_function
def my_function(arg0, arg1=2.0, *otherargs):
  print(arg0, arg1, otherargs)

# Example usage
>>> my_function(1)
1 2.0 ()
>>> my_function(1, 5.0)
1 5.0 ()
>>> my_function(1, 5.0, 3, 4, 5)
1 5.0 (3, 4, 5)
```

Listing 3.2: Example of default and variadic arguments in Python

For the aspect of type-specifiers that specify allowable types, the type-specifier acts like a *variant* and not an *abstract data type* (ADT). Pierce defines variants [138, 11.10 Variants] as a generalised form of *sum* types [138,

11.9 Sums], where the value of a variant can be one of a fixed set of types.[2] With variants, the data type does not know about the variant, but the variant knows about the data type.

Liskov and Zilles describe ADTs as "a class of abstract objects which is completely characterized by the operations available on those objects" [117]. With ADTs, concrete types are defined as being or implementing an ADT. Here, the concrete type must support being used as an instance of an ADT, thus the type knows about the ADT. This implies a relationship between the concrete types and that they share some aspect of meaning and behavior.

Initial attempts at analysing Csound's type-specifiers using an object-oriented approach failed when encountering the `T` type-specifier, which permits either an `i`- or `S`-type to be used as an argument to an opcode. These two types do not share qualities that would allow expressing `T` as an ADT that `i`- or `S`-types implement. However, framing `T` as a variant does successfully express the meaning and usage of `T` within the system.

The design of types and type-specifiers based around a single-character would have been practical in early versions of Csound when there were only three data types available. As more types were introduced, the design showed limitations in extensibility and expressiveness.

Implementation of Types Prior to Csound6

Prior to Csound6, Csound did not have a *system* for working with types. One could not programmatically define a type, nor could one query a system for

[2]Variants are usually implemented by *tagging* a data structure. In Csound, Runtime Type Information (RTTI, described in 3.2.5) is used to determine the actual data type of an argument.

all known types. The concept of types existed, but there was no concrete representation of types within the program.

Instead, types were a convention based around the single-letter name of a type. To understand the type of a variable, code would have to analyse the variable's name for the type. This would be done every time a variable was used, as the system also did not reify the concept of a variable into a concrete data structure. Instead, the name of a variable was all that was tracked during compilation.

For example, if an area of code allocated memory for a variable, it would receive as an argument the name of the variable. If the name was `ivar`, the first-letter would be used to determine it was of type `i`. The code would then use if-else branches, checking the found type against known types, then process and allocate memory. If a variable was used elsewhere, it was again passed as just a name, the type would again be analysed, and another if-else ladder of conditions would be run based on the found type.

The result was that code related to data types was spread out in numerous places. To allocate a variable of a given type, the allocation code would have to know what each type meant and what to do with it. This was the same for initialising, copying, and freeing of variables, as well as tracking of variables during compilation. In each of these areas of code, the specific knowledge of a type was required.

Additionally, the use of type-specifiers was also by convention. One could not define a new type-specifier except by updating the code that processed them. This was less of a problem than for data types, as type-specifier validation was isolated to one specific area of the compiler. However, by not

having type-specifiers defined concretely, the meanings and usage of specifiers as a whole was obscured.

With the code prior to Csound6, it was difficult to understand how to define new types as well as understand where and how general processing of variables occurred. The lack of concretely-defined data types made introducing new data types both difficult for core developers and impossible for third-party developers. The implementation of type-specifiers and their processing was similarly difficult to understand and extend.

Goals of the New TypeSystem

The goals for the new TypeSystem were that it should be able to handle the same data types found previously, support the new array type (see Section 3.2.2) in Csound 6, and provide extensibility for the introduction of new types. It should also isolate all type-specific code and properties to a single location and, where possible, remove all type-specific code from the areas of the codebase. This should make defining data types an explicit process, simplify the code base, and prepare Csound for easier modification for future type-related changes.

Redesigning types in Csound was outside the scope for the initial implementation of the TypeSystem. The use of single-letter type names and the argument type specifications for opcodes were to be maintained. All previous opcodes should operate without modification to their code or their definitions (`OENTRYs`). Finally, all expected behavior based around types and type-specifiers was also to be maintained.

Implementation

The implementation of the TypeSystem required defining new data structures to represent Csound data types and variables, then updating code to use these new data structures.[3] It also required redefining all previous standard data types within the new TypeSystem. The following will discuss these implementation changes as well as modifications performed related to type-specifiers.

Defining Data Types In Csound 6, variable types are concretely defined by creating an instance of the `CS_TYPE` data structure and registering it with the `CSOUND` engine. The `CS_TYPE` defines all common aspects of a data type. Registering it with the engine makes that type available for use in the system. Listing 3.3 shows the C code for `CS_TYPE`.

```
// include/csound_type_system.h:40
typedef struct cstype {
  char* varTypeName;
  char* varDescription;
  int argtype; // used to denote if allowed as in-arg,
   out-arg, or both
  struct csvariable* (*createVariable)(void*, void*);
  void (*copyValue)(void* csound, void* dest, void* src);
  struct cstype** unionTypes;
  void (*freeVariableMemory)(void* csound, void* varMem);
} CS_TYPE;
```

Listing 3.3: Definition of CS_TYPE struct in Csound 6

[3]The header files that define the basic type system code are include/csound_type_system.h and include/csound_standard_types.h. The implementation files are found in Engine/csound_type_system.c and Engine/csound_standard_types.c.

Each of the fields in `CS_TYPE` was defined to capture some aspect of data types that was already in use in the pre-Csound 6 codebase. The `varTypeName` corresponds to the single-letter name of the data type. This uses a `char*` instead of `char` to eventually support multi-character type names. `createVariable`, `copyValue`, and `freeVariableMemory` are all pointers to functions for creating variables, copying values from one variable of the same type to another, and freeing the memory for a variable. These are the primary fields in use.

Other fields shown were a part of the Csound6 design but not used. `varTypeDescription` provides a human-readable description about the type. `argType` and `unionTypes` were designed for using `CS_TYPEs` to define information for use as type-specifiers. These fields were retained for future exploration.

In Csound 6, all of the standard concrete types in Csound were defined using `CS_TYPE` definitions. Listing 3.4 shows the relevant code for the definition of the `f`-type and is an example of how one defines a data type in the TypeSystem.

```
// Engine/csound_standard_types.c:47
void fsig_copy_value(void* csound, void* dest, void* src) {
  PVSDAT *fsigout = (PVSDAT*) dest;
  PVSDAT *fsigin = (PVSDAT*) src;
  int N = fsigin->N;
  memcpy(dest, src, sizeof(PVSDAT) - sizeof(AUXCH));
  if(fsigout->frame.auxp == NULL ||
     fsigout->frame.size < (N + 2) * sizeof(float))
    ((CSOUND *)csound)->AuxAlloc(csound,
        (N + 2) * sizeof(float), &fsigout->frame);
  memcpy(fsigout->frame.auxp, fsigin->frame.auxp,
```

```
      (N + 2) * sizeof(float));
}

// Engine/csound_standard_types.c:197
CS_VARIABLE* createFsig(void* cs, void* p) {
  CSOUND* csound = (CSOUND*)cs;
  CS_VARIABLE* var = csound->Calloc(csound,
      sizeof (CS_VARIABLE));
  IGN(p);
  var->memBlockSize = CS_FLOAT_ALIGN(sizeof(PVSDAT));
  var->initialiseVariableMemory = &varInitMemory;
  return var;
}

// Engine/csound_standard_types.c:308
const CS_TYPE CS_VAR_TYPE_F = {
  "f", "f-sig", CS_ARG_TYPE_BOTH, createFsig,
  fsig_copy_value, NULL, NULL
};
```

Listing 3.4: Definition of `f` data type

When a Csound engine is created and initialised, an empty `TYPE_POOL` struct, shown in Listing 3.5, is also created, then populated with the standard Csound types. The `TYPE_POOL` holds the head of a singly-linked list of `CS_TYPE_ITEMs`, which in turn hold instances of `CS_TYPEs`. New Csound API functions are provided for adding new data type definitions, creating `CS_VARIABLEs` from types, and querying for types by name.

```
// include/csound_type_system.h:81
typedef struct cstypeitem {
  CS_TYPE* cstype;
  struct cstypeitem* next;
```

```
} CS_TYPE_ITEM;

typedef struct typepool {
    CS_TYPE_ITEM* head;
} TYPE_POOL;

/* Adds a new type to Csound's type table
   Returns if variable type redefined */
PUBLIC int csoundAddVariableType(CSOUND* csound,
    TYPE_POOL* pool, CS_TYPE* typeInstance);
PUBLIC CS_VARIABLE* csoundCreateVariable(void* csound,
    TYPE_POOL* pool, CS_TYPE* type, char* name,
    void* typeArg);
PUBLIC CS_TYPE* csoundGetTypeWithVarTypeName(
    TYPE_POOL* pool, char* typeName);
PUBLIC CS_TYPE* csoundGetTypeForVarName(TYPE_POOL* pool,
    char* typeName);
```

Listing 3.5: Type system structures and functions

Defining Variables Variables are defined as part of instrument definitions or as global variables. Variables are defined with a name and a data type. They are used as input and output arguments to and from opcodes. They can have one of two scopes: either globally-scoped, usable from any instrument, or locally-scoped, usable only within a single instrument.[4]

Prior to Csound6, the parser would read variable names and check that they were initialised before they were used. The type of variable would be

[4]Note that UDOs are compiled internally as instruments, so local-scoping for UDOs is equivalent to an instrument's local scope and a UDO's variables can not be read or used outside of the UDO.

determined by the first character of the variable name. Each time a variable was defined, the type would be checked, a counter for that type would be incremented, and the count would be assigned to that variable's name. At compile time, a calculation would be made that multiplied each count by the size of the variable's memory requirement (i.e., `kcounter * sizeof(MYFLT)`, `acounter * (sizeof(MYFLT) * ksmps)`), and the sum total would be used as the total size of local variable memory for new instances of instruments.

The problem with this approach is that for any new type, a new counter would need to be added to the compiler to track that type. Additionally, the memory requirements for that type would also have to be hardcoded in the area of code that calculates local variable memory size. While effective when there were few variable types in Csound, the system was difficult to extend and became more difficult to understand as more types were added.

In Csound 6, variables are defined using `CS_VARIABLEs` and tracked in `CS_VAR_POOLs`. When the parser reads a variable name, if it is a definition of a variable, the name of the variable's type – determined by the first character – is used to find a registered `CS_TYPE`. Once a `CS_TYPE` is found, a `CS_VARIABLE` is created from the `CS_TYPE` and added to the appropriate local or global `CS_VAR_POOL`. Listing 3.6 shows the definitions for `CS_VARIABLE` and `CS_VAR_POOL`.

```
// include/csound_type_system.h:57
typedef struct csvariable {
  char* varName;
  CS_TYPE* varType;
  /* memBlockSize must be a multiple of sizeof(MYFLT), as
     Csound uses MYFLT* and pointer arithmetic to assign
     var locations */
  int memBlockSize;
```

```
  int memBlockIndex;
  int dimensions;  // used by arrays
  int refCount;
  struct csvariable* next;
  CS_TYPE* subType;
  void (*updateMemBlockSize)(void*, struct csvariable*);
  void (*initializeVariableMemory)(struct csvariable*,
      MYFLT*);
  CS_VAR_MEM *memBlock;
} CS_VARIABLE;

// include/csound_type_system.h:106
typedef struct csvarpool {
  CS_HASH_TABLE* table;
  CS_VARIABLE* head;
  CS_VARIABLE* tail;
  int poolSize;
  struct csvarpool* parent;
  int varCount;
  int synthArgCount;
} CS_VAR_POOL;

PUBLIC CS_VAR_POOL* csoundCreateVarPool(CSOUND* csound);
PUBLIC void csoundFreeVarPool(CSOUND* csound,
    CS_VAR_POOL* pool);
PUBLIC char* getVarSimpleName(CSOUND* csound,
    const char* name);
PUBLIC CS_VARIABLE* csoundFindVariableWithName(
    CSOUND* csound, CS_VAR_POOL* pool, const char* name);
PUBLIC int csoundAddVariable(CSOUND* csound,
    CS_VAR_POOL* pool, CS_VARIABLE* var);
PUBLIC void recalculateVarPoolMemory(void* csound,
```

```
    CS_VAR_POOL* pool);
PUBLIC void reallocateVarPoolMemory(void* csound,
    CS_VAR_POOL* pool);
PUBLIC void initialiseVarPool(MYFLT* memBlock,
    CS_VAR_POOL* pool);
```

Listing 3.6: Variables structures and functions

Instantiating Variables The information collected in local and global variable pools is used to determine memory to allocate at runtime for instrument instances and global variable memory. The two paths are handled in two different manners, as each has different requirements.

For instrument instances, Csound calculates the total memory required for the entire instance and allocates it as one large-block of memory. This memory is then subdivided into various parts: some parts are used to represent state data for opcodes, others are used to represent variable memory.

When an instrument instance is created, `insprep()` is called, which in turn calls the new `recalculateVarPoolMemory()` function. The latter function performs two tasks. Firstly, it calculates the sum total memory requirements of all variables in the pool and records the `poolSize`. The `poolSize` is used as part of the calculation for the total memory of an instance of an instrument. Secondly, it calculates indexes for each variable to use for assigning memory. The indexes are used as offsets from the base address of the location designated for variable memory within the total instrument instance memory.

For global memory, variable memory is allocated with a different strategy. Instead of creating a large block of memory that is then divided – the strategy prior to Csound 6, and the one used for instruments – memory is allocated for each variable individually. The variable memory is assigned to the `memBlock`

field directly in `CS_VARIABLE`. When global variable memory pointers are assigned to opcode arguments, they will point to the memory held in the `memBlock` field.

Initialisation, Copying, and Freeing The other behavior of variables besides allocation are initialisation, copying, and freeing. While all types share these behaviors, the exact process that happens is type-specific. Prior to Csound6, different areas of code would handle these aspects of variables, and type-specific processing code would be repeated in each of these areas.

With Csound6, these areas of code were found and modified. Instead of looking at just the type name for a variable to determine what to do, code could now look at the variable's `CS_TYPE`. From there, the appropriate function pointer would be used to perform the type-specific operation. By modifying the codebase to work with `CS_TYPEs`, any new types would automatically work with the rest of the system without requiring changes to the codebase.

Type-specifiers Prior to Csound6, code in the compiler would directly compare each of found types for arguments in large switch-statements to see if they matched specific type specifiers. If so, then code for processing optionality and cardinality were done inline. The specification of a type specifier then was directly mixed in with its processing implementation.

For Csound6, type-specifier code was rewritten to better express the intention of each specifier. Firstly, the kinds of specifiers were specifically defined in a separate location than their use and organised by the specifier's qualities. Listing 3.7 shows the definitions of `POLY`, `OPTIONAL`, and `VAR_ARG` specifiers, together with their allowed types.

// Engine/csound_standard_types.c:358

```
const char* POLY_IN_TYPES[] = {
    "x", "kacpri",
    "T", "Sicpr",
    "U", "Sikcpr",
    "i", "cpri",
    "k", "cprki",
    "B", "Bb", NULL};
const char* OPTIONAL_IN_TYPES[] = {
    "o", "icpr",
    "p", "icpr",
    "q", "icpr",
    "v", "icpr",
    "j", "icpr",
    "h", "icpr",
    "O", "kicpr",
    "J", "kicpr",
    "V", "kicpr",
    "P", "kicpr", NULL
};
const char* VAR_ARG_IN_TYPES[] = {
    "m", "icrp",
    "M", "icrpka",
    "N", "icrpkaS",
    /* 'n' requires odd number of args... */
    "n", "icrp",
    "W", "S",
    "y", "a",
    "z", "kicrp",
    /* 'Z' needs to be ka alternating... */
    "Z", "kaicrp",  NULL
};
const char* POLY_OUT_TYPES[] = {
```

```
    "s", "ka",
    "i", "pi", NULL
};

const char* VAR_ARG_OUT_TYPES[] = {
    "m", "a",
    "z", "k",
    "I", "Sip",
    "X", "akip",
    "N", "akipS",
    "F", "f", NULL
};
```

Listing 3.7: Type-specifier definitions

Next, code that verified found arguments and opcode argument type specifications was updated. Switch-statements were removed and replaced with function calls that checked if a found argument's type, read from the variable's `CS_TYPE`, matched against one of the specifier definitions. By using functions with names like `is_in_optional_arg` or `is_in_var_arg`, the intention of code was easier to see and understand.

This code change in Csound6 improved the situation in understanding the use of type-specifiers. It explicitly defined type-specifiers, separately from their usage, and classified them by their intent. However, while these changes improved the situation, more could be done. The original design was to use `CS_TYPEs` to define variant types and do type checking using those. However, this would have only addressed that aspect of type-specifiers and not optionality and cardinality. To address these qualities, it was determined that changes may be required both to the type specification format for opcodes as well as possibly opcode definitions themselves. Instead, the above was

implemented, and the focus of Csound6's type system was limited to data type definition. Development of the type specification system is reserved for future work.

Summary

The new TypeSystem in Csound6 provides a formalised system for creating Csound data types. By explicitly defining data types using C data structures, the TypeSystem provides a means by which code can work with with variables generically through their types as well as make data types extensible by third-party developers. The implementation has isolated type-specific code in Csound and has arguably clarified the use of types in the rest of the codebase.

3.2.2 Arrays

For Csound6, a feature that was requested by the user community was the implementation of array types. Arrays are sets of data of type x, where x can be any non-array type. Arrays are useful for creating and operating on sets of values. For example, they can be used to create multi-channel audio signals or to hold analysis bin data for FFTs.

The following will begin by describing the specification for arrays. Example code will follow that demonstrates array usage by users in Csound Orchestra code. Next, the implementation of arrays will be described in detail. Finally, a conclusion will discuss the impact of arrays for users and their works.

Specification

Arrays in Csound have the following properties: they are *generic*, they are *homogeneous*, and they are *multi-dimensional*. Firstly, arrays are *generic*.

This means that arrays can be created of any type x. Arrays then are a type that are specified in terms of other types (i.e., a universal type). The implementation of Csound arrays then must require that a variable's type include both that it is an array and what type of array it is. Note that while all arrays share the same operations, arrays of different types are not equivalent. It is invalid to try to use an array of type x where an array of type y is expected.

Secondly, arrays in Csound are homogeneous. Each element of the array is of the same type x. This requirement allows for each member to be the same size in memory. When an array is created, one large block of memory can be allocated for the entire array that is equal to the number of elements multiplied by the size of the x. Homegenous memory layout allows efficient, constant-time access to members of the array.

Finally, arrays in Csound are multi-dimensional. Each dimension has a fixed size when the array is first created. The total number of elements in the array is equal to the product of the sizes of all dimensions. For example, for a 2-dimensional array with sizes 2 and 4, the total size of the array is 8.

Example Array Code

Listing 3.8 shows example Csound Orchestra code that uses a single-dimension array. Line 3 shows the declaration of an `i`-type array called `iarray`. It it initialised to size `isize`, which iresults in `iarray` having 10 elements. Next, starting in line 5, an until-loop is used to iterate and set each member of the `iarray` with a value calculated from the `indx` variable. Each value is also printed out using the `print` opcode. Lines 6 and 7 shows left-hand side expressions being used within the array access notation (described further in

Section 3.2.2) to calculate the index into the array for setting a value. Lines 9 and 10 show right-hand side expressions being used to access members from the array for printing.

```
1  indx = 0
2  isize = 10
3  iarray[] init isize
4
5  until (indx >= isize) do
6    iarray[indx] = indx
7    iarray[indx + 1] = indx
8
9    print iarray[indx]
10   print iarray[indx + 1]
11
12   indx += 2
13 od
```

Listing 3.8: Csound array example

The output of running the example in Listing 3.8 is shown in Listing 3.9.

```
instr 1:  #i4 = 0.000
instr 1:  #i6 = 0.000
instr 1:  #i4 = 2.000
instr 1:  #i6 = 2.000
instr 1:  #i4 = 4.000
instr 1:  #i6 = 4.000
instr 1:  #i4 = 6.000
instr 1:  #i6 = 6.000
instr 1:  #i4 = 8.000
instr 1:  #i6 = 8.000
```

Listing 3.9: Csound array example output

Note that the use of empty brackets is only for the declaration of the array variable. Declaration, like for non-array variables, is done at the first time a variable is used on the left-hand side of an opcode. Once a variable is first defined, its type is registered in the `CS_VAR_POOL`. Afterwards, references to the array need only use the variable's name without empty brackets. This is seen in Listing 3.10, where the array is passed as an argument to the lenarray [181] opcode.

```
;; initialise 2 member i-type array
ival[] init 2
ival[0] = 1
ival[1] = 2

;; Note, only using variable name
ilen = lenarray(ival)
```

Listing 3.10: Csound array use without empty brackets

To create and use multi-dimensional arrays, additional sizes can be given to the init opcode. Listing 3.11 shows a variable called `iarray`. This array is of type `i[][]` and is initialised to two dimensions of size 2 and 4.[5] Two nested until-loops are then used to set and print the contents of the array.

```
iarray[][] init 2, 4
indx = 0

until (indx >= 2) do
  indx2 = 0
  until (indx2 >= 4) do
    iarray[indx][indx2] = (indx + 1) * indx2
```

[5]For comparison, this would be equivalent to creating a multi-dimensional float array in C using "float iarray[2][4]".

```
    print iarray[indx][indx2]
    indx2 += 1
  od
  indx += 1
od
```

Listing 3.11: Csound multi-dimensional array example

The output of running Listing 3.11 is shown in Listing 3.12.

```
instr 1:  #i5 = 0.000
instr 1:  #i5 = 1.000
instr 1:  #i5 = 2.000
instr 1:  #i5 = 3.000
instr 1:  #i5 = 0.000
instr 1:  #i5 = 2.000
instr 1:  #i5 = 4.000
instr 1:  #i5 = 6.000
```

Listing 3.12: Csound multi-dimensional array example output

Finally, Listing 3.13 shows the use of arrays with UDOs. Here, the `sum_array` UDO takes in a single i[] array as an input argument. The code loops to sum up all values within the array and returns the accumulated value. Running instrument 1 will call `sum_array` and return the value 21.

```
opcode sum_array, i, i[]
  iarray[] xin
  indx init 0
  ival init 0
  until (indx >= lenarray(iarray)) do
    ival += iarray[indx]
    indx += 1
  od
  xout ival
```

```
endop

instr 1
  iarray[] init 3
  iarray[0] = 1
  iarray[1] = 7
  iarray[2] = 13

  print sum_array(iarray)
  turnoff
endin
```

Listing 3.13: Csound array UDO example

Array member access

Array members are accessed using an array variable's name, followed by brackets that contain an expression. The result of the expression must be either of type `i` or `k`. Using an i-type variable will cause the array access to occur at initialisation time only, and using a k-type variable will cause the access to occur at initialisation and performance times. The expression within the brackets determines the index of the array member to read or write.

Array member access is itself considered and treated as an expression. This means array access can be written and used anywhere that other expressions are allowed. Before arrays were introduced in Csound 6, expressions only existed on the right-hand side of opcode calls. In other words, expressions could be used as input arguments to opcode calls, but were not found on the left-hand side. For arrays, Csound's parser and compiler required modification to handle left-hand side expressions.

Array Implementation

Implementing generic arrays of any type in Csound required a number of changes to the compiler and engine. The following lists the order in which features were implemented. Each feature builds upon the previous one, and implementation details are described below.

1. Declare a variable of an array of type x.
2. Instantiate an array variable.
3. Modify C opcodes to work with arrays.
4. Process array member access.
5. Modify UDOs to work with arrays.

Declaring an array variable In Csound, variables are declared on their first assignment. When declared, the variables name is recorded, as is its type. For Csound, the first letter of the variable name is used for the type. However, a single letter would not be enough to declare an array, as an array requires both that it be declared an array as well as the type of array.

To handle this situation, the array declaration syntax was formed. To declare an array, a variable requires an open and close bracket to be appended to its name. For example, `iarray[]` declares an array variable with name `iarray`. The brackets at the end of the declaration tell Csound it is being declared as an array, and the first letter declares that the array is of type `i`.

To implement this, a new `arrayident` rule was added to Csound's grammar. Listing 3.14 begins with the rule for reading array identifiers. The rule for `arrayident` is recursive, which allows it to have multiple pairs of brackets

after the identifier name to accommodate multi-dimensional arrays. The `arrayident` rules is then further used as the return value in the `statement` rule as well as part of the `ans` rule. These two changes allow array declarations to be used as output arguments from opcodes.

```
arrayident: arrayident '[' ']'
          | ident '[' ']';

// excerpt of rule for statement
// Engine/csound_orc.y:410
arrayident '=' expr NEWLINE

// basic definition of ans
// Engine/csound_orc.y:500
ans : ident
    | arrayident
    | arrayexpr
    | T_IDENT error
    | ans ',' ident
    | ans ',' arrayident
    | ans ',' arrayexpr;
```

Listing 3.14: Array identifier syntax

After the array declaration was recognised by the parser, the compiler and runtime required changes to handle the new array type. Using the new type system discussed in Section 3.2.1, a new `CS_TYPE` was introduced that defines the generic array type (shown in Listing 3.15). In conjunction with `CS_VAR_TYPE_ARRAY`, the `CS_VARIABLE` data structure (shown in Listing 3.6) contains `subType` and `dimensions` fields. These fields are used to specify the specific aspects of the array for the variable.

In Section 3.2.1, I discussed how determining a variable's type changed from always using the first character of the variable's name to searching through a `CS_VAR_POOL` for a registered type. In large part that change was implemented to facilitate array variable definitions. Instead of determining the variable type by re-interrogating the variable name each time, the change allowed the variable's type to be determined once and looked up afterwards. If the previous lookup system was not changed, then the variable name would require having all information as part of its name everywhere it was used.

If a variable required all type information to be a part of its name every time it was used, two problems would occur with arrays. The first problem is that the use of arrays would be very verbose and difficult to read, especially when accessing members for reading and writing. For example, instead of "iarray[0] = 1", one would have to use "iarray[][0] = 1", so that the primary part of the variable name could be read to understand it is in an array.

The second problem is that parsing an array variable definition requires more time than a simple single-character lookup to understand its type. Array type processing requires finding the subtype, as well as figuring out the number of dimensions the array variable has by using the number of pairs of brackets found. Doing this each time a variable name is checked for its type would add a lot of processing for something that could be done once and looked up from a table.

Instantiation Once an array is defined as a `CS_VARIABLE`, at runtime, an instance of `ARRAYDAT` is used to hold the actual data for the array in memory. The `ARRAYDAT` structure is shown in Listing 3.15. Within this structure, the *data* field contains the actual values of each array member, while the other fields provide the required meta-data for opcodes to work with arrays.

```
// include/csoundCore.h:359
typedef struct {
  int      dimensions;
  int*     sizes;              /* size of each dimensions */
  int      arrayMemberSize;
  CS_TYPE* arrayType;
  MYFLT*   data;
} ARRAYDAT;

// Engine/csound_standard_types.c:326
const CS_TYPE CS_VAR_TYPE_ARRAY = {
  "[", "array", CS_ARG_TYPE_BOTH, createArray,
   array_copy_value, NULL, NULL
};
```

Listing 3.15: Csound 6 array implementation code

Modifying C Opcodes to work with arrays On the one hand, modifying opcodes written in C to work with arrays was simple. Opcodes could use `ARRAYDAT`s as arguments just as using any other data type – i.e., `MYFLT`, `STRINGDAT`, `PVSDAT` – in Csound. Listing 3.16 shows an example of the data structure for the array-version of the init opcode that uses an ARRAYDAT as an output argument.

```
// Opcodes/arrays.c:36
typedef struct {
  OPDS    h;
  ARRAYDAT* arrayDat; // Output argument for opcode
  MYFLT   *isizes[VARGMAX];
} ARRAYINIT;

// Opcodes/arrays.c:1914
```

```
// OENTRY for array_init opcode using the "." any type
    specifier
{ "init.O", sizeof(ARRAYINIT), 0, 1, ".[]", "m",
    (SUBR)array_init },
```

Listing 3.16: Csound 6 array init opcode

However, two problems did present themselves when defining opcodes. Firstly, the input and output argument specification did not work with multi-character type specifications. Secondly, there was no type specifier available to specify an `any` type, which would allow any kind of type to be used. This would be necessary for certain opcodes to allow working with any kind of array generically.

To address these concerns, the type specification and processing code for opcode input and output types was modified. First, rather than search for one character at a time through the argument strings, the code would look-ahead to see if any brackets were found. If so, the code would identify the argument as an array and then continue advancing through each pair of brackets found to determine how many dimensions would be required for that array argument. Second, to handle the `any` type, the "." character was introduced as a valid type-specifier and related processing code was updated. With these changes, opcodes could now specify arguments of arrays of a specific type as well as of `any` type.

Process array member access Array member access involves two scenarios: reading from arrays and writing to arrays. Reading from arrays is done when an array access is found on the right-hand side of an opcode and used as an input argument. Writing to arrays is done when an array access is found on the left-hand side of an opcode.

For the parser, array accesses are treated and processed as expressions. Like other expressions, when a statement is found that contains expressions, the statement is rewritten into multiple opcode statements containing one operation per statement. With expressions as input arguments, the generated statements would be prepended to the original statement line. The return value of the expression would be written to a synthetic variable and the variable would then be used as the input argument to the original statement.

For expressions as output arguments, the generated statement would be appended after the original statement. The original statement would first write its value to a new synthetic variable, and the synthetic variable would then be used as the input argument to the array writing statement. Note: this kind of processing for left-hand side expressions was introduced in Csound6 specifically to accommodate array member writing.

```
1  ;; BEFORE
2  indx init 2
3  iarray2[indx + 1] = iarray1[indx] + 3
4
5  ;; AFTER
6  indx init2
7  #i0  array_get iarray1, indx
8  #i1  sum       #i0, 3
9  #i2  sum       indx, 1
10      array_set iarray2, #i1, #i2
```

Listing 3.17: Compiled array access code

Listing 3.17 shows a before and after compilation of array access code. Line 3 shows code that reads an `i`-type value from `iarray1` at index `indx`, adds the number 3 to that value, then assigns that value to the `iarray2` array at index `indx` + 1. Lines 6 through 9 show the generated code after Line 3 is

processed by the compiler. Array reads are converted into `array_get` opcode calls, and array writes are converted into `array_set` opcode calls.

Modifying UDOs to work with arrays The final part of implementing arrays for Csound6 was modifying UDOs to handle processing of arrays. For UDOs, this meant updating the type specification for input and output arguments to allow specifying arrays as well as handling copying of those arguments from and to the caller of the UDO. For the first part, as UDOs shared the same argument type parsing and handling code as for C opcodes, this was already implemented. The only necessary thing to do then was to document and advertise to users how to specify array arguments for their UDOs.

For the second part, UDO argument processing required a new implementation. Previously, UDO arguments were interrogated using their single-character types, and custom code was done per type found. This was hard-coded into the UDO processing code. To handle arrays, rather than add more type checks and processing code to the existing implementation, a new strategy was implemented. Instead of the hard-coded type processing code, the `copy_value` function pointer stored in the `CS_TYPE` type definition was used. Now, when UDOs went to transfer argument values to and from the local variables, it would lookup the type of the argument and call the `copy_value` function for that type.

The result of changing the copying strategy was that not only would UDOs now work with arrays, they would also work with any type registered with the type system. This also meant that UDOs would automatically work with any future types introduced into Csound.

Summary

Arrays provide users with a new generic data type for handling sets of values, with efficient, constant-time member access. This allows new kinds of musical processing code to be written by the user. The implementation builds upon features from both the NewParser and new type system, which, arguably, validates their designs. In addition, the compiler was updated to handle left-hand side expressions, and the UDO implementation was also updated to handle arguments in a generic way.

The implementation of arrays also demonstrates are real use of the type system to implement new types. Core developers and plugins developers can now reference the above changes and use them as a model to define their own data types for Csound.

3.2.3 Opcode Polymorphism

Opcode polymorphism in Csound allows for an opcode of a given name to have multiple implementations depending upon the types of its arguments. This corresponds to Christopher Strachey's classification of *ad hoc polymorphism* [169]. With polymorphic opcodes, there are two key aspects to how they operate: how polymorphic opcodes are defined and how the correct version of an opcode is selected.

Motivations

Polymorphic opcodes have existed in Csound since at least since Csound1988. They are a fundamental part of the Csound language. However, the pre-Csound6 implementation of polymorphism had limitations. Firstly, the system was complicated to understand. This made it difficult for developers

to implement polymorphic opcodes. Secondly, due to the implementation, it was not possible for users to define their own polymorphic UDOs. This limited user extensibility. Finally, the implementation of function calls depended upon the implementation of polymorphism. This severely limited what opcodes could be used in function-call syntax.

Pre-Csound6 Polymorphism

Prior to Csound6, polymorphic opcodes were defined using a two-part system. For an opcode that only had one implementation, a single `OENTRY` was used to define the opcode in the system. For polymorphic opcodes, multiple `OENTRY` definitions were given, one for each implementation of the opcode, as well as one additional `OENTRY` that was a special marker entry. The special marker `OENTRY` would use the normal name of the opcode and used a special flag in the `dsblksize` field, a convention which marked this opcode as polymorphic. For each of the implementation entries, specially named versions of the opcode that followed a convention (described below) were used. Listing 3.18 shows an example of polymorphism using the *pow* opcode.

```
// Engine/entry1.c:124
{ "pow", 0xffff, },
// Engine/entry1.c:479
{ "pow.i", S(POW), 1, "i", "iip", ipow, NULL, NULL },
{ "pow.k", S(POW), 2, "k", "kkp", NULL, ipow, NULL },
{ "pow.a", S(POW), 4, "a", "akp", NULL, NULL, apow },
```

Listing 3.18: Csound 5 polymorphic opcode example

During compilation, for each opcode found in the user's code, the compiler would search for a corresponding `OENTRY` by name. If an OENTRY is found, it would then check if the `dsblksize` was one of the special values (shown in

Listing 3.19). If the entry was determined to be polymorphic, then a second lookup was performed to find a specific implementation based on the found arguments' types.

```
// Engine/entry1.c:78
/* If dsblksize is
      0xffff then translate on output arg
      0xfffe then translate two (oscil)
      0xfffd then translate on first input arg (peak)
      0xfffc then translate two (divz)
      0xfffb then translate on first input arg (loop_l) */
```

Listing 3.19: Csound 5 comments on polymorphic OENTRY

At this point, the original opcode name would be used as the basis for a new `opname` to search for. Depending on the type of polymorphism defined, either one or two additional type characters would be appended to the opcode name, after first appending a period. For example, if the code "ival pow 10, 2" was compiled, first, the `OENTRY` with "pow" would be found. Determining that the polymorphism was dispatched on the single output type, a new name of "pow.i" would be created, as the type of `ival` is an `i`-type. A second lookup would then search for a "pow.i" `OENTRY`. If found, the entry would be further checked if all input and output types matched, and then that opcode version would be used for compilation. This same process of appending to opcode names depending on first one or two input or output arguments types was used for each of the polymorphism types.

Criticism

The system of polymorphism in Csound prior to Csound6 was effective in providing a form of polymorphism to the language. However, the design of

the system also presented issues. In particular, because a special entry was required in addition to using special opcode names, it was not possible to create polymorphic UDOs. Also, because of the limited types of polymorphism, if future opcodes wanted to do polymorphic dispatch on more than two input or output types, it would require altering the internal engine by adding a new polymorphic type specifier (e.g., something like `0xfffa`) as well as adding additional code for synthesising the polymorphic opcode name to lookup. To address these issues, a new design was necessary.

Polymorphism in Csound 6

For Csound 6, the system of polymorphism was modified from looking for special entries to doing a type-based search for opcodes by arguments. In pre-Csound6, two lookups were done for an opcode: first by plain opcode name, then again by the new synthesised name. In Csound 6, the system does a single lookup by opcode plain name, which returns all `OENTRYs` that match that name. Next, the found types for both the input and output arguments are used to match against the specified input and output argument types for each `OENTRY`. When a match is found, that `OENTRY` is used.

To facilitate the newer system, all `OENTRYs` with polymorphic identifiers were first removed, as these were no longer used to identify polymorphism. Next, the system that registers `OENTRYs` was modified to use only the base name of an `OENTRY` (i.e., everything before a period is found). Multiple entries would now be registered for the same base opcode name.

In pre-Csound6, the registry for `OENTRYs` used a hash table where the keys were opcode names and values were indexes to `OENTRYs` in a global opcode list. This effectively allowed only one `OENTRY` per key. In Csound 6, the hash

table was changed to have values of `CS_CONS` cell lists. This allowed a list of `OENTRYs` to be registered for an opcode name.

The result of this change is that the part of opcode names after periods were no longer used by the system. However, the `OENTRYs` using names with periods were retained in the source code to act as a form of documentation. For example, in Listing 3.18, the result of registering those opcodes would be a single `pow` entry in the opcode table with three opcodes in the list as the value for that entry.

With the new polymorphism system, processing of previous example would proceed as follows. Firstly, the compiler would find the `pow` opcode was used with two input-arguments of type `i` and `i`, and one output argument of type `k`. The compiler would then do a search for the `pow` opcode and find three entries. Secondly, it would check each entry to see if types matched. Matching is first done using input argument types, then by output types. For the first entry, "pow.i", the two `i` types would match against the "iip" input argument types specified, since `p` denotes an optional input argument, but the output type "i" would not match against the found `k` output argument. This would be an invalid match, and the process would continue to the "pow.k" entry. This time, the "pow.k" entry's input argument of "kkp" would also match, as `i`-types can be used where `k`-types are specified. The output `k`-type would also match the found `k`-type. In this case, the "pow.k" opcode entry would then be used for this line of code.

Note that it was very important for the new polymorphism system to take into account both input *and* output argument types when type matching. This was important as pre-Csound6 allowed for polymorphic opcodes to be defined using the same input types but different output types. When implementing

the Csound6 system, all pre-Csound6 polymorphic opcodes were reviewed. It was determined that the new type matching rule properly resolved to the correct entry for all polymorphic opcodes from pre-Csound6. The result is that the modifications to polymorphism provided compatible language semantics with pre-Csound6, ensuring backwards compatibility, while also extending the system.

Benefits

The changes to the polymorphism system in Csound6 open up possibilities for both users and developers. For developers, defining new polymorphic opcodes is arguably simpler to do. Developers can now simply define multiple `OENTRYs` for an opcode and concern themselves only with the argument types. They would not have to also deal with marking up what arguments should be used for polymorphic dispatch, or making sure the opcode names are formatted correctly.

For users, the changes to polymorphism now allows users to write their own polymorphic UDOs. Listing 3.20 shows an example of an overloaded `add` UDO. The first version takes in an `i` and `S`-type arguments, while the latter takes the same types but in a different order. The definition of instrument 1 shows both forms of the UDO in use. The output of running the example is shown in Listing 3.21. This demonstrates that both forms of the `add` UDO are defined in the system and available for use.

```
<CsoundSynthesizer>
<CsInstruments>

sr      =       44100
ksmps   =       1
```

```
nchnls  =       2
0dbfs   =       1

opcode add, S,iS
  ival, Sval xin
  Sout sprintf "%d%s", ival, Sval
  xout Sout
endop

opcode add, S,Si
  Sval, ival xin
  Sout sprintf "%s%d", Sval, ival
  xout Sout
endop

instr 1
  ival = 2
  Sval = "TEST"
  Sout = add(ival,Sval)
  Sout2 = add(Sval,ival)

  prints(Sout)
  prints("\n")
  prints(Sout2)
  prints("\n")
endin

</CsInstruments>
<CsScore>
i1 0 0.1
</CsScore>
```

```
</CsoundSynthesizer>
```

Listing 3.20: Csound 6 polymorphic UDOs

```
new alloc for instr 1:
2TEST
TEST2
```

Listing 3.21: Csound 6 polymorphic UDO output

Note that in pre-Csound6, if the UDO definitions in Listing 3.20 were used, the second definition would have replaced the first definition. This was due to there being only a one-to-one mapping between opcode names and `OENTRYs`. In Csound6, redefinition is only done if a new UDO is defined with the exact same input and output argument types.

Summary

The new Csound6 polymorphism system changed lookup of opcodes from using special markup and naming conventions to using type-based search. This simplified both how polymorphic opcodes are defined as well as how opcode entries are looked up. As a result, this made it easier for developers to implement their own polymorphic opcodes. Also, it enabled users to write their own polymorphics UDOs, a feature that was entirely new in Csound6.

3.2.4 Function-Call Syntax

Csound allows using opcodes with function-call syntax as arguments to other opcodes. However, the implementation was restricted to using opcodes with single inputs and outputs, and a separate list of approved opcodes for use as functions was maintained. The implementation also used a function resolution

system that would only work with polymorphic opcodes. Because of these limitations, only a small set of opcodes was available for use in function calls.

For Csound6, the function-call system was rewritten to expand use of function-call syntax to the vast majority of opcodes. This opened up new ways of programming Csound instruments and UDOs using a more functional programming, expression-based style. The following discusses the changes implemented to enable this new feature. This will cover multiple-arguments, opcode lookup, output argument synthesis, and function annotations.

Analysis of Prior Function-call System

The limitation of function-call syntax to a single argument is found as early as Csound1988 and is present in the OldParser implementation through Csound5. In the expression processing code, the function-call was hardcoded to use a single argument as shown in Listing 3.22.[6]

```
if (prec == FCALL && argcnt >= 1) { /* function call: */
  pp->incount = 1;                  /* takes one arg */
```

Listing 3.22: OldParser function call processing code

A new opcode name would be synthesised using the function name, a period, and the single argument's type. This format matched the pre-Csound6 opcode system for polymorphism and made function-to-opcode mapping dependent upon that convention. As mentioned in Section 3.2.3, only a limited set of opcodes were polymorphic.

Note, this same limitation was encoded into the NewParser. The expression processing code was entirely new in Csound5, but retained the same limitation

[6]Expression processing was implemented in `express.c` in the Csound1988 code.

of a single-argument. This was due to using the same polymorphic opcode lookup system as the OldParser.

MultiArgument Function Calls

The primary issue to resolve for multiple-argument function calls was opcode lookup. As described in Section 3.2.3, opcode lookup was modified as a whole to handle the new polymorphism system. This had the effect that function-call processing no longer depended upon specially formatted opcode names.

With that in place, the rest of the implementation required modifying all of the parts leading up to the opcode lookup. Firstly, the grammar was modified so that functions could take in a list of expressions as their arguments, rather than just a single expression. Secondly, the compiling code was modified to look for multiple arguments. It is here that opcode lookup was modified to use the new system with two key differences: only input argument types were used, and only opcodes with single outputs were allowed. As function-calls are processed as expressions, the output argument is synthesised and further used as an input into the calling opcode. This was done using the output type of the found opcode.

Output argument synthesis

Output argument synthesis using the new opcode lookup system was initially problematic. Because opcodes could be polymorphic on output types, it was possible to create opcodes with the same input types but different output types. The previous limitations on function-calls prevented this ambiguity as there were no polymorphic opcodes with single arguments that only differed

on output type. However, with generalised lookup for function-calls, opcodes with multiple input arguments and polymorphic only on output type were found.

To address the ambiguous opcode scenario in Csound6, the opcode lookup was done in such a way that the first opcode that matched the input argument types would be the one used for a function-call. This rule was chosen for two reasons. Firstly, this was the simplest system to implement, and, in testing, the found opcode was most often the one that matched user expectations. Secondly, for the opcodes that were documented to be used as functions in pre-Csound6, the rule was determined to correctly resolve the same exact opcode that would have been used in pre-Csound 6; thus, the new rule preserved backwards compatibility.

Implementing more advanced type inference for what opcode to choose depending on the context of the code was seen as a desirable feature. However, implementing type inference would have required many other changes to the Csound Orchestra implementation. This remains an area to research in future versions of Csound and is discussed further in Section 7.2.

Function annotations

While the new rule for opcode lookup and argument synthesis generally works to find a suitable version of an opcode, it does not cover situations where a user would want to use a matching version of an opcode other than what is found by default. To address this new requirement, a new feature called *function annotations* was implemented. This allows the user to explicitly specify what form of an opcode to use in function-call syntax.

For example, Listing 3.23 shows two calls to the oscil opcode that uses the same input arguments. The first call returns a `k`-type variable, while the second call returns an `a`-type variable.

```
ksig oscil 0.5, 440, 1
asig oscil 0.5, 440, 1
```

Listing 3.23: Polymorphism on output type

In the opcode-call syntax, the output arguments are already provided by the user and their types can be used to disambiguate which opcode to use. However, with function-call syntax, the output argument is synthesised. For function-call syntax, I worked together with Victor Lazzarini on the Csound6 design to specify type annotations for function-calls. In this system, for a function call where a specific output type was desired, the user can annotate the function name with a specified output type. Listing 3.24 shows an example of explicitly requesting that that `vco2` opcode that outputs an `a`-type variable be used, as well as specifically requesting that `k`-type output be used with the `adsr` opcode. The generated equivalent in opcode-call syntax is also shown.

```
;; function call with type annotation
asig = vco2:a(1, 440) * adsr:k(0.1, 0.1, 0.9, 0.1)

;; generated opcode calls
#a0   vco2  1, 440
#k0   adsr  0.1, 0.1, 0.9, 0.1
#a1   mul   #a0, #k0
asig  =     #a1
```

Listing 3.24: Csound 6 function-call syntax

Summary

Function-call syntax was extended to allow using opcodes with more than one input argument. With an updated opcode lookup system, many more opcodes were now available for use as function-calls. To support this work, output argument synthesis was modified to use a found opcode's output argument type. In addition, function annotation was implemented to provide users the ability to explicitly choose what version of an opcode to use.

The result of these changes is that users can now program their works using a more functional programming, expression-based syntax. As this change was additive, existing code continues to function, and users can opt to use existing practices or avail themselves of the newer programming style.

3.2.5 Runtime Type Identification

Runtime Type Identification (RTTI) is the ability to identify the type of a variable at runtime. The facility for RTTI exists in numerous languages. Examples for C++, Java, Python, Ruby, and Common Lisp are shown in Table 3.1.

Language	Syntax
C++	typeid(*ptr)
Java	obj.getClass()
Python	type(obj)
Ruby	obj.class
Common Lisp	(type-of v)

Table 3.1: Runtime Type Identification in various programming languages

In general programing languages such as those given in Table 3.1, RTTI is useful to conditionally branch to perform work, depending on what type of data has been given to a function. In Csound, opcodes are written using RTTI so that one implementation of a function can be used to cover multiple combinations of argument types. This would be the case if the same C function was used with multiple `OENTRYs` for an opcode.

However, the pre-Csound6 RTTI system was problematic. The system only worked to discern two types – `a` and `S` – and the implementation code was not clear to read, maintain, or use. In Csound6, with the introduction of the new type system, a more formal implementation of RTTI was introduced. This simplified RTTI as a whole as well as made it work for all types found in Csound.

The following will start by discussing the evolution of RTTI in Csound prior to Csound6. Next, it will discuss the technical limitations of the pre-Csound6 RTTI implementation. Finally, the new implementation in Csound6 will be described.

Evolution of RTTI pre-Csound6

Csound1988 In Csound1988, a field in `OPTXT` called `xincod` was used to track the type of input arguments to an opcode. This value was an integer that was used both as a bit-flag and as an index. When the parser read in Orchestra code, for each opcode statement, it would set the the second or first bit of `xincod` to 1 if the corresponding first or second argument for the opcode was of type `a`. This marking would end up with `xincod` being a value between 0 and 3.

At the time, opcodes were defined in `ENTRY` data structures.[7] `ENTRYs` had up to four `aopadr` function pointers defined. This allowed one opcode ENTRY to specify `x` types for its input argument types, where the `x` would mean "k- or a-type argument". The four `aopdr` functions would then correspond to the four variations allowed for the two "xx" arguments: `kk`, `ka`, `ak`, or `aa`. This allowed a single opcode like `oscil` to be specified once, but accommodate working with different variations of input types.

At runtime, when a new instance of an instrument was created, the `xincod` value would be used as an index into the `ENTRY`'s `aopadr` array to determine which function to use. Listing 3.25 shows the definition of the `ENTRY` data structure with its `aopadr` array. It is followed by the `ENTRY` for `oscil`, which uses `x`-types and defines multiple `aopadr` functions. Next shows the code in `rdorch.c` that shows the marking of `xincod` in the parser. Finally, the code from `oload.c` shows the use of `xincod` to determine which performance function from the aopadr array to use for an opcode.

```
// cs.h:101
typedef struct entry {
  char  *opname;
  int dsblksiz;
  int thread;
  char  *outypes;
  char  *intypes;
  SUBR  iopadr;
  SUBR  kopadr;
  SUBR  aopadr[4];
} ENTRY;
```

[7] This would later become `OENTRY`.

```
// entry.c:195
{ "oscil",  S(OSC), 11, "s",  "xxio", oscset, koscil,
   osckk, oscka, oscak, oscaa},

// rdorch.c:392
if (tfound == 'a' && n < 2)
  tp->xincod += 2-n;

// oload.c:211
else opds->opadr = ep->aopadr[ttp->xincod];
```

Listing 3.25: Use of xincod field in Csound (1988)

Csound5 By the time of Csound5, the usage of `xincod` had changed. Firstly, `xincod` was still used as a bit-flag, but all input arguments were marked whether they were an `a`-type or not, up to the size of `xincod`. The first bit would correspond to the first argument, the second bit to the second argument, and so on. This allowed the first sixteen arguments to be marked as `xincod` was a 16-bit integer.

Secondly, the use of an `aopadr` array was abandoned. This was largely to accommodate the use of more than two arguments that might be of `a`-type. If the previous system was maintained, the potential number of variations would require an `aopadr` array to be of size 65536 and many functions to be implemented.

Instead, an `aopadr` field was added to `OPTXT` that could be dynamically set at runtime. Also, functions were modified to read in `xincod` at runtime to use different branches of code depending on if an argument was an `a`-type or not. Listing 3.26 shows an example use of `xincod` in Csound5. Here, the `buzz` opcode is specified with an input argument specification of "xxkio". During

performance, at init-time, the `buzz` opcode's init function performs a check of `xincod` and caches whether its `amp` and `cps` arguments were set to `a`-type arguments or not. At performance-time, the `buzz` opcode's performance function would first do a single calculation as-if an argument was a scalar value (i.e, `k`- or `i`-type), but then check if arguments were `a`-type within its performance loop and conditionally do further processing.

```
// Engine/entry1.c:399
{ "buzz", S(BUZZ), TR|5, "a", "xxkio", bzzset, NULL, buzz },

// rdorch.c:1945
// csound_orc_compile.c:313
if (tfound == 'a' && n < 31) /* JMC added for FOG */
         /* 4 for FOF, 8 for FOG; expanded to 15  */
  tp->xincod |= (1 << n);

// H/csoundCore.h:115
#define XINCODE     ORTXT.xincod
#define XINARG1   (p->XINCODE & 1)
#define XINARG2   (p->XINCODE & 2)

// OOps/ugens4.c:38
p->ampcod = (XINARG1) ? 1 : 0;
p->cpscod = (XINARG2) ? 1 : 0;

// OOps/ugens4.c:83
if (p->ampcod)
  scal = *++ampp * over2n;
if (p->cpscod)
  inc = (int32)(*++cpsp * sicvt2);
```

Listing 3.26: Use of xincod field in Csound 5

In addition to `xincod`, type tracking was extended to use three other fields: `xoutcod`, `xincod_str`, and `xoutcod_str`. Each of these were used as bit-flags in the same way `xincod` was used. These fields were used to track if output arguments were an a-type, or if input or output arguments were `S`-types, respectively.

Analysis The system of `xincod`, `xoutcod`, `xincod_str`, and `xoutcod_str` for tracking type information was useful for simplifying opcode writing. However, looking towards the future, the system of using bits and adding new bit-flag fields per-type would not scale. If a developer wanted to add tracking for new types, they would have to add new fields and add further code for checking and setting bit-flags. Also, the bit-flag system required that the system itself have prior knowledge of the type it was trying to track. This would be an impossible situation to track types defined in a third-party plugin using the new type system. Due to the limitations above, as well as the difficulty in understanding the code, a new system was devised.

RTTI in Csound 6

In Csound 6.04, I introduced a new system for generic RTTI for any opcode input or output argument. All bit-flags and tracking code were removed. In its place, all variables used in Csound had their corresponding `CS_TYPE` set in memory at a negative offset from the data pointer. This change affected how memory was calculated and laid out for instrument instances as well as how the Csound channel database was allocated and managed. Also, Csound API methods were added to simplify interrogating an argument's types.

A comprehensive discussion of memory layout for instruments, variables, and opcodes was presented in [202]. That work was produced before this

work for RTTI was introduced. Figure 3.1 shows the layout of memory for an instrument instance pre-Csound6. Memory is laid out in three main regions: the instrument header (an INSDS data structure), the variable memory space, and the opcode memory space. For instrument instances, the sum total of variable memory for an instrument instance was previously calculated as total of the sizes for each variable's type. For example, `k`-type arguments are defined as a single `MYFLT` instance.[8] If an instrument had 3 `k`-type variables, each instrument instance would allocate `3 * sizeof(MYFLT)` amount of memory for the variables used. The memory allocated for the variables would then be partitioned with the address of each partition assigned to the input and output argument pointers for each opcode instance.

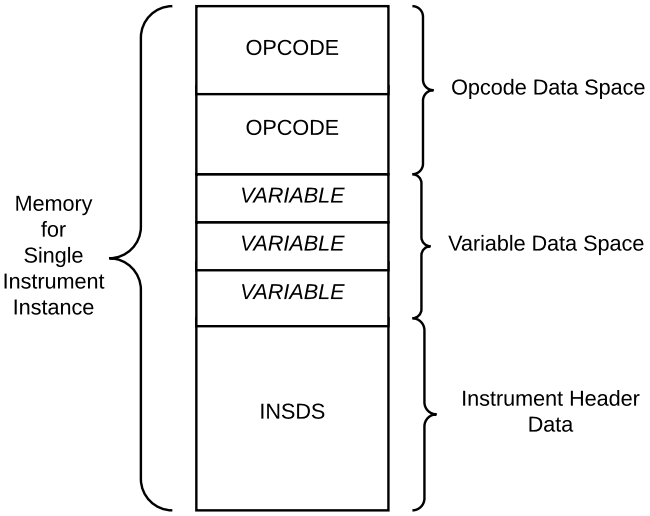

Figure 3.1: Memory layout diagram for pre-RTTI Csound instrument instance.

With RTTI, the memory allocation strategy was modified to include an additional `sizeof(CS_TYPE*)` for each variable. For the example above, the

[8]In Csound, MYFLT is a macro assigned to either `float` or `double`. This allows Csound to be compiled to use 32-bit or 64-bit numeric floating point precision for processing. The default for Csound 6 is to use `double`.

memory allocated would be `3 * (sizeof(MYFLT) + sizeof(CS_TYPE*))`. Also, the process of dividing up the total variable memory for an instrument instance consequently changed. The memory would now be interpreted as alternating pointers to `CS_TYPE` and variable memory, as shown in Figure 3.2.

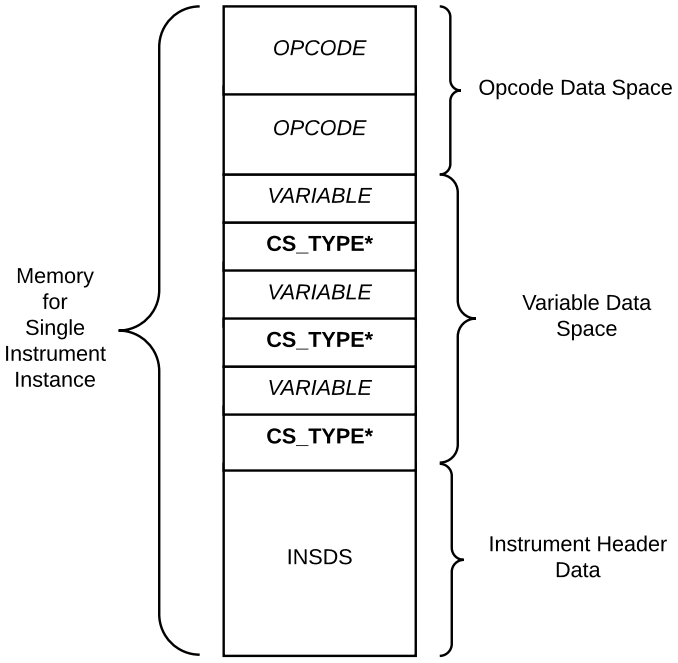

Figure 3.2: Memory layout diagram for Csound instrument instance with RTTI.

When assigning variable memory to opcodes, the system partitions the variable memory space and casts the partitions to `CS_VAR_MEM*` pointers. The members of the `CS_VAR_MEM` data structure, shown in Listing 3.27, are used within the system to clarify the intention of the code and express how the block of memory is being used. Casting memory as a `CS_VAR_MEM` does have one drawback in that compilers may align data structure members differently and introduce padding to the member data's addresses. To deal with this situation, a `CS_VAR_TYPE_OFFSET` is calculated in a low-level way to account

for any potential alignment issues. The calculated offset can then be used to find the address of the `CS_TYPE` for any given variable's data pointer.

```
// include/csound_type_system.h:51
typedef struct csvarmem {
  CS_TYPE* varType;
  MYFLT value;
} CS_VAR_MEM;


// include/csound_type_system.h:54
#define CS_VAR_TYPE_OFFSET (sizeof(CS_VAR_MEM) -
    sizeof(MYFLT))

// Top/csound.c:3747
/** Returns the CS_TYPE for an opcode's arg pointer */
CS_TYPE* csoundGetTypeForArg(void* argPtr) {
  char* ptr = (char*)argPtr;
  CS_TYPE* varType = *(CS_TYPE**)(ptr - CS_VAR_TYPE_OFFSET);
  return varType;
}
```

Listing 3.27: RTTI-related code in Csound 6

With the new RTTI system, opcodes read and write values to and from arguments in the same exact way as before this work. For opcodes that did not use RTTI, no changes were necessary. For opcodes that do use RTTI, the information retrieved for an argument was no longer based on argument index, but directly retrieved from the the data pointer for the argument using the `csoundGetTypeForArg()` function (shown in Listing 3.27). The developer would not have to remember which argument mapped to what index as they would have with the bit-flag system; rather, he would just ask of Csound what

is the type for the argument. This, together with looking at the `CS_TYPE`, has arguably led to simpler and easier to read code.

The new RTTI system in Csound6 is a simpler, more robust, and more extensible system than previous implementations. It provides a generic solution to retrieve the type for any variable. All arguments for opcodes now have their `CS_TYPE` available at runtime. Any new types will automatically be used as part of the system, freeing developers to create opcodes that use RTTI with new types without additional tracking work. Since 6.04, the new RTTI system has been employed and the previous RTTI-related code is now removed from the system.

3.2.6 Csound 6 Summary

In this section, I discussed changes to the design and implementation of the Csound Orchestra language in Csound 6. The development of the new type system, new implementation of opcode polymorphism, and runtime type identification have helped to make the core of the language implementation simpler to maintain and easier to extend. The extensions to function-call syntax and introduction of array types have furthered the design of the language and provided users with new ways to think about and write their code. These changes not only provide new features: they also maintain backwards compatibility with previous projects while also laying the foundation for future language developments.

3.3 Csound 7: New Parser, New Possibilities

In this section, I will discuss original work on the Csound Orchestra Language design and implementation I have done as part of Csound 7 and this thesis. This includes a new parser design, called *Parser3*, explicit types, User-Defined Types (UDTs, or *structs*), and a new User-Defined Opcode syntax. For each feature, I will discuss motivations, design, and implementation details. I will discuss these features in terms of their design for extensibility and potential impact.

While the features here are already designed and implemented, the final forms of these features may change as Csound 7 is not yet released.

3.3.1 Parser3

Parser3 is a new parser design and implementation for Csound 7. It is based on the NewParser that was introduced in Csound 5, using the same Flex and Bison tools for lexer and parser generation. However, it takes a very different strategy to parsing than the NewParser. This new strategy was designed to address aspects of semantic analysis in the NewParser design that limited the extensibility of the Csound Orchestra language. The following will discuss motivations for pursuing a new parser design, followed by the design and implementation of Parser3.

Motivations

The primary problem of the NewParser is that its design introduced some aspects of semantic analysis early in the compiling process, specifically in the tokenizing (i.e., lexing) step. These tokens then were used throughout the Bison grammar for the parser with the result that, arguably, the grammar

was overly complicated. As a result, implementing new language features for Csound 7 was becoming difficult using the existing grammar.

The design of the NewParser originated in Csound5, where the initial implementation was closely based upon the design and implementation of the OldParser. With the Csound5 NewParser, semantic analysis and verification were being done in various areas of the lexer, parser, and compiler. Following the design of the OldParser allowed for easier verification that the NewParser was generating equivalent results, as well as following the same rules, as the OldParser.

In Csound6, the semantic analysis code that was found in the compiler was separated out into its own phase, run after parsing but before compilation. This was a marked improvement in the clarity of the compiler code and simplified modification for both the analysis and compilation phases. However, the semantic analysis of tokens to determine if they were things like opcode names or reserved identifiers was still being done within the lexer. As a result, the grammar of the parser was still defined in terms of the numerous tokens generated by the lexer, which led to the writing of some complex rules.

The goal for Parser3 then was to continue the work started in Csound6 and to move all semantic analysis into the specific phase run after parsing. Doing so would simplify the specifications for both the lexer and parser. That in turn would make the parser easier to maintain as well as extend.

Implementation

The implementation of Parser3 moved all semantic analysis from the lexer and parser into the semantic analysis phase. Changes were required in each of those parts. They will be discussed individually below.

Lexer Firstly, all lookup-related code was removed from the lexer. In the NewParser, before any compiling was done, a special table was loaded that contained a copy of all opcode names and whether they were `T_OPCODE` or `T_OPCODE0` token types. At parse time, any time an identifier was found (identifiers are words made up of an initial letter, followed by zero or more letters, numbers, or underscores), the lexer would first do a lookup in the special table to see if it was an opcode. If so, the lexer would emit one of the two token types found in the special table. If not, the lexer would emit the token as just an identifier using the `T_IDENT` type.

For Parser3, the special table, the table initialisation code, and the opcode lookup were all removed. Instead, when an identifier was found, it would always emit a token with `T_IDENT` type. The rules in the lexer to identify reserved identifiers (`sr`, `kr`, `ksmps`, `nchnls`, and `nchnls_i`) were also removed. This removed all semantic knowledge about what an identifier meant from the lexer.

Parser Next, the grammar was rewritten to use only identifiers. In the NewParser, rules were written using the semantically aware tokens. It was here that language ambiguities were also handled, which required knowledge about the types of tokens. This wove together both recognition of the structure of the language as well as the meaning of the language.

With the Csound Orchestra language, the language had known ambiguities regarding opcode-call syntax. For example, if a line of text was found with two words, such as "word word2", it would be ambiguous whether the statement was a `word` opcode with a single input argument `word2`, or if it was a `word2` opcode with a single `word` output argument.

With the knowledge of whether one of these words was an opcode name, the ambiguity could be resolved. The NewParser then was able to generate a single tree format for all opcode statements. Consequently, the NewParser's semantic analyser and compiler could treat all opcode statements in the same way. While this worked to handle the ambiguities and simplify the compiler, it also complicated the grammar.

For Parser3, the grammar was updated to reflect the changes from the lexer. All rules were rewritten using only identifiers, which saw a number of rules removed. However, parsing opcode-statements now required a more complex set of rules (shown in Listing 3.28).

```
opcall  : identifier NEWLINE
        | out_arg_list expr_list NEWLINE
        | out_arg_list '(' ')' NEWLINE
        | out_arg_list identifier expr_list NEWLINE
        ;
```

Listing 3.28: opcall rule in Parser3

With the `opcall` rule, four different tree formats could be generated for opcode calls, depending on the structure of the opcode call statement. `opcall` became a sort of catch-all rule. It would still only match opcode-statements that would be valid in the NewParser, so that aspect was not lost. However, the generated `TREEs` for opcode-statements could not longer be used as-is by the analyser or compiler.

Semantic Analyser In the NewParser, while opcode names were recognised in the lexer, the actual lookup of the `OENTRY` for an opcode name was not done until the semantic analysis phase. The `OENTRY` defines the

opcode, including its input and output argument types. This information was necessary only when verifying that opcode use was semantically correct.

In Parser3, the semantic analyser largely stayed the same with the exception of one additional step. Previously, when the analyser encounted an opcode-statement, the `TREE` structures were all formed in the same way. Now, when the analyser first encounters an opcode-statement, it will run a `TREE` rewriting step to re-form trees into the same structure as was previously used in the NewParser. With the addition of this disambiguation step, the rest of the analyser could continue to function as-is, as could the compiler.

Note, the general algorithm applied in the `TREE` rewriting was designed to follow the same exact process found in the NewParser. This reads through the words found in the `TREE`, checks to see if they are opcode names, then checks against the variable pools, and so on. By applying the same algorithm here, the same process of disambiguation was successfully moved from the lexer and parser to the analyser.

Summary

Parser3 provides a new approach compared to the NewParser. All semantic analysis has now been removed from the lexer and parser and moved to the semantic analysis phase. Resolution of language ambiguities present in the Csound Orchestra language were consequently moved to a single location in the analyser. Parser3 also remains backwards compatible with the NewParser, meaning all previous code that could be processed with the NewParser is also valid with Parser3.

The result is that the lexer and grammar specifications have been simplified, making them easier to maintain and extend by core developers. This work would provide a foundation for other language developments in Csound7.

3.3.2 Explicit Types

Explicit types is a feature where one can declare the the type of a variable separately from the variable's name. The goal of this is to allow more flexible naming of variables and to provide a mechanism for declaring multi-character type names. The following will discuss motivations, design, and implementation of explicit types.

Motivation

Prior to Csound 7, variable types could only be determined by the initial character of the variable name, or the second letter if the first letter was a `g`, denoting a global variable. For example, a variable with the name `ivar` would be of `i`-type, and a variable with the name `gkvar` would be a variable of `k`-type. This system of naming variables has similarities to the Hungarian Notation system [162], but with an important difference: in Hungarian this system is a convention, whereas in prior versions of Csound it was a requirement.

This system for naming variables has limitations. Firstly, a user could not name a variable according to other conventions or tastes. Secondly, the number of single letters that could be used as types had a fixed limit, as the assumption was that only ASCII characters (a-z and A-Z) would be used. Third, type names were not very descriptive. Using `a` for an audio signal may be easy to remember, but using `f` for phase vocoder signals and `w` for spectral signals may be less clear.

Of these three drawbacks, the limited number of single-letters that were available to name types posed the biggest challenge to the system's extensibility. Both users and developers should be free to create new data types for Csound without having to worry about using up all possible type letters for the future. The freedom to create new types in an expressive way was a major concern when implementing user-defined types (discussed in Section 3.3.3).

Design

Explicit types separate the name and type of a variable into two distinct parts when declaring a variable. Listing 3.29 shows an example use of explicitly typed variables. The syntax defines an explicitly typed variable as one that uses an identifier, followed by a colon, followed by another identifier. The first identifier defines the name of the variable, while the second identifier names the type of the variable.

```
;; Implicitly typed ival variable
ival = 2.0
aout = doSomething(ival)

;; Explicitly typed value variable
value:i = 2.0
aout = doSomething(value)
```

Listing 3.29: Explicitly typed variables in Csound 7

The use of explicit types is only necessary the first time the variable is assigned a value. This kind of practice is also found in numerous typed languages, such as C, C++, and Java (example shown in Listing 3.30). After a variable's type is determined the first time, subsequent lookups of a variable's type are done using the entry for the variable in the currently loaded type

table while parsing. In Listing 3.29, the `value` variable is first explicitly typed as an `i`-type variable on its first assignment. In the following line, the value variable is found in the type table, and an i-type is used when resolving the arguments to the imaginary `doSomething` opcode.

```
double ival = 2.0;
double[] aout = doSomething(ival);
```

Listing 3.30: Variable declaration and use in C/C++/Java

Implementation

Implementing explicit types required changes to the lexer, parser, and semantic analyser. Firstly, a new rule called `TYPED_IDENTIFIER` was introduced. This rule matches text of the format "identifier:identifier" and is shown in Listing 3.31. Next, the parser was updated to handle the new `TYPED_IDENTIFIER` token. This was done in the `out_arg` rule, which means that typed identifiers could now be used as output arguments (also shown in Listing 3.31).

```
// Lexer rule for typed identifiers
TYPED_IDENTIFIER
    [a-zA-Z_][a-zA-Z0-9_]*:[a-zA-Z_][a-zA-Z0-9_]*

// Parser3 rule for out_arg
out_arg : identifier
        | typed_identifier
        | array_identifier
        | array_expr
        | struct_expr
        ;
```

Listing 3.31: Lexer and parser changes for typed identifiers

Finally, the semantic analyser was modified to check for typed identifier output arguments. For output arguments, the analyser would already check if a variable was registered in the type table. If not, it would previously determine the argument's type using the first or second letter as well as check if it was an array or not. That code was then modified to first check if the argument was a typed identifier. If a typed identifer was found, the left-side of the colon would be used as the variable's name and the right-side would be used as the variable's type. From here, processing continued as it had before and identifier lookups in the type table would now find the name used with the typed identifier.

Summary

Explicit types were implemented for Csound7. They provide greater flexibility for users to name variables as they wish, without requiring that the variable's type be part of the name. Additionally, as the variable's type was separated out from the name, the type name could now be of any length. This would be important for implementing and using User-Defined Types (Section 3.3.3) and the new UDO syntax (Section 3.3.4). Finally, the change was implemented as a new option for declaring types, and the previous method for determining types from variable names was retained, thus providing complete backwards compatibility.

3.3.3 User-Defined Types: Structs

User-Defined Types (also called *structs* in Csound) are a feature that allows users to define their own data types using Csound Orchestra language code.

This provides extensibility within the language for the user to create new kinds of data and signals for processing.

Motivation

In Csound6, the new type system (described in Section 3.2.1) provided concrete type definitions. For core developers, it clarified both definition and internal use of type-related code. For third party developers, it made data types extensible, providing a systematic way to define and introduce new types.

For Csound7, the ability to define types is extended out to users and implemented as *structs*. The goal of this is provide users with the same ability to create their own signal representations that developers received in Csound6. This permits new kinds of research to be done by users.

Design

In Csound 7, user-defined types are called *structs*. They are based on C's concept and implementation of structures, which are defined using the `struct` keyword. Kernighan and Ritchie define structures in C as:

> a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, notably Pascal.) Structures help to organise complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.[9]

[9] Chapter 6 - Structures, page 103. [99]

Like C, Csound structs are defined in terms of other existing types. They may use natively-defined types (such as `a`-, `k`-, and `i`-types), arrays, as well as other defined struct types. Structs in Csound can also be used with arrays, such that one can define an array of structs.

```
struct TypeName varName1, varName2 [, varName3...]
```

Listing 3.32: Csound struct syntax

Listing 3.32 shows the syntax for defining structs in Csound Orchestra code. The user defines a struct type by using the `struct` keyword, followed by the name of the type, then a comma-separated list of members of the structure. As in other areas of Csound, struct member variables are typed using the same implicit or explicit type rules as other variables. Listing 3.33 presents a sample definition of a complex number type using explicitly-typed variables.

```
struct ComplexNumber real:k, imaginary:k
```

Listing 3.33: Csound struct example: ComplexNumber

Using Structs Defining a struct registers the data type with Csound's type definition table. Once registered, users can write code that creates instances of the struct, read from and write values to the struct, as well as use the data type as arguments in UDOs. Listing 3.34 shows these various facets in use. (Note that the opcode definition here uses the new-style UDO syntax, discussed in Section 3.3.4.)

```
struct Rectangular x:i, y:i
struct Polar R:i, t:i
```

```
opcode to_polar(num:Rectangular):(Polar)
  ipolarR = sqrt(num.x ^ 2 + num.y ^ 2)
  ipolart = taninv2(num.y, num.x) * (360 / (2 * $M_PI))
  retVal:Polar init ipolarR, ipolart
  xout retVal
endop

instr 1
  r:Rectangular init 1.0, 0.5
  polar:Polar = to_polar(r)

  print polar.R ; 1.118
  print polar.t ; 26.565
endin
```

Listing 3.34: Csound struct usage example

In the example, the `to_polar` UDO takes in a single argument called `num` of type `Rectangular`. A variable called `retVal` of type `Polar` is initialised using values that are calculated using the `x` and `y` members of the passed in `num`. The syntax to access member values of a struct variable follows the same syntax as C, using the variable's name, followed by a period, followed by a member name. In the example, `num.x` and `num.y` are used to read the `x` and `y` member values from the num variable.

Implementation

Implementing structs in Csound required modifications to the parser, semantic analyser, compiler, and engine runtime. Also, new type system related code was required, which will be described below. For the parser, new rules

(shown in Listing 3.35) were added to Csound's grammar for processing struct definitions and member access. The `struct_expr` rule was also added to rules for input and output arguments (not shown).

```
struct_definition : STRUCT_TOKEN identifier struct_arg_list
                  ;

struct_arg_list : struct_arg_list ',' struct_arg
                | struct_arg
                ;

struct_arg : identifier
           | typed_identifier
           | array_identifier;


struct_expr : struct_expr '.' identifier
            | identifier '.' identifier
            ;
```

Listing 3.35: Struct-related grammar rules

Next, the analyser was modified for processing struct definitions. When a definition is found, a new `CS_TYPE` is generated and registered with the type system, and a new `init` opcode `OENTRY` is synthesised and registered. Once the type is registered, it can be understood by the rest of the system and used for variable declarations, UDOs, etc. The `init` opcode is what allows users to create new instances of the struct.

For the generated `CS_TYPE`, the members in the struct definition are parsed and added to the type definition's `members` field as a list of `CS_VARIABLEs`. The variables define both the name and the type of the member.

Other type-related functions for `CS_TYPE`, `copyValue` and `createVariable`, are written in a generic way. They receive both the `CS_TYPE` and the memory allocated for the variable. When structs are copied or newly created, these functions reference the member variables from the `CS_TYPE` to perform their processing with the variable memory.

```
typedef struct csstructvar {
  CS_VAR_MEM** members;
} CS_STRUCT_VAR;
```

Listing 3.36: C data structure for Csound struct variables

Listing 3.36 shows the C data structure used for Csound struct variables. When a new Csound struct variable is created, the size to allocate for the variable will be calculated from the sizes of the members defined in the `CS_TYPE`. This uses the `CS_VAR_MEM` data structure so that the `CS_TYPE` for each of the members precedes its variable data, which allows RTTI to function with struct member data.

For struct member access, they are treated specially. For the analyser, the information from the struct member's type is used for semantic verification. In the compiler, the member access is converted into a special address notation. At runtime, when an instrument instance is created, the notation is used to find the location in memory for the specified member. The process starts at base struct's address, then navigates using the notation to find the specific member's address using the `members` field from `CS_STRUCT_VAR`. The result is that when struct members are used with opcodes, the data pointer set for the opcode is the direct location for the member. This implementation adds a small cost at initialisation time to find the correct data address; however, this

adds no additional opcode calls at runtime to retrieve or set values within the struct.

Summary

Structs provide a user-extensible way to introduce new data types into Csound. The syntax for definition and usage were modeled upon C's syntax and semantics for structures. The implementation provides efficient runtime characteristics as no additional opcode calls were necessary. The result is that users can now do new kinds of research and musical work with Csound that requires new data types.

3.3.4 New User-Defined Opcode Syntax

The new-style UDO syntax provides an alternate way to define UDOs from the previous system (here called "old-style UDO"). This was done to work with explicit types and structs, which the older-style UDO syntax could not accommodate.

Motivations

With the introduction of the new type system in Csound6 and structs in Csound7, the old-style UDO syntax presented problems for defining arguments with multi-character type names. Listing 3.37 shows an example of old-style UDO syntax. Input and argument types are given as a series of single-letter types, using one letter per argument.

```
opcode myAdd, k, kk
  k1, k2 xin
  kval = k1 + k2
```

```
  xout kval
endop
```

Listing 3.37: Pre-Csound 7 UDO definition

Attempting to extend the old-style specification would have been awkward at best. Listing 3.38 shows some possible approaches to extend the old-style syntax. Note that each form requires a special start character to determine when a possible multi-character type started, and a special end character to determine when the type name ended. Arguably, these alternatives are not very easy to read or understand.

```
opcode my_opcode, a'MyType;kk, a
  ...
endop

opcode my_opcode, a"MyType"kk, a
  ...
endop

opcode my_opcode, a:MyType;kk, a
  ...
endop
```

Listing 3.38: Possible alternate syntaxes for old-style UDOs

Also, another problem with old-style UDOs is that the argument specification is done using text instead of a quoted string. Because of this, special lexer and parser rules were required to handle the case of UDO argument specifications. This made the lexer and parser rules for UDOs awkward to extend.

Design

The design of new-style UDOs took into account both the requirements for multi-character type names as well as conventions found in other programming languages. Listing 3.39 shows examples of a `myAdd` function in C/C++/Java and Pascal. Like Csound, these are statically typed languages; they all share a common approach to function definition. Input arguments for a function are listed with both the name of the input argument as well as its type. For output arguments, these languages specify only the types of the output arguments, but not names.

```
// C, C++, Java
float myAdd(float k1, float k2) {
  return k1 + k1;
}

{ Pascal }
function myAdd(k1, k2: real): real;
begin
  myAdd := k1 + k2;
end;
```

Listing 3.39: Function definitions in various programming languages

In the C/C++/Java example, `float` is specified as the output argument type, then the `myAdd` function name is given, and `float k1` and `float k2` are specified in a comma-separated list within parentheses. In the Pascal example, `k1` and `k2` are declared as input arguments with the `real` type, and the output argument type is also specified as a `real`. Note that the type of the value returned from the function, as specified in the `return` line in

the C example and with the `myAdd :=` statement in the Pascal example, is type-checked to agree with the specified output type.

A new syntax was developed for Csound 7 to accommodate types that are longer than one character in length. An example of this syntax is presented in Listing 3.40. This syntax shares common properties found in the examples in Listing 3.39: input arguments are given as a list of names together with their types, and the output types are specified without names in a second list. Because Csound allows for returning multiple values from an opcode, a single output type-specifier was insufficient to accommodate Csound opcode definitions. Consequently, a list within parentheses was used.

```
; void return
opcode no_return(k1, k2):void
endop

;; Single k-type return
opcode myAdd(k1, k2):k
endop

;; Empty return, equivalent to void
opcode myAdd(k1, k2):()
endop

;; Single k-type return
opcode myAdd(k1, k2):(k)
endop

;; Multiple type return
opcode myAdd(k1, k2):(k, a, Rectangular, Polar)
```

```
endop
```

Listing 3.40: Csound 7 new-style UDO definitions

The new-style UDO syntax was chosen to look similar to function definition forms found in other languages. It also adds the ability to specify both argument names and types. In Listing 3.40, the input arguments are specified as simply `k1` and `k2`, but in Listing 3.34, the `to_polar` opcode takes in an argument of `num:Rectangular`, using an explicitly-typed variable name.

The practice of defining variable names for input arguments follows the same conventions for variable names found elsewhere in the Csound Orchestra language. Users can choose to use non-qualified variable names, falling back on the first-letter rule to determine the type of the variable, or use the `name:type` convention to explicitly give the type for the argument. It is the latter convention that allows for variables to use multi-character type names.

The types listed for the output argument of new-style UDOs can be of three forms: a single output type, a comma-separated list of output types within parentheses, or the word `void`, which denotes no return arguments. Of the three, the parentheses form is the standard practice, as it is generic and works with zero, one, or multiple return types. The void and single-arg forms are therefore syntactic sugar for quicker and easier reading and writing.

Implementation

Implementing new-style UDOs required changes in the lexer, parser, and semantic analyser. For the lexer, the only new requirement was for a new `VOID_TOKEN` type that would match the word `void`. For the parser, the existing `udo_definition` was augmented to include the new-style UDO syntax, and additional rules were added for the input and output argument

lists for new-style UDOs. The code for the grammar changes are shown in Listing 3.41.

```
udo_definition : UDOSTART_DEFINITION identifier ','
   UDO_IDENT ',' UDO_IDENT NEWLINE statement_list
   UDOEND_TOKEN NEWLINE
               | UDOSTART_DEFINITION identifier udo_arg_list
   ':' udo_out_arg_list NEWLINE statement_list UDOEND_TOKEN
   NEWLINE
               ;

udo_arg_list : '(' out_arg_list ')'
             ;

udo_out_arg_list : '(' out_type_list ')'
                 | VOID_TOKEN
                 ;

out_type_list : out_type_list ',' out_type
              ;

out_type : identifier
         | array_identifier
         ;
```

Listing 3.41: New-style UDO grammar rules

Finally, two changes were made to the semantic analyser. Firstly, internal representations of opcode argument specifications were modified to allow multi-character types by using delimiters. The form used was the one shown in the last example in Listing 3.38, using a colon and semi-colon as start and end delimiters. While this would have been awkward for users to use,

using it internally would have no impact on the user. This approach was taken as all opcodes, including native ones, ultimately hold their input and output arguments in strings. Updating the code that parses argument strings and adding delimited multi-character types was vastly simpler than trying to change argument specifications for all opcodes.

Secondly, when the analyser encountered a new-style UDO, it would first rewrite the `TREE`. The analyser would look at all input arguments and their types, generate an input argument string, and generate an `xin` opcode call. It would then look at the specified output argument types, generate an output argument string, and verify that a type-appropriate `xout` opcode call was found. Finally, the input and argument type strings were appended to the UDO `TREE`.

By the end of the `TREE` rewrite process, the UDO `TREE` would be in a format compatible with the old-style UDO format. From here, semantic analysis and compilation for UDOs proceeded as before.

Summary

Csound7 provides a new-style UDO syntax that is designed to work with multi-character type names. This allows UDOs to work with structs and is enabled by the use of explicitly typed variable names. The new syntax lets users specify input arguments and types together, rather than specifying types in one place and arguments by name in another, as was done with old-style UDOs. Finally, the new-style UDO syntax is arguably simpler to read and write, and uses a familiar design found in other programming languages.

3.3.5 Csound 7 Summary

In this section, I discussed changes to the design and implementation of the Csound Orchestra language in the upcoming Csound 7. The new Parser3 design provides a revised approach to semantic analysis in the compiler that simplifies the lexer and parser, a change which facilitates new language design work. This work in turn was the foundation for the implementation of explicit types, User-Defined Types, and the new User-Defined Opcode syntax. With explicit types and new UDO syntax, the user is provided with more expressive ways to define their code. With UDTs, users are given a completely new way to extend their work by defining new data types in Orchestra code. These features build upon the work of Csound 6 and continue the evolution of the Csound Orchestra language.

3.4 Conclusions

In this chapter, I have discussed new work for this thesis that further developed the extensibility of Csound's language for both users and developers. In Csound 6, I introduced a new type system that made defining Csound data types an explicit operation, organised code related to types, and clarified the use of types within the Csound system. I extended the function-call syntax and implementation of opcode polymorphism, which led to a new way for users to express their code using a more functional programming, expression-based style. Finally, I implemented generic arrays, offering a new data type for opcode writers and users to use. This provided new ways to write things such as multi-channel audio code.

In Csound 7, I introduced a new parser design called Parser3. This moved all semantic analysis into its own phase and simplified the lexer and parser specifications. This work would simplify three new language developments. Firstly, *explicit types* provided a new syntax that freed the user to define variable names without restriction; it also enabled definition of variables using multi-character type names. Secondly, User-Defined Types allowed users to define their own data types using the Csound Orchestra language. Finally, the new-style UDO syntax provided a clearer way to define UDOs that would work together with UDTs and explicit types.

The work presented has contributed to the evolution of the Csound Orchestra language in Csound 6 and in Csound 7. It has extended both the design and implementation of the language as well as provided developers with new ways to do the same. It has given users new ways to write their code as well as new ways to extend the system themselves. These changes have all been additive and preserve backwards compatibility. Thus, the history of Csound-based work has been preserved in the context of an active and living system.

Chapter 4

Extending the Reach of Csound

This chapter will discuss *platform extensibility*: extending the use of music software by porting it to other platforms. It will begin by looking at properties of cross-platform software and employing dependency analysis to understand the challenges of platform extensibility. Next, it will analyse portability in existing computer music systems. Afterwards, the CsoundObj Application Programming Interface (API) will be presented, designed to address the conflicting goals of project portability, which requires feature parity on each platform, and utilisation of novel features, which are by definition unique to each platform. The chapter will then look at work in extending Csound [35] to three new platforms: iOS, Android, and the Web. Finally, various case studies of software that uses Csound on these new platforms will be presented. The chapter conclusion summarises the value of platform extensibility.

Much of this work has been presented by the author in [203], [111], [109], [107], [108], and [98]. The work was done in collaboration with Victor Lazzarini, John ffitch, and Edward Costello. My own original work for this thesis includes designing the CsoundObj API, creating the Android and

iOS Examples projects, and researching and setting up toolchains and build scripts for iOS, Android, and Emscripten versions of Csound. The following will cover lower-level design decisions and look at the work in the context of extensible systems.

4.1 Overview

Csound has long been a cross-platform program, running on desktop, laptop, server, and embedded devices. Using standard cross-platform build techniques and coding practices, the Csound program has grown over time to adapt to work on many systems. With each new platform comes new challenges to assumptions held within the code and build system, and changes to the system in response to those new challenges.

Developing programs to support multiple platforms brings numerous benefits. It keeps the programs alive when platforms become obsolete. It allows users to take advantage of new features on new platforms, while leveraging their pre-existing knowledge and experience. It lets the works of the past function in the musical world of today and provides a means to directly explore the history of practice for a music program.

4.2 Platform Extensibility and Cross-platform Development

One of the primary facets of evaluating music systems today involves examining the platforms on which these systems run. Understanding the platform-specific features that programs require gives a picture of how dif-

ficult it may be to move that software to new platforms. This affects not only the portability of the program itself but also the projects that users can create using that software. Additionally, dependencies on specific features may also impact the program over time on the same platform as the platform itself evolves.

In the early days of computer music, the Music-N family of systems was often defined by the very platform that these systems ran on (e.g., MUSIC 360 and the IBM 360, Music 11 and the PDP-11). These systems shared a common history and similarities in design and features. However, the small differences between systems – such as what unit generators were available and differences in language syntax – meant there were incompatibilities between each program. As the software was intimately tied to its platforms – such as Music 11 using assembly language that only worked with PDP-11 machines – the software was not portable, nor were the projects created with it. Users would have to port their own projects when moving from one system to the next.

Later, as the world of computing evolved and higher-level languages and their compilers developed, it was possible to create cross-platform programs that could be compiled and run upon different platforms.[1] Using a single codebase, together with build-time and compile-time configuration, a cross-platform application or programming library could be made.

The benefits of cross-platform programs can be understood in a few ways. Firstly, extending a program to a new platform may be an act of preservation, both of works and of workflows. By extending a program to new platforms, existing works and knowledge can continue to function in a living context,

[1]By *higher-level*, I am referring to languages that do not map directly to machine code, but instead offer more structured programming constructs. For more information, see [134].

even if a prior supported platform becomes obsolete. Secondly, supporting new platforms extends the usability of the program. A user can learn the program on one system, but employ it on multiple platforms. Users may then take advantage of unique features specific to each platform, but still reuse existing knowledge and experience using a portable program. Finally, supporting new platforms can be viewed as a means to take advantage of new developments in hardware. In this regard, users may not gain new features, but do gain in potential performance improvements. The following will explore what is necessary to develop cross-platform music software. The work will begin by discussing the general process for developing software for a single platform. Next, it will look at cross-platform software and develop a framework for evaluating program requirements as a graph of dependencies that must be satisfied for the software to operate. The practice of dependency analysis will be further extended to users' projects written with that software. Finally, the concepts discussed will be applied to analyse Csound and related projects.

4.2.1 Single-Platform Software Development

Developers create software that users use on a target platform. A platform is a general term used to describe the environment where programs are executed. The term may be refer to technologies such as operating systems (e.g., Windows, Linux, OSX), interpreters (e.g., Java, Ruby, Python), and hardware (e.g., Intel or ARM CPUs). From the perspective of the software developer, a platform defines both what software (i.e., libraries and and executables) and resources will be present for their own software to use, as well as what format their software must be in to operate on that platform. For example, on the Windows platform, a native application can utilize Windows-

provided libraries to access hardware features, and the application must be compiled into the Windows binary executable format for the platform to run the application.

Developers create and use source files to define their programs. Source files can be classified as code (i.e., text files written in a programming language) or resources. Code defines a program's operations and resources are used at runtime by the program as a source of data. The software may be completely defined by its source files, or it may require features provided by libraries or executables — whether they are provided by the target platform or by a third-party. The libraries and executables that a program requires are called the program's *dependencies*.

The format for source files may or may not be the in same format required for execution. The source and target formats will be the same when the target platform can interpret the source code at execution time. For example, when developing software for Web Browser platforms (e.g., Chrome, Firefox, Safari), developers might write their applications using HTML, Javascript, and Cascading Stylesheet (CSS) code files. Users operating their browsers will execute the developer's software by loading in the source code.

When a platform requires a format that is different from the source code, developers will employ various tools (i.e., executable applications) to translate (i.e., compile or build) the source into the target format. The build process may be as simple as a single application of a tool (e.g., using a C compiler with C source files to generate an executable application) or may be an application of multiple tools organised into a processing network. In a more complex build, one tool may process one-to-many inputs and generate one-to-many outputs, and the outputs may be further processed by other tools.

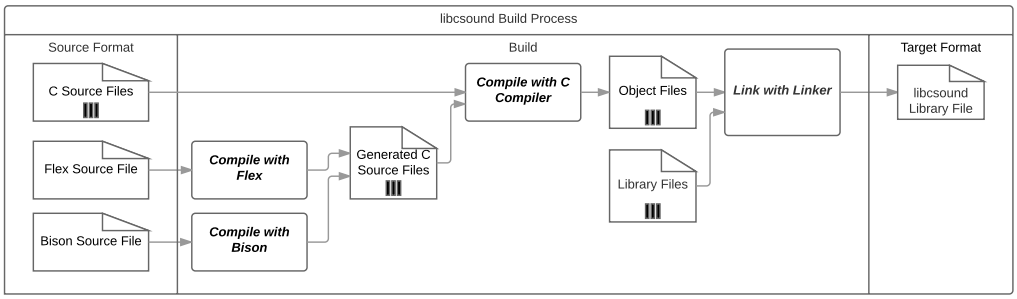

Figure 4.1: libcsound build process

Figure 4.1 shows an example of the build process for Csound's main library, libcsound. Data icons represent concrete files, rounded rectangles describe processing by tools, and arrows denote the flow of processing, starting from the source files on the left and ending with the libcsound library on the right. The first step involves the compilation of Flex and Bison grammar source files to generate C source files. The generated files are next compiled together with other developer-written C source files to produce objects files that contain native machine code. The final step uses a linker tool to join together the various objects files and either statically link in machine code from static libraries, or create dynamic (i.e., run-time) links to dynamically-loaded libraries, depending upon the link configuration set for the build. The end result of the process is a library file in the format of the target platform.

After building a program, developers may further *package* and *deploy* a program to release it to users. Figure 4.2 illustrates a typical release process for desktop and mobile music applications. Packaging tools are used to create packages (i.e., archive files in a specific format) or installers (i.e., executable applications) that include programs and resources (e.g., documentation, icons). The packaging artifact is then deployed (i.e., pushed) to a public server to make the software available for users. The deployment process could be a

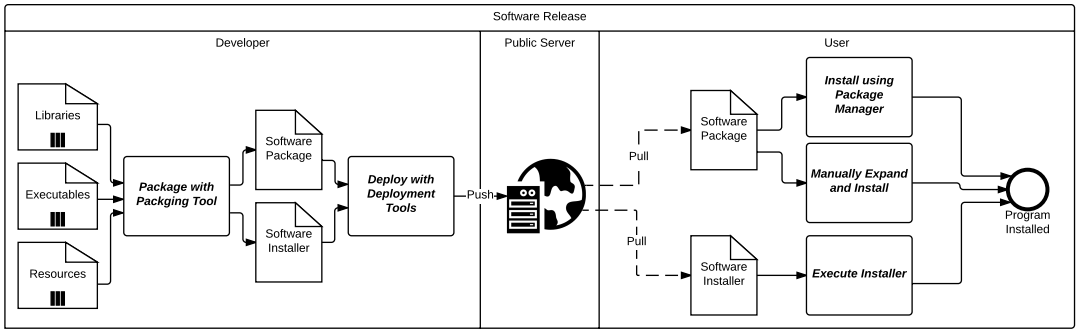

Figure 4.2: Software release process

manual one where a developer uploads a file to a publically accessible web server, but it may also be an automated one involving its own set of tools. Users then download (i.e., pull) a released program and install the package, making the software available for use on their system.

The complete set of tools used for building, packaging, and deploying a program is called the *toolchain* for the program. Developers may operate the individual tools manually to transform code, but more often they will automate various processes using a *build tool*. Build tools use some form of *build file* (i.e., project or script file) that describes what are the source files for the program, what tools are used for transformations, and what is the order of operations required to build the software. Build tools may be stand-alone programs (e.g., Make) or be an included feature as part of an Integrated Development Environment (IDE) (e.g., Xcode, Visual Studio). Build files will typically group sets of operations into *targets* that perform build, package, deployment, and other tasks. Targets may have dependencies upon other targets. For example, if a user executes the deployment target, the build tool may first package the application, which itself may require building all build targets.

The general process of developing software for a single platform involves the application of a toolchain to transform software from its source files to a target format. Build tools organise and operate the toolchain to create the end program. The build and release processes described above are only examples of what may be done. Other software may require more complex processes employing many more tools, libraries, and steps to build and deploy the program. Other platforms, such as web browsers, may have different execution models that would make installation to a server a part of the developer's deployment process and delivery to the client as part of the execution process. The software development process is customised for each program and each target platform. The process for building software for multiple platforms builds upon the practices of single-platform development by specifying toolchains and processes for each platform, further described below.

4.2.2 Cross-platform Software and Projects

Cross-platform software is one that uses the same source files to produce programs for multiple target platforms. Each platform has its own set of available libraries, tools, and resources that may intersect, or be completely disjoint, with those found on another platform. As a result, the toolchains and software development processes (i.e., build, packaging, deployment) for each platform may share much in common with, or be completely unique from, each other.

Developers must make a choice as to how to adjust to each platform difference, identifying those features within their program that are *required* and those that are *optional*. Depending upon what a platform supports,

programs may not be able to run at all on a given platform if all of their required features are not supported, even if all optional features are available.

The degree to which a program is cross-platform has a direct impact on the portability of projects used with the program. If all of a program's features are available on all supported platforms then the projects for that program will be completely cross-platform. If the software allows for partial cross-platform compliance then the user must decide whether to take advantage of platform-specific features; the risk is that their work may then operate only on a subset of available platforms.

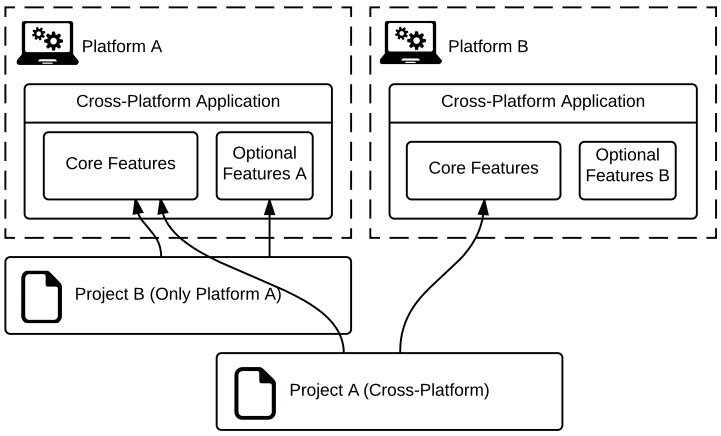

Figure 4.3: Project and Program Dependencies

Figure 4.3 illustrates the dependencies of two different projects upon a cross-platform program with optionally supported features. For Project A, which only has a dependency on the Core Features, it is able to run on both Platform A and Platform B, where the Core Features are equally available. However, for Project B, which depends on both the Core Features and Optional Features A, it is only possible to use this project on Platform A. While Platform B offers the Core Features, it does not offer Optional Feature A, and thus Project B cannot run as its dependencies have not been satisfied.

Ultimately, the process of making software cross-platform is an exercise in understanding completely the program's dependencies. Similarly, a user who wants to work across platforms must perform the same task of identifying the project's or work's dependencies. The following will discuss dependency analysis. Afterwards, techniques will be described for dealing with those dependencies when moving across platforms.

4.2.3 Analysing Dependencies

Programs have dependencies that must be satisfied for them to build and run. For programs that require building, all of their required *build-time dependencies* must be satisfied to compile the program. Examples of build-time dependencies may be the presence of particular programming libraries, a certain operating system, or specific build tools.

For a program to run, all of its *run-time dependencies* must be satisfied. Examples of run-time dependencies include the presence of specific dynamically-linked libraries,[2] as well as the presence of certain CPU features. For example, if a program is compiled to use AVX instructions, it will run only on hardware platforms where the CPU supports those instructions.

Dependencies may be *abstract*. It is possible that more than one program, library, or tool may satisfy a program's requirements. For example, a program written in C may compile using the GCC and Clang C-language compilers but not with the Microsoft Visual C (MSVC) compiler. In this scenario, the tool dependency is satisfied with the first two options but not the latter.

[2]Dynamically-linked libraries are linked into the program at runtime, as opposed to statically-linked libraries, which are linked and permanently compiled into the program at build-time. For further information, see [113].

Another example is a program that may have been compiled using version 2.0 of library X but only version 2.1 is available on the system. However, version 2.1 of the library may be backwards compatible with 2.0, and thus may be used to satisfy the dependency. In these cases, the abstract dependency has been satisfied by a *concrete* implementation.

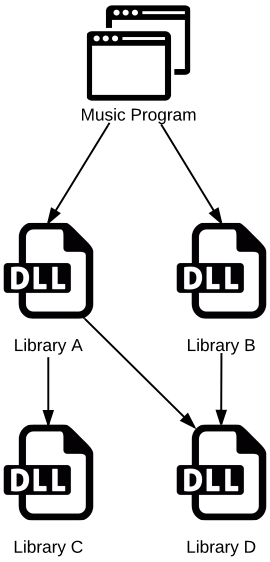

Figure 4.4: Example dependency graph

A software's network of dependencies can be organised into a directed graph. Figure 4.4 shows an example graph representation for a music program. Here, the program depends directly on Library A and B. In turn, Library B depends on C, and both Library A and B depend on D. Here, Library C and D are called *transitive* dependencies of the music program.[3]

For developers, dependency analysis aids in seeing a complete picture of what is required to build and run a program. Dependency analysis of their works can be a beneficial practice for users as well. For example, if a user creates a real-time music work, they may use Software X, Plugins Y and Z,

[3]For further discussion of transitive dependencies, see [127, 3.4 Project Dependencies].

and Hardware A and B. Like the developer, the user has to account for each one of these dependencies to reproduce the work. Users must understand that when they use multiple applications and plugins together the overall robustness of the project – its ability to load and operate as originally intended over time – is directly related to the work's dependency graph.

4.2.4 Moving Across Platforms

Moving a program to a new platform involves identifying all dependencies and satisfying them. Some dependencies are already known from previous platforms. In these situations, finding a compatible version is all that is required.

However, sometimes new dependencies are discovered. This can be an issue when an assumption about the software becomes invalid in a new context. For example, as discussed in Section 4.5.1, the ability to dynamically load plugins was an assumption that was a part of Csound's architecture. This assumption was invalid when moving Csound to iOS where no plugin loading is allowed. The ability to load plugins was then recognized as a new dependency. In this situation, the dependency was made optional, and an alternative architecture for building Csound-based applications was developed (see CsoundObj in Section 4.4).

Software must be able to adapt to differences between platforms. This is usually done through a mix of build configuration and conditional compile-time checks. For build configuration, developers may use multiple build files for each platform or employ a *configuration tool*, such as Autotools [41] and CMake [120], to interrogate a platform and generate customised build files for the platform. Configuration tools will report to developers when required

dependencies are missing and the build process is unable to continue. For optional dependencies, the tools will write a configuration file (i.e., a generated source file) or in some way communicate what was found to the building program. Also, the person using the configuration tool may have options to explicitly enable or disable features.

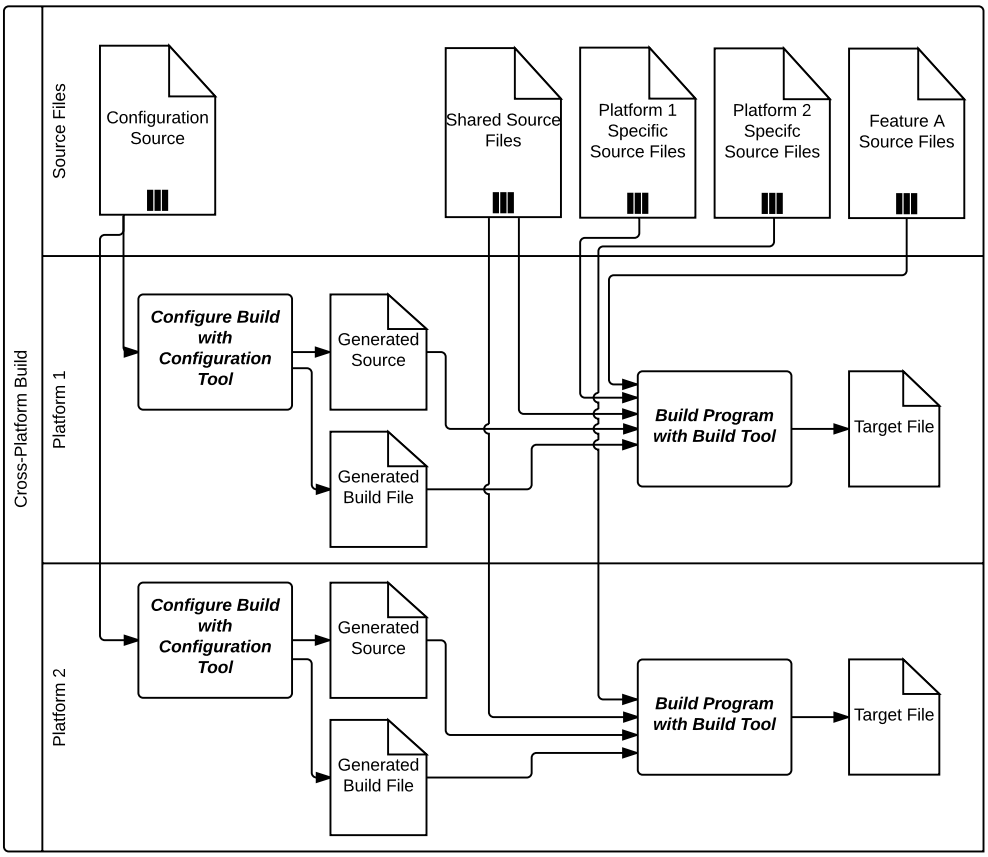

Figure 4.5: Cross-platform configuration and build

Figure 4.5 illustrates the process of using a configuration tool for cross-platform development. Source files written by developers are shown as data icons in the top swim-lane. The builds for Platform 1 and 2 both begin by using a configuration tool to process a configuration source file. The tool interrogates the platform and generates source files (e.g., a configuration

header file) as well as a build file. Next, the developer uses a build tool on each platform to build the program using shared, generated, platform-specific, and feature source files. The configuration tool determines the exact set of source files to use to build the program on each platform. In this example, the source code for Feature A is included as part of the build only for Platform 1, but not for Platform 2. This illustrates a case where an optional dependency was found on one platform but not on another, and, consequently, a feature in the program was enabled for only one platform.

Once a build starts, program code can be written to check the results of the configuration. The code can conditionally enable or disable parts of code using that information. The use of these compile-time configuration checks are an an additional cross-platform development technique to the conditional inclusion of source files described above. By using a mix of configuration and compile-time checks, programs can be made to adapt to the software and resources found on a platform.

As a general rule, the greater the number of dependencies, the more difficult it is to adapt software to new platforms. This rule also applies to users' works and the software used to create them. Minimising the number of dependencies and requirements when developing software and works is a good practice for increasing robustness and platform extensibility.

By looking at the graph of required dependencies for a software, one can see what must be satisfied when moving a software to run on another platform. By looking at the optional dependencies that are available, one can see how much of a software is cross-platform. The degree of platform-dependent features used therefore determines the portability of a user's work.

4.2.5 Summary

This section discussed aspects of cross-platform development to consider for software and works. Using dependency analysis and considering all transitive dependencies can help show what must be accounted for when moving to new platforms. Using configuration and compile-time conditional checks can make software adapt to each unique situation. These tools can aid in making more robust software and help preserve users works over time.

4.3 Related Work

Many computer music systems today are cross-platform, though some are more amenable to porting to new platforms than others. The following will provide an analysis of SuperCollider 3 [123] and Pure Data [148]. These systems are actively developed and used, open-source, and cross-platform. I will look at their dependencies as well as various aspects of their architecture and design that contribute to their platform extensibility.

4.3.1 SuperCollider 3

SuperCollider 3 (SC3) is a computer music language and audio engine server. Originally by James McCartney, it is now developed and maintained by a community of developers and users. The code for SC3 is written in C++. It is available and primarily used on Linux and OSX operating systems; Windows versions are also available.[4] A version of SC3 is available on Android [161], though it uses its own version of SC3 code that is a separate fork from the

[4]Windows versions were not always available for SC3. However, the latest stable release, 3.6.6, does provide a Windows installer.

main project. The codebase for SC3 includes a `README_IPHONE.md` file for building SC3 for iOS, but the file has not been edited in years and refers to a directory for iPhone-specific code that does not exist. A separate version of SC3 for iOS [189] is available, though it does not appear to be maintained at this time.

Required

- Tools
  - C++ Compiler (GCC 4.7+, Clang, Intel C++ Compiler, MSVC).
  - CMake.
- Libraries
  - Boost.
  - Pthreads (non-MSVC platforms).
  - yaml-cpp (falls back to version in SC3 source tree).
  - Audio API (either CoreAudio, JACK, or PortAudio, depending on platform).
  - QT 5 (for SC-IDE).

Optional

- Libraries
  - FFT Library (VDSP or FFTW3, falls back to fftlib.c by John Green, provided in SC3 source tree).
  - libsndfile.
  - libavahi (for non-OSX platforms).

Figure 4.6: SuperCollider 3 Dependencies

Figure 4.6 shows a basic breakdown of SC3's dependencies for a typical build of SC3. This includes the sclang and scsynth modules, as well as the SC-IDE GUI application. SC3 uses the CMake [120] configuration system, which in turn generates build files for various other build systems. SC3's build system is fairly flexible; some required dependencies are made such that they can use a version available on the system if found, or otherwise fall back to one provided within the SC3 source-tree.

Looking at the dependencies, the ones that are truly third-party libraries – such as Boost, QT, and libsndfile – are readily available across most platforms. Others – such as VDSP, FFTW3, and libavahi – are platform dependent, but may be optionally used. Overall, the build system and source of SC3 has developed over time to adapt well to new platforms.

The typical SC3 user writes SuperCollider code, using SC-IDE or other code editing environment (e.g., Emacs, Vim). In turn, the code is then evaluated by an interpreter, *sclang*, which parses and compiles the code into OSC messages. These messages are then sent to the SuperCollider engine, *scsynth*.

A developer can compile and use scsynth as a library. Using C++ code with the library, developers can create and embed an SC3 engine (called `SCWorld`) into their application. The public functions for `SC_WorldOptions.h` only provide few options for working with the engine, mostly starting and stopping an engine, and sending OSC messages to it. Use of the library is not typical for third-party application makers and its availability is not well publicized.

For third-party developers, a more typical design uses the standard scsynth server executable in a separate process, then communicates with the application via OSC over TCP or UDP network protocols. This allows users

to use the SuperCollider language and sclang to work with SC3, or use a completely different front-end language that is able to communicate with scserver using the same messages and protocols as sclang.[5]

Like Csound and PD, SuperCollider provides developer extensibility in the form of shared-library plugins.[6] The plugins may include their own dependencies outside of those found already in SC3. Users using features provided in plugins must account for each plugin's degree of portability when evaluating the portability of their own work.

Overall, SC3 has good platform extensibility. Its codebase is in C++ which is well-supported across many platforms. It is currently actively supported on desktop operating systems and embedded systems. While the main repository does have support for Android and iOS, these do not look to be standard build targets as part of the standard release process.

4.3.2 Pure Data

Pure Data (PD) is primarily known as graphical computer music system. A version of PD called *libpd* is available that runs PD patches but does not provide a graphical user interface. The engine code for PD is written in C, while the application code is written in TCL, using the TK user interface library [45].

In terms of platform extensibility, there are two aspects of PD: the runtime audio engine and the graphical application (pd-gui). PD's engine code is very portable as it is written in C and has a minimal number of required

[5]Projects such as Overtone [11] and ScalaCollider [154] are examples of using the Clojure and Scala programming languages as front-ends to scsynth via OSC.

[6]For more information on SC3's UGen plugins, see [190, 25. Writing Unit Generator Plugins].

Required

- Tools
  - C Compiler (GCC, Clang, MSVC).
  - Autoconf.
  - Make.
- Libraries
  - Pthreads (non-MSVC platforms).
  - Audio API (either CoreAudio, ASIO, JACK, ALSA, or PortAudio, depending on availability).
  - TCL/TK (for pd-gui applications).

Optional

- Libraries
  - FFTW3 (falls back to built-in FFT routines).

Figure 4.7: Pure Data Dependencies

dependencies. The engine is capable of loading and running .pd patch files that are written in a human-readable text file format.

pd-gui – the primary tool that users use to create PD patches – is written in TCL/TK, which requires that a TCL/TK interpreter be available on the target platform. Since TCL and TK are generally available cross-platform, this simplifies porting of the graphical application across desktop systems. However, TCL/TK may not be available on other platforms, such as mobile ones like Android and iOS.

The canonical source for PD comes from its original author, Miller Puckette, and is often called PD-vanilla (as opposed to the PD-extended version that includes numerous extensions and changes). PD-vanilla contains both the runtime engine as well as the pd-gui. A separate project, libpd [39], reuses the source code for the engine part of PD-vanilla and provides an API for developers to use for embedding libpd within their programs. Architecturally, it would make sense for libpd to be a central part of PD-vanilla, with pd-gui being written as a client of libpd. However, the projects remain separate today.

Between the two, libpd is the more extensible in regards to platform extensibility when compared to pd-gui. This is primarily due to not requiring TCL/TK, a somewhat heavy dependency. libpd's platform extensibility is evidenced by the number of platforms it is available on, both in terms of operating systems (desktop, embedded, and mobile) as well as programming languages (C, C++, Objective-C, Java, Python).

One primary difference between libpd and pd-gui is that libpd is able to run a PD patch but it is not capable of authoring PD patches. Instead, the typical workflow involves authoring a patch on a desktop system with pd-gui, then running that patch on the target platform without the use of the standard GUI. In turn, the user of libpd would likely create a customised GUI, then run the PD patch and communicate with it using libpd's API. This workflow is somewhat similar to ways one can use Csound, but with the difference that PD's patch text format is not usually edited directly by users, while Csound's project format uses Csound text code that is normally edited by users.

Also like Csound and SC3, PD supports plugins known as *externals*. Externals allow for developer extensibility but limit project portability, which depends upon the plugin on all target platforms. However, this is a common problem for all software that offers plugin capabilities. Issues regarding platforms that lack plugin loading altogether will be discussed later in this thesis.

Overall, PD is a highly platform-extensible program with few required dependencies. pd-gui has been ported to numerous desktop and embedded systems, and libpd has ported to even further platforms, including mobile ones. While the pd-gui application is less portable than the engine itself, PD as a system has proven to be resilient over time and robust to address users' needs across platforms. The architecture is also well separated between the UI application and the engine, such that alternative UI applications may be developed.

4.4 CsoundObj: A Platform-Specific API

CsoundObj is a high-level, platform-specific API for developing musical applications with Csound. It provides pre-made customisations for the target platform and is designed to work well with the native development language. It shares amongst its implementations a common architecture and design.

The following will discuss the architecture of Csound and its layers of APIs. It will look at how its design served desktop platforms well, but also how certain assumptions within its design did not hold when moved to newer mobile platforms. Following this, the discussion will examine the design of the CsoundObj API and how it addresses the needs for platform-specific development while also working upon a portable core library.

4.4.1 The Architecture of Csound

The following will describe the high-level architecture of Csound. I will begin by discussing *libcsound*, which contains the core of Csound itself. Next I will discuss how Csound provides developer extensibility in two ways: firstly, by development of plugins, and secondly, by client use of libcsound and its API. I will discuss how the public API of libcsound is used by both plugins and library clients. These aspects of Csound's architecture form the base upon which the CsoundObj library was developed.

Core Library: libcsound

Csound employs a layered architecture that isolates portable code from platform-specific code. At its core is *libcsound*, a portable library that has two main library dependencies: *libsndfile* and *pthreads*. Beyond these two libraries, Csound requires either libraries available on POSIX-compliant systems (i.e., Linux, BSD, OSX) or Windows standard libraries. For tools, Csound requires Flex and Bison, as well as a C99-compliant C-language compiler. Csound uses the CMake build tool to generate build files for other build tools (e.g., GNU Make, XCode project files). Csound's dependencies are shown in Figure 4.8.

libcsound contains the essence of Csound. The library includes Csound's language compilers, core audio engine, and built-in set of opcodes. Any feature of Csound that has a third-party dependency outside of the standard C library, libsndfile, and pthreads, is handled externally to libcsound. These features may be supplied either through plugins or by host applications that use libcsound.

At this level of architecture, libcsound is only a library. One can not execute libcsound. In this state it is only of use to developers who would

Required

- Tools
  - CMake.
  - Build System (Make, XCode, Ninja).
  - C Compiler (GCC, Clang, MSVC).
  - Flex.
  - Bison.
- Libraries
  - Pthreads (non-MSVC platforms).
  - libsndfile.

Optional

- Libraries
  - gettext (for internationalisation).

Figure 4.8: Csound Dependencies

build applications or their own libraries based on Csound. Developers using libcsound will use the public Csound API, located in two places: the `csound.h` header file and the `CSOUND` struct itself. `csound.h` lists public function prototypes that host applications can use to embed Csound into their application. These functions include operations such as creating and running a Csound engine, as well as communicating with an engine via Csound's channel system.

The functions that make up the API in `csound.h` are also available as function pointers in the `CSOUND` data structure. This allows plugins to have access to the same functions as in `csound.h`, just by dereferencing the function

pointer from the CSOUND data structure. The reason for using function pointers for plugins is that one can compile plugins using just the Csound-provided development headers, without requiring linking to libcsound itself. Requiring plugins to link to libcsound was not a problem for Unix-style operating systems (i.e., Linux and OSX) but did cause problems on Windows.

Client Applications and Libraries

Csound started its life as a single, monolithic command-line program. In Csound 5, the core of Csound was isolated into libcsound and a public API was developed to allow building programs using Csound as a library. It was at this time that the command-line version of Csound moved from a monolithic application to become a smaller executable that itself linked to libcsound.

By having a standard core library and well-defined public API, developers had a clear way to embed Csound into their applications. Consequently, users of the library also all shared the same implementation. If one wanted to modify Csound for their own use, they could contribute a change and all users of the library would benefit.

Additionally, while Csound provides many features, the libcsound API design is such that it does not try to do everything itself. Rather, libcsound tries to provide all of the necessary lower-level functionality that would enable client applications to build what they need themselves. In this light, developers can use libcsound and *extend* the capabilities of Csound using their own application code.

libcsound is not only the basis for applications, but also other libraries. Csound's standard distributions include cross-language wrappers (also known as Adapters [70]) in C++, Python, and Java. These language-specific classes

and libraries wrap libcsound's API functions and data types and offer another API that is appropriate for their target languages. Adjustments for languages include presenting a class-based API as well as mapping data types from one language to another. For example, users may pass `String` objects to the Java API for Csound and the wrapper translates these to C's `char*` type when further calling a wrapped libcsound API function.

For C++, Csound provides a `csound.hpp` header file that offers an object-oriented, class-based view of libcsound's API. For Python and Java, an intermediary library, libcsnd6, is written in C++ to provide both an object-oriented API, but also additional glue classes that help to make certain aspects of Csound usage more idiomatic for the target language (e.g., expose a class for wrapping access to a-rate variables, rather than passing a pointer to the data to the host). SWIG (Simplified Wrapper and Interface Generator) — a *wrapper generator* program — is then used to analyze the libcsnd6 API and automatically create all of the wrapper code that bridges the target language and the native code.[7]

The relationship of applications and libraries to libcsound is shown in Figure 4.9.

Plugins

In addition to client extension, one can create plugin libraries that are loaded by libcsound. When a Csound engine starts, it will first load any plugins found

[7]The design and usage of SWIG is described in [29]. The application and documentation is available at [170]. As discussed in [171], SWIG is sometimes compared with *interface compilers* such as CORBA [83] and COM [124], but SWIG does not use an Interface Description Language (IDL), does not generate stubs, does not define protocols, and does not to define component models.

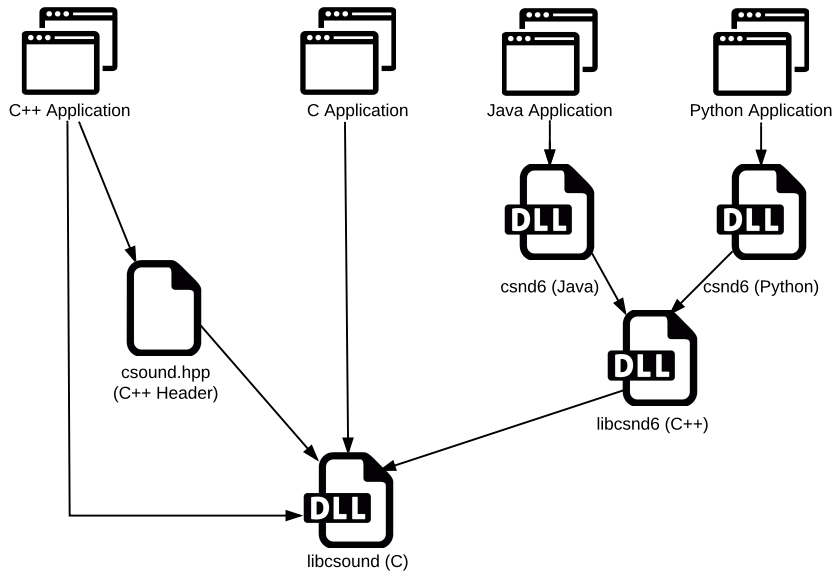

Figure 4.9: Relationship of libcsound to other libraries and applications

in the directory path defined in the `OPCODE6DIR64` environment variable, as well as any libraries explicitly given to the Csound program as an argument using the `-opcode-lib=` flag.[8]

When Csound first finds a library, it attempts to load it. If the library does successfully load, it means that its dependencies have been found and successfully satisfied. Upon a successful load, three functions are sought out in the library: `csoundModuleCreate()`, `csoundModuleInit()`, and `csoundMo-`

[8]The environment variable used depends upon the version of Csound used. Csound can be compiled to use 32-bit or 64-bit floating point precision for its processing. These correspond to float or double numeric types in C. For Csound 5, `OPCODEDIR` and `OPCODEDIR64` were used for float and double versions of Csound. This allowed users to have both versions of Csound installed on the same system and to load plugins from separate locations. For Csound 6, the variable names were changed to `OPCODE6DIR` and `OPCODE6DIR64`. This was to allow having both versions of both Csound 5 and Csound 6 installed on the same system. In Csound 6, the standard version distributed for desktop users is the 64-bit doubles version, so `OPCODE6DIR64` is used here.

`duleDestroy()`. These three functions are called at various points in the Csound engine life-cycle.

Each of the above functions takes in a single argument: a pointer to a `CSOUND` engine. Csound plugins use the Csound API through the function pointers provided as part of the `CSOUND` data structure. As noted earlier, these are the same API functions that are in the public Csound API in `csound.h`. Consequently, anything that a plugin is capable of doing – such as registering opcodes, audio drivers, MIDI drivers, and graphics drawing functions – a host application can do as well.

However, the opposite is not true: plugins can not extend Csound in all of the same ways as a host application. Plugins can not – or at least, should not – alter the flow of control of the engine. However, host programs can and are expected to operate the Csound engine however they would like. In this way, plugins are meant to add features to known extension points, while hosts may do that as well as add completely new abstractions and features on top of the Csound engine.

Discussion

The architecture of Csound is centered on the core libcsound library. Client applications build upon and use the Csound engine, and may also provide their own extensions to Csound. Plugins are loaded by Csound itself and are designed to only provide extensions.

Figure 4.10 illustrates the relationship between Csound, clients, and plugins. For plugins, they provide features that are available to all clients of libcsound. All features in Plugin X and Plugin Y are available to Client A and Client B. For clients, the features they provide are limited to their own

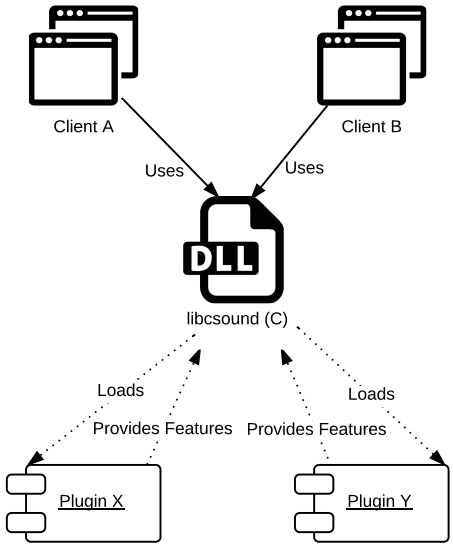

Figure 4.10: Csound, clients, and plugins

application. Any Csound extensions that Client A provides are not available to Client B, and vice versa.

The two methods of developer extension in Csound both employ the same Csound API. Their roles within the architecture of the system dictate when to develop a plugin and when to develop a feature within a client application itself. The ecosystem of Csound prior to the work in this chapter has largely been developed with the assumption that both extension mechanisms would be available. However, as will be explored below, not all platforms support plugin loading and features traditionally provided as plugins would have to be managed in a new way.

4.4.2 CsoundObj API

The CsoundObj API is a new API developed in response to issues found when developing Csound for iOS (Section 4.5.1), Android (Section 4.5.2), and the Web (Section 4.5.3). It builds upon the portable libcsound API

and handles platform-specific requirements previously addressed with plugin-based architecture. It also includes pre-made tools for quickly integrating platform-specific features into the user's work. It shares a common design amongst its implementations to provide easier porting of works from one platform to another.

Design

The design of CsoundObj differs somewhat from the libcsound API. With classical Csound usage, dealing with hardware I/O is done using plugins. Plugins in turn may interact with graphical user interface toolkits (e.g., FLTK Widget opcodes, Slider opcodes) or hardware I/O (e.g., MIDI I/O, Audio I/O). With the classical system, the features available to the user depends on what plugins are loaded. Conversely, if a project depends on a feature from a plugin that is not available, it is unable to run.

With CsoundObj, the design is inverted. Instead of extension by plugin, CsoundObj itself creates a libcsound CSOUND instance and extends the functionality by wrapping calls to the libcsound API. For example, rather than depend upon an Audio or MIDI I/O plugin to implement features, CsoundObj will directly register those callbacks with a Csound engine using the appropriate libcsound API methods. Also, instead of using opcodes to wrap hardware or GUI interface values, CsoundObj uses the Csound channel system for bi-directional communication with Csound. By not using plugins, these features are made always available when using CsoundObj.[9]

Csound's channel system is a generic, named bus where signals of different types can be sent into and read from Csound. At the time of this writing, `i-`,

[9]This is especially important as certain platforms do not provide runtime loading of plugins, such as iOS (Section 4.5.1) and the Web (Section 4.5.3).

k-, a-, and S-type signals are available for transfer over the bus. One benefit of this is that users can write Csound projects that depend on values coming from a channel, without concern for what is reading from or writing to that channel. The channel's readers and writers can thus differ from platform to platform: on one system, a value may be mapped to homemade hardware communicating to CsoundObj over a USB serial connection, on another it may be driven by values over a WIFI network, and on another it may be driven by a graphical user interface. Figure 4.11 illustrates the channel system.

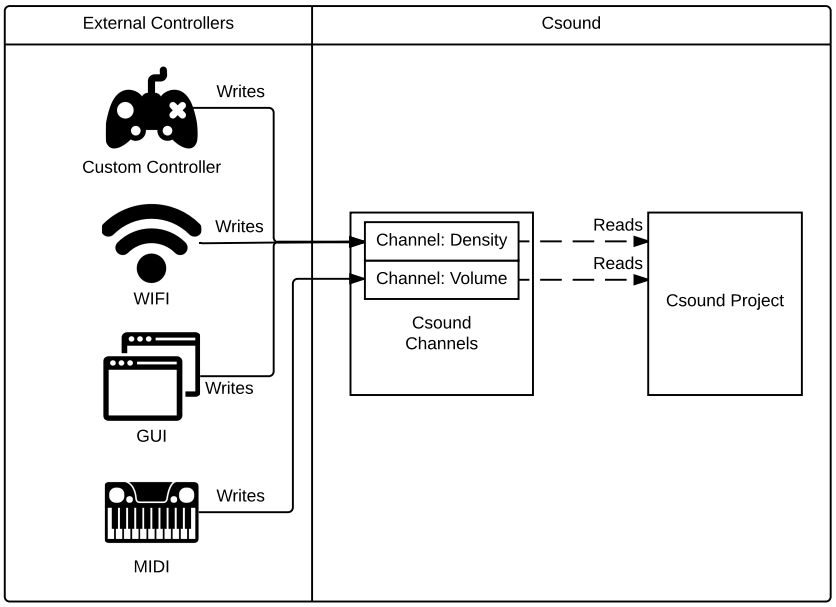

Figure 4.11: Csound Channel System

CsoundBindings

The CsoundObj library offers the CsoundBinding system to support arbitrary and generic communication with a Csound engine. To use this system, a developer will create an object that implements the CsoundBinding interface, then register that binding with CsoundObj. CsoundObj will in turn call

various methods within the CsoundBinding at various times during the execution of the engine. Bindings are run *synchronously* with the engine and have strong time guarantees on when they will be executed.

```
package com.csounds.bindings;

import com.csounds.CsoundObj;

public interface CsoundBinding {
  public void setup(CsoundObj csoundObj);
  public void updateValuesToCsound();
  public void updateValuesFromCsound();
  public void cleanup();
}
```

Listing 4.1: Android version of CsoundBinding interface

Listing 4.1 shows the Android version of the CsoundBinding interface. Figure 4.12 shows the life cycle of a CsoundBinding as it relates to the execution of a Csound engine by CsoundObj. The interface contains four methods described below:

setup()

Executes code to set up a CsoundBinding for run-time. This method is often used by the CsoundBinding to pre-calculate values as well as cache channel pointers, acquired from the passed-in CsoundObj object. If the CsoundBinding is registered with CsoundObj before the CsoundObj object is set into a running state, setup() will be called during the initialisation phase of the running state, just before the first samples are generated by the Csound engine. If the CsoundBinding is registered with a CsoundObj that is already in the running state, setup() will be called

before either updateValuesToCsound() or updateValuesFromCsound() are executed. This is done between block boundaries.

updateValuesToCsound()

Used to write values to a Csound engine. This is executed once per block of engine computation, prior to the block computation call to the Csound engine.

updateValuesFromCsound()

Used to read values from a Csound engine. This is executed once per block of engine computation, following the performance call to the Csound engine.

cleanup()

Called after a Csound engine performance has completed, but before the Csound engine has been cleaned up and released. This allows the object to release any resources it may have allocated or acquired.

CsoundBindings are a generic extension mechanism to the Csound engine. They are run synchronously with the engine. Developers using a CsoundBinding with their CsoundObj instances can choose how much work to do in the binding. For example, a single CsoundBinding may be used that will update all inputs to and outputs from the Csound engine for the application. Another strategy may be to use multiple CsoundBindings, using one per graphical user interface widget (i.e., sliders, knobs). Because the interface only defines when things will happen and not what they will do, the developer is in complete control on how to use the system.

For the CsoundObj API, all pre-made GUI element wrappers and hardware I/O wrappers are implemented as individual CsoundBinding implementations.

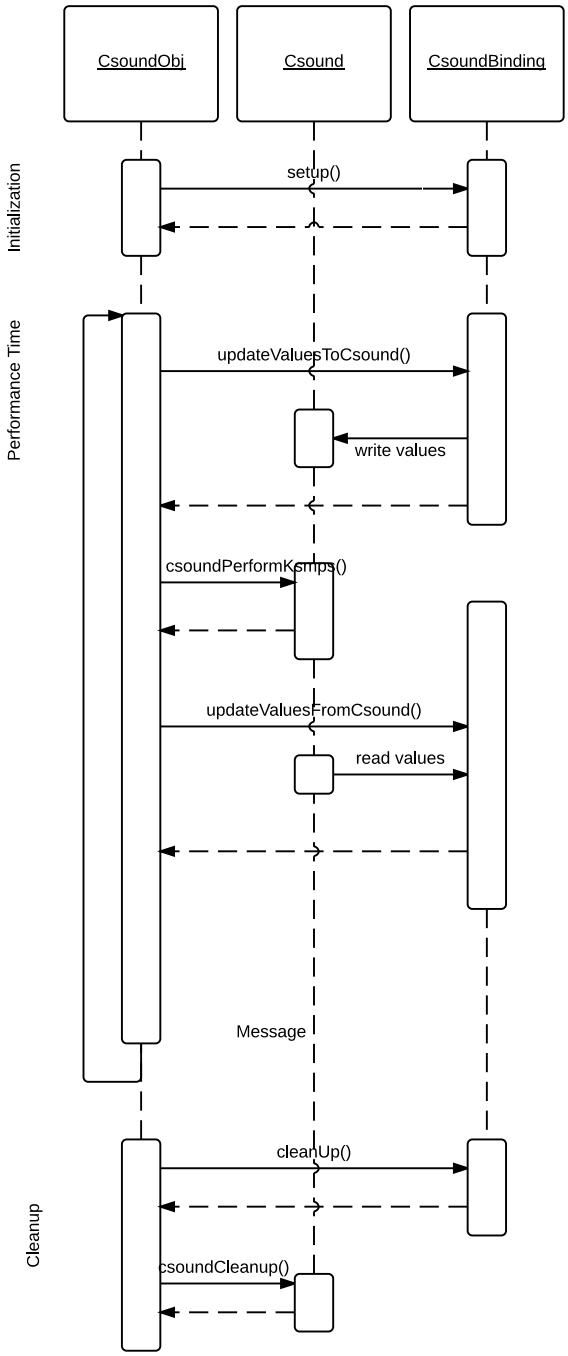

Figure 4.12: CsoundObj, Csound, and CsoundBinding Life Cycle

For example, if a developer uses the CsoundUI helper class to add a slider binding to a CsoundObj instance, a single CsoundSliderBinding will be created and registered. After registration, the current value of the slider will be sent to Csound via the channel system each time a block of samples is generated. If a user uses the CsoundMotion helper class to enable reading from a device's accelerometer, a CsoundAcceleromter binding will be registered with the engine that writes the current value of the accelerometer into a Csound channel.

Discussion

The CsoundObj API was designed to handle the most common use cases for features for applications. It was designed to be platform- and language-specific, offering an API in a way that was idiomatic to the developer on the target platform. CsoundObj comes pre-wired for real-time and non-real-time audio rendering, common GUI widget wrappers, and MIDI I/O handling (where available by platform). Additionally, for uncommon use cases, CsoundObj exposes the libcsound `CSOUND` instance so that users can fallback to using the lower-level libcsound API.

The CsoundObj API was first designed as part of the Csound for iOS software development kit (SDK). The design was then reimplemented in the Csound for Android SDK and was an influence on the Web versions of Csound. While the CsoundObj implementations differ in what they can offer, they do all follow the same architecture. Also, they do follow each other in class and method naming; as such, users can more easily port their own application code from one platform to another. The following section will further explore the

design of CsoundObj and its role in developing platform-specific Csound-based SDKs.

4.5 Extending Csound to Mobile and the Web

This section will explore the work of porting Csound to three new platforms: iOS, Android, and the Web. Extending Csound to these platforms offers users new ways and places to employ Csound. It has also increased the focus of using Csound not only through its language but also through its engine and API as the core of custom music applications.

Common goals for each platform include:

- Running existing Csound projects on these new platforms.
- Allowing content creation on these platforms (i.e., users can author Csound-language projects on the target platforms).
- Building music applications using Csound.
- Simplifying cross-platform development of Csound-based music applications.
- Allowing users to take advantage of platform-specific features, but provide a graceful fallback solution if features are not available.

This section will first cover two mobile platforms, iOS and Android. Next, it will discuss two implementations of Csound for the Web: one using Emscripten, the other using Portable NativeClient (PNaCl). A summary of this work will conclude this section. Applications of the technology developed here will be further explored through case studies in Section 4.6.

4.5.1 Csound for iOS SDK

Csound for iOS is the name of the software development kit created for building Csound-based iOS applications. The SDK was first released with Csound 5.17.3 and new versions have been released with each new version of Csound. The SDK includes statically compiled versions of libcsound and libsndfile, development headers for working with libcsound, class files and headers for the CsoundObj API, and an examples project that demonstrates various use cases for how to use the SDK. A manual is also included.

The Csound for iOS SDK is released together with each release of Csound. The version of Csound included in the SDK is built using the same source code as that used for desktop Csound and other platform releases. By building using the same source code as other releases, users can be assured that all bug fixes and features available in new versions of Csound are also available when using Csound on iOS.

About the Platform

iOS [25] is a closed-source, BSD-based operating system developed by Apple. It is available only on the company's iPhone, iPod Touch, and iPad devices. The operating system's popularity is closely tied to the popularity of the devices themselves. Developers programming for iOS primarily program in Objective-C or Swift, though they may also use C and C++. It is the compatibility with C and C++ that allows for many libraries and applications from desktop platforms to be easily compiled and used on iOS.

Regarding the hardware, the CPUs available in the devices for iOS have been ARM-based processors. These CPUs are available in 32-bit and 64-bit; devices prior to iPhone 5 were 32-bit, while those made after are mostly

64-bit. The ARM CPU architecture is big-endian, which is common to other embedded CPUs such as those available from MIPS but differs from little-endian CPUs – such as those made by Intel and AMD – that are available in most desktop and laptop devices.

In terms of libraries and linking, iOS differs from OSX. OSX has three main types of libraries: static libraries, shared libraries, and frameworks. Static and shared libraries are commonly found on desktop systems and allow one to link an application or library binary to another library either at compile-time or at runtime. Frameworks are an Apple invention that packages libraries (either static or shared), development headers, documentation, and other resources into a specified folder structure.

iOS prior to version 8.0 did not allow for use of dynamically-linked shared libraries outside of those found in frameworks provided by Apple. Developers of third-party libraries were then required to build and distribute static libraries. While there was little technical difference between using static and dynamically-linked libraries, there were implications for compatibility with open-source licensing. Since iOS version 8.0, the use of dynamically-linked libraries is now permitted, opening up the use of many open-source libraries in applications.

Platform Analysis

Developers building applications for iOS will find the technologies and workflow very similar to those available when developing for desktop applications on OSX. iOS provides many of the same build tools and development frameworks as provided on OSX. Users used to building music applications using

CoreAudio and CoreMIDI will find that much of their code can be reused when moving to iOS.

Development of iOS applications does not happen directly on the device. Instead, applications are written on a desktop system, then cross-compiled from the native system to the target architecture (i.e., ARM7, ARM64), before being deployed and installed on the device to be executed. This style of development with cross-compilation is common for embedded systems, especially considering the era in which the earliest iOS devices were made and the processing power available at the time. While CPU speeds on iOS devices has increased greatly, it is likely that the style of development with cross-compilation will continue on.

For Audio and MIDI, iOS provides the same CoreAudio and CoreMIDI frameworks that are found on OSX. CoreAudio provides a low-latency audio system for all iOS devices; CoreMIDI provides a single, consistent way to interact with MIDI devices connected to the device either physically or by network. As all hardware devices are tightly controlled by Apple, there is a consistency of what to expect on all iOS devices that makes it simpler to develop multimedia applications, when compared to more hardware-heterogeneous platforms such as Android.

For user interfaces, developers will generally use the UIKit framework provided by iOS. This framework also exists on OSX but the two platforms differ in what classes and features are provided. Developers can also opt to use alternative toolkits such as QT [149].

In general, iOS is a more homogeneous platform than others such as Android. Screen resolutions are limited to only a few sizes and I/O characteristics for Audio and MIDI are consistent across devices. This simplifies

development for the developer, who has less variables to account for when developing for iOS.

SDK Design

The Csound for iOS SDK is designed in three parts. The first is the core Csound library, *libcsound*. The version of libcsound that is built uses the same sources as it does when it is built on desktop, and the same development headers are provided on iOS as they are in the desktop releases. libcsound is built and provided as a static library, together with a static library for libsndfile – the only required dependency libcsound has.[10] These two libraries plus the Csound development headers comprise the core of Csound.

Users who develop applications on the desktop as well as on iOS have access to the same C API. One difference between desktop and iOS versions of Csound is that the desktop releases of Csound include command-line executables as well as dynamically-loaded Csound plugins. These plugins can include opcodes but they may also be I/O providers for interfacing with audio and MIDI systems. Opcode and I/O driver plugins are not available on iOS because iOS does not allow the building of applications that can dynamically load plugins.

With libcsound, the user has full access to the core of Csound, comprised of over 1400 opcodes, and the main Csound engine itself. However, because no audio drivers are available, the user would have to know how to use Csound as a library and know how to read and write samples from and to a running Csound engine. The core of Csound can be used out-of-the-box; however, one

[10]This is as of Csound 6.04.0. Shared library versions of Csound and libsndfile have been built by third-parties, and plans are to change the core Csound for iOS release to only provide shared libraries in the future.

can not drop it in a project, add a few lines of code and expect to generate and process real-time audio.

Rather than modify the sources of libcsound through introducing conditional code specific to iOS, a different design was implemented using a higher-level library called CsoundObj. As discussed in Section 4.4, the CsoundObj API was introduced to simplify development when building Csound-based applications. The main CsoundObj class comes with a standard set of methods for instantiating, running, and communicating with an instance of Csound. It is written in the primary language of the platform – Objective-C – and thus is designed to be familiar for those developing for iOS. Additional methods were added to simplify communication between Csound and application code.

The CsoundObj implementation for iOS handles communication between a running Csound engine instance and CoreAudio (the audio system on iOS). `CsoundMIDI`, `CsoundMotion`, and `CsoundUI` helper classes are provided to perform MIDI, sensor, and GUI value binding with a `CsoundObj` instance. These helper classes all use the CsoundBinding system to provide synchronous value communication between the the bound component and Csound channels.

The motivation for the design was to make native Objective-C development easy to do using the CsoundObj class. The CsoundObj class is not designed to provide everything the primary Csound C API provides. Rather, it provides the most commonly used items in a way that follows the common practices of the platform (e.g., CsoundObj is written to use `NSString` values, rather than `char*`, as `NSString` is more commonly used in iOS development). The CsoundObj API does provide access to the `CSOUND` engine instance should the user require more of the Csound API than is provided through CsoundObj. Providing the higher-level, more platform-appropriate API, while

also providing the ability to access the full Csound API, gives a good balance between ease of use and full development potential.

Figure 4.13 illustrates the relationship of the user's application source and the Csound for iOS SDK library parts. Application code may use either CsoundObj or libcsound, or both, and CsoundObj uses libcsound itself.

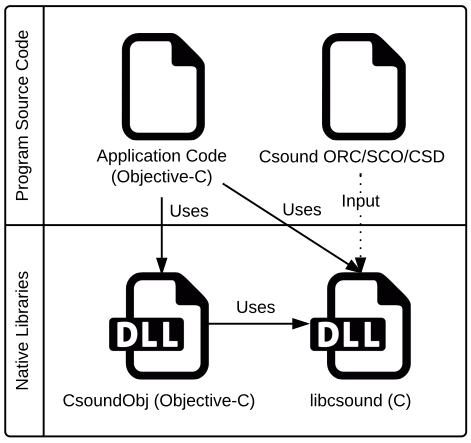

Figure 4.13: iOS CsoundObj Diagram

The final part of the SDK is the manual and examples project. The manual provides information about the design and usage of the CsoundObj API on iOS. The examples project provides working code that demonstrates individual aspects of using the API. By providing working code, the user can copy and use the code to quickly start off their music projects.

Porting Csound to iOS

As Apple's iOS development SDK provides many of the same build tools, libraries, and frameworks as the OSX desktop, porting Csound to iOS was fairly straightforward. The build scripts for Csound on iOS reuse the same CMake build files used on the desktop to generate XCode project appropriate for iOS. In turn, the `xcodebuild` command-line tool is used to build the

project twice, first with an `.xcconfig` file containing settings appropriate for hardware devices, then second with an `.xcconfig` file appropriate for the iOS simulator. These `.xcconfig` files contain different settings to use for compilation, with the primary difference being what CPU architectures to target. The resulting libcsound libraries are then joined using the `lipo` tool into a single universal binary library.

For libcsound on iOS, no new build-time challenges were found. The existing build system for Csound already accounted for CPU-related issues such as CPU endian-ness. All library and development headers that required checks were already accounted for.

The primary challenge in bringing Csound to iOS was the lack of runtime library loading on the platform. This meant that no plugins could be used with Csound on iOS and that all existing I/O functionality would have to be reconsidered. This was a new architectural challenge, as previously supported platforms for Csound did support plugin loading.

The result was the development of the first implementation of the CsoundObj API. Here, client-based extension was used by CsoundObj to supply the missing functionality that was previously supplied by plugins. This was done for required features to operate Csound for real-time audio processing. All other optional features provided by plugins were not redeveloped for the iOS implementation of Csound.

Summary

Csound for iOS provides a working version of Csound ported to the iOS platform. The work in porting Csound to iOS provided a blueprint for how to approach platforms that do not provide library loading. This in turn

manifested in the CsoundObj API design. The resulting Csound for iOS SDK has been successfully released together with each new version of Csound and used for personal and commercial work on iOS.

4.5.2 Csound for Android SDK

Csound for Android is the name of the software development kit created for building Csound-based Android applications. The SDK was first released with Csound 5.17.3 and new versions have been released with each new version of Csound. The SDK includes shared library versions of libcsound and libsndfile [57], binary class files and headers for the Java Csound API, and Java source files for the CsoundObj API. The SDK also includes an examples project that demonstrates various use cases for how to use the library and a manual is also included.

The Csound for Android SDK is released together with each release of Csound. Like the Csound for iOS SDK, the version of Csound included in the SDK is built using the same source code as that used for Csound desktop and other platform releases. This provides the same benefits as discussed for the Csound for iOS SDK.

About the Platform

Android [76] is a Linux-based platform for mobile devices. On top of the Linux core are two runtimes: Dalvik and Android Runtime (ART) [19]. Dalvik was the initial runtime used on Android and it executes platform-independent bytecode using a just-in-time (JIT) compilation model, similar to the Java Virtual Machine. ART provides ahead-of-time (AOT) compilation model

that compiles the same platform-independent bytecode to native code before running.

Developers primarily use the Java programming language to program Android applications. The Java source is first compiled into Java bytecode, then translated into Dalvik bytecode, before being run on the Android platform. Pre-compiled Java classes may also be used with Android projects and will be translated into Dalvik bytecode format by the platform's build tools.

Developers may also use C and C++ to generate hardware-dependent libraries and applications. For Android, developers typically compile dynamically-linked libraries that are then used from Dalvik-compiled programs via the Java Native Interface (JNI) [115]. Libraries must be compiled with support for each CPU-architecture that the application targets.[11] Developers also have the option to write their entire application in natively-compiled C/C++ code, though this is less commonly found than the Java-language based approach.

As the Android platform is open-source, it has been used on a much wider variety of hardware platforms when compared to iOS. Android supports not only ARM-based processors but also Intel and MIPS-designed CPUs. These processors vary in endian-ness and word-size (32- and 64-bit). Using natively-compiled code requires that the library or executable be compiled multiple times, once for each target CPU architecture. On the other hand, Dalvik bytecode is platform-independent; once written, it is JIT- or AOT-compiled on each platform.

[11]For more information on supported CPU architectures for Android are available, see [21].

Android supports both static and dynamic linking of libraries. Static linking is supported when compiling native code. Dynamic linking is supported both at compile-time as well as runtime.

Platform Analysis

Developers building applications for Android most commonly use the Java programming language using the Android Software Development Kit (SDK). They may also use C and C++ using Android's Native Development Kit (NDK). Like iOS, development is performed on a desktop system and cross-compiled to run on Android devices. For projects using only Java code, the code is compiled once into platform-independent bytecode and run on multiple CPU architectures. For those using C/C++ code, the code is compiled once per target CPU architecture.

Audio services on Android are most commonly accessed either through the Dalvik layer using the system-provided AudioTrack class or through the native layer using the OpenSL ES C API. At the time of this writing, Android does not provide a standard MIDI API.

For user interfaces, users developing in Java will use the classes in the system-provided `android.widgets` package. Developers may develop their interfaces directly in Java code but they more commonly use XML interface files that declaratively specify the user-interface. These are either hand-written or developed using a GUI editing program. Native development options are also available, such as using the QT widget toolkit.

In general, developers writing applications for Android face a much more heterogeneous target than iOS. Differences include a wider variety of screen-sizes, CPU-types, as well as capabilities supported (e.g., sample rates allowed

for audio streams). Android as a whole does not support the same low-latency as provided by iOS.[12]. While Android's latency limitations make the range of supported music application use cases smaller, the ubiquitousness and open-source nature of Android make it an attractive target platform.

SDK Design

The Csound for Android SDK was designed to mirror the design of the Csound for iOS SDK as closely as possible, with appropriate differences made to adhere to conventions common to Android. While applications may be developed purely in C and C++, it is more common on Android to use Java, thus that was the target user for this project.

The base of the Csound for Android SDK begins with the same Csound Java API that is available on the desktop. libcsound and libsndfile are provided as shared libraries. A Java language wrapper for Csound is generated using SWIG [29], as it is on the desktop. In addition to the standard libcsnd6 sources, one additional C++ class, called `AndroidCsound`, is also compiled into the library. This class is a subclass of the `Csound` class and adds one additional method to register native OpenSL audio callbacks with Csound. This class is also wrapped with SWIG. The Java AudioTrack class is also supported by CsoundObj on Android, but the OpenSL audio path is the default and recommended path to use for performance reasons.

Like Csound for iOS, a CsoundObj API implementation is provided that builds upon the lower-level Java csnd6 API. The methods and classes implemented in the Android version of CsoundObj are closely named to their iOS counterparts. The reason for this was to simplify creating cross-platform

[12]Measurements for common Android devices are available in [20].

applications between the two platforms. Although the programming languages are different between iOS and Android, both are object-oriented languages. By using the same class designs and similarly named methods, users can easily translate application code between the platforms.

Figure 4.14 illustrates the relationship of the user's application source and the Csound for Android SDK library parts. Like Csound for iOS, application code may use either CsoundObj or csnd6 Java classes. These Java class libraries in turn communicate over JNI to the natively-compiled libcsnd6 library, built upon libcsound.

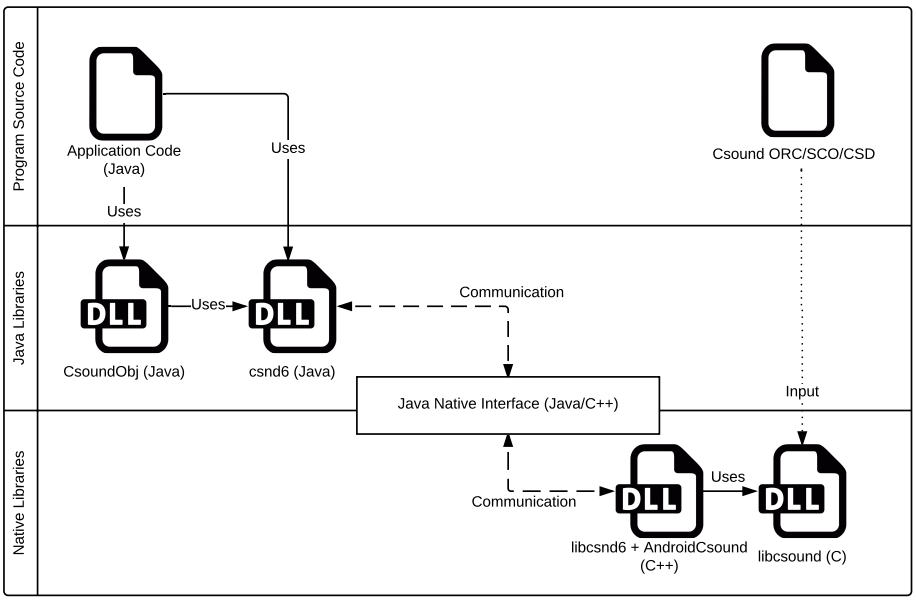

Figure 4.14: Android CsoundObj Diagram

Listing 4.2 illustrates the API parity between CsoundObj implementations on iOS and Android. The code fragments both create an instance of `CsoundObj`, register the current class as a listener to `CsoundObj`, load a Csound CSD project file, then use the `CsoundMotion` class to enable different hardware sensors. The iOS version enables accelerometer, attitude, and gyro-

scope, while the Android version only enables the accelerometer, as Android does not support the other two hardware sensors.

```
/* iOS Objective-C Example */
NSString *tempFile = [[NSBundle mainBundle]
  pathForResource:@"hardwareTest"
  ofType:@"csd"];
self.csound = [[CsoundObj alloc] init];
[self.csound addListener:self];

CsoundMotion *csoundMotion = [[CsoundMotion alloc]
  initWithCsoundObj:self.csound];

[csoundMotion enableAccelerometer];
[csoundMotion enableAttitude];
[csoundMotion enableGyroscope];

[self.csound play:tempFile];

/* Android Java Example */
CsoundObj csoundObj = new CsoundObj();
csoundObj.addListener(this);
String csd = getResourceFileAsString(R.raw.hardware_test);
File f = createTempFile(csd);

CsoundMotion csoundMotion = new CsoundMotion(csoundObj);
csoundMotion.enableAccelerometer(AccelerometerActivity.this);

csoundObj.startCsound(f);
```

Listing 4.2: Android and iOS CsoundObj example

One of the goals for Csound on these platforms is that the same Csound CSD project may be used by either Android or iOS applications, even if there are differences in available features. When this happens, the project should degrade gracefully. In the case of Listing 4.2, the iOS version sends values using Csound's channel system that the Android version would not. On the receiving end in the CSD, as shown in Listing 4.3, the `chnget` opcode is used to read from channels where hardware sensor values are written. On iOS, all nine of the channels will receive data, while on Android, only the three accelerometer channels will receive data. For the other six channels on Android, the default 0 value for channels will be read.

```
kaccelX chnget "accelerometerX"
kaccelY chnget "accelerometerY"
kaccelZ chnget "accelerometerZ"

kgyroX chnget "gyroX"
kgyroY chnget "gyroY"
kgyroZ chnget "gyroZ"

kattRoll chnget "attitudeRoll"
kattPitch chnget "attitudePitch"
kattYaw chnget "attitudeYaw"
```

Listing 4.3: Csound Channel Reading Code

The example Csound code shown can run on either iOS or Android *without* modification. Additionally, this project can run on any other version of Csound as well, such as on desktop platforms, where no sensors may be available and all channels give 0 values. By having the default values for channels, this simplifies porting the Csound project to various platforms and minimises the worst case scenario such that at least the project will run.

The Csound for Android SDK provides the same SDK features as Csound for iOS. It includes the pre-built native libraries and CsoundObj API, a Csound Examples project, and a user manual. The examples project provides the same examples as those found in the Csound for iOS SDK, with the exception of MIDI examples as Android does not support MIDI. The CSDs used in the example project are the same ones from the iOS SDK, used without modification.

Porting Csound to Android

Google provides both an Android SDK and NDK (Native Development Kit). The SDK provides Java-language, Dalvik bytecode related build tools, while the NDK provides C/C++ build tools. For the NDK, the provided development headers and libraries and tools resemble closely to those commonly found on desktop Linux distributions.

When Csound for Android work commenced, it was not clear how one would use CMake with Android's NDK tools. Instead, a typical `Android.mk` and `Application.mk` file were written that would work with the NDK-provided `ndk-build` tool. This made the method of building the native libcsound library consistent with Android-specific development practices, though required having a separate build recipe than the other main Csound builds. The result is that the Android build files must be manually kept in sync with the primary Csound CMake build recipe. This has added a small amount of additional work for the core Csound developers but simplified building Csound for Android.[13]

[13]It would be ideal to have an Android build based on CMake. Work towards that goal is reserved for future research.

Like the desktop Csound Java API, SWIG is used to generate both C and Java class files that wrap Csound. The generated Java classes are compiled and packaged for developers to use. The generated C files are compiled as part of the libcsound.so library.

To simplify the runtime loading of the native library portion of Csound, all of the native components are compiled together. This includes libcsound, libsndfile, generated SWIG C files, and the additional `AndroidCsound` C++ class. While these could have been compiled into separate libraries, there was little reason found to compile them separately for this project. The decision to provide one monolithic library can be easily changed at a later time should a compelling reason be found.

Unlike iOS, dynamic library loading of plugins is available on Android. However, due to the nature of Android's sandboxed applications, support for multiple CPU architectures, and concerns for application sizes, it was easier to follow the design used for iOS and provide a CsoundObj API that handled I/O and other features. However, plugins for Csound have been developed and used on Android and are provided as optional features for developers to use.

While the build-time issues were simple to resolve, there was one large issue found in Csound to Android: Csound's use of temporary files to store pre-processed score did not work. Prior to development with Android, when Csound would load a score file, it would process the score language into a set of score events, write the processed score to a temporary file on disk, then open an input stream with the temporary file to read in score events at runtime. Reading from disk limited the number of events in memory, saving

space. However, since creating temporary files was problematic, the process of score rendering was changed.

To support this situation with Android, ffitch introduced the new CORFILE system. The CORFILE system mimics the same file saving, opening, and reading functions that were in use with Csound. However, instead of reading from disk, the CORFILE system reads from and writes to memory. Since memory is much more abundant today compared to when Csound was first developed, this solution seemed like a reasonable one to pursue.

The CORFILE API provides a near drop-in replacement for the set of file I/O functions used previously for working with temporary files. Listing 4.4 shows the data structure for a CORFIL. It contains a `char*` body, the length of the body, and the current position `p`. CORFILs function much like RAM-based files and are processed using the same kinds of operations as found for file-based I/O.

Listing 4.4 also shows the function prototypes for working with CORFIL structures. The CORFILE API provides funcions for creating and working with single file-like entities, but it does not try to implement a full RAM-based file system (i.e., it does not support directory listing, file metadata, etc.). The API is small in size, uses only standard C library functions, and does not use platform-specific functionality. These qualities make the CORFILE system easily portable across platforms.

```
// include/csoundCore.h:197
typedef struct CORFIL {
  char    *body;
  unsigned int     len;
  unsigned int     p;
} CORFIL;
```

```
// H/corfile.h
CORFIL *corfile_create_w(void);
CORFIL *corfile_create_r(const char *text);
void corfile_putc(int c, CORFIL *f);
void corfile_puts(const char *s, CORFIL *f);
void corfile_flush(CORFIL *f);
void corfile_rm(CORFIL **ff);
int corfile_getc(CORFIL *f);
void corfile_ungetc(CORFIL *f);
#define corfile_ungetc(f)  (--f->p)
MYFLT corfile_get_flt(CORFIL *f);
void corfile_reset(CORFIL *f);
#define corfile_reset(f) (f->body[f->p=0]='\0')
void corfile_rewind(CORFIL *f);
#define corfile_rewind(f) (f->p=0)
int corfile_tell(CORFIL *f);
#define corfile_tell(f) (f->p)
char *corfile_body(CORFIL *f);
#define corfile_body(f) (f->body)
char *corfile_current(CORFIL *f);
#define corfile_current(f) (f->body+f->p)
CORFIL *copy_to_corefile(CSOUND *, const char *,
    const char *, int);
CORFIL *copy_url_corefile(CSOUND *, const char *, int);
int corfile_length(CORFIL *f);
#define corfile_length(f) (strlen(f->body))
void corfile_set(CORFIL *f, int n);
#define corfile_set(f,n) (f->p = n)
void corfile_seek(CORFIL *f, int n, int dir);
```

```
void corfile_preputs(const char *s, CORFIL *f);
```

Listing 4.4: CORFILE data structure and function prototypes

After implementing the CORFILE system, ffitch also updated the score processing code in Csound to use the CORFIL system. The result of this is that temporary files were no longer used in Csound's score processing, and Csound was then able to render without problems on Android. Because the source code is shared across all Csound-supported platforms (i.e., Windows, OSX, Linux, iOS, etc.), the problem of temporary files has been solved for all current and future platforms.[14]

Summary

Csound for Android provides a working version of Csound ported to the Android platform. It employs the same architecture and design as Csound for iOS, making it easy to port applications from one platform to another. The parity of code and use of same CSDs in the corresponding examples projects demonstrate the viability of cross-platform, Csound-based music application development as well as provide a set of models to use in building new applications. The work also identified the limitation of disk-based temporary file usage and work by ffitch has solved that problem for new platforms moving forward. The resulting Csound for Android SDK has been successfully released with each new version of Csound and has been used by developers and users for their personal and commercial work on Android.

[14]This proved to be immediately useful in porting Csound to the Web, where temporary file usage would also have been problematic.

4.5.3 Csound on the Web

This section will discuss Csound on the Web – two versions of Csound that work cross-platform among various browsers and operating systems. The first version is built using the Emscripten [205] compiler and is a pure-Javascript version of Csound that runs in any browser supporting the WebAudio API [191]. The second implementation is a Pepper API (PPAPI)-based [80] plugin built using Google's Portable Native Client (PNaCl) system [78] that runs cross-platform in Chrome and Chromium browsers.

The goals in extending Csound to the Web are to:

- Run Csound on the client-side in a browser.
- Run Csound without the user having to additionally install anything besides the browser.
- Run existing Csound code projects without modification.
- Author new Csound code content within a browser.

Being able to reuse Csound within the browser offers existing Csound users a way to apply their existing knowledge to create web-based projects, without having to learn or create a new computer music system. Conversely, new users who might learn to use Csound in a browser have the opportunity to transfer those skills to create desktop and mobile-based music applications. Finally, by creating web pages that can render Csound code without requiring installation of any plugins or applications, the web offers a solution for both long-term preservation of works as well as ubiquitous sound and music computing.

The following will begin with setting out common design goals for both Web versions of Csound. Next, the implementation of Emscripten and PNaCl

ports will be covered separately. Finally, a comparison of the two ports will be provided.

Design

When Lazzarini, Costello, and I first started looking at Csound on the Web, our technical goals were to:

- Have a complete version of Csound available for building web applications.
- Build the web version of Csound using the same source code as the desktop and mobile versions of Csound.
- Create real-time and non-real-time music applications written using HTML, Javascript, and Csound.
- Provide simple deployment of applications and pieces.
- Have applications run client-side within the browser, using web technologies that do not require any installation of plugins.
- Be cross-platform.

It was important to the Csound developers that we look at solutions that did not require rewriting Csound in another language (i.e., Javascript), but that instead used the same C source code as the desktop and mobile versions of Csound. Using a common codebase was critical, as it would have been time consuming for the community to maintain and test a separate version of Csound. A common codebase also ensured that the same Csound music code would render in exactly the same way regardless of what platform it is on, including the Web.

We also wanted the development experience of web applications using Csound on the Web to mimic very closely the experience of developing applications on the desktop and mobile platforms. The expectations are that the same Csound music code could be used in any project, and the only code that would need porting would be specific to each target platform. To achieve that end, we wanted to provide a Csound API implementation that was very similar to those provided on other platforms. That way, users who start by building a web application could more easily port their code to a mobile or desktop platform, and vice versa.

Furthermore, we wanted to ensure that Csound-based web applications were cross-platform across operating systems. We also wanted the implementation of Csound on the Web to be cross-browser, though this was less of a priority than working across operating systems. Working across operating systems meant that a user could create a project and have it run on as many computers as possible. Working across browsers would add an additional level of reach so that users could open projects in their own preferred browser.

After evaluating the available client-side web technologies, two systems stood out: Emscripten and PNaCl. These two systems provide viable solutions for the goals we wanted to achieve. However, the two systems also come with certain tradeoffs. The following will discuss to the two technologies and the approaches we took in employing them. Afterwards, I will provide an analysis of the tradeoffs between the two systems.

Emscripten

Emscripten [65] is an LLVM-based compiler technology that allows cross-compilation from C and C++ into a subset form of Javascript called ASM.js [88].

As ASM.js is a subset of Javascript, code produced by Emscripten runs in any Javascript interpreter. However, certain browsers, such as Firefox, are optimised for ASM.js and are capable of running ASM.js code faster than standard Javascript code. Emscripten has been used to port games and other applications to run within the browser.

As the result of running Emscripten is ultimately Javascript, the limitations of using this technology are mostly a result of the services and technologies provided by the browser itself. For Javascript, the current standard to generate and process audio is done using the WebAudio API. (The WebAudio API, in its current form, has limitations that influence the usability of Csound. These limitations will be discussed below.) Also, one of the biggest hurdles for Emscripten use at this time is that it does not support threads – in particular, the cross-platform pthreads library.

Getting Csound to operate in the browser via Emscripten was largely done in two parts. Firstly, we wanted to get libsndfile and the standard libcsound C API compiled and available as a Javascript library. Secondly, we needed to wrap the compiled library to connect it to the audio system of the browser via the WebAudio API. The first part was done primarily by myself and required modifying Csound's source files and CMake build files to compile Csound with Emscripten's toolchain. The latter part was done primarily by Edward Costello to design and implement the WebAudio connection with Csound, as well as build up the platform-specific CsoundObj-like API for this project.

To get Csound compiled with Emscripten, we first had to build libsndfile. Using libsndfile's own autoconf-based build, together with Emscripten's build toolchain, we found we only required one patch [201] to the source code to compile libsndfile into an LLVM bytecode (.bc) library. The generated target

at this stage is LLVM bytecode as this is later used for linking, before the final Javascript code is generated.

Compiling Csound with Emscripten was a larger challenge due to Csound's dependence on the pthreads library. Originally, a platform-specific, Csound-provided threads implementation [10] was used throughout the codebase. Later, the platform-specific parts were removed and only two implementations remained: one implemented with pthreads and the other providing a dummy implementation. However, as the pthreads library was set as a required library in the build system, the dummy implemented was no longer used. Also, as pthreads was assumed to always be available, we found that pthreads-provided functions were being used directly in the codebase.

To get around the issue of pthreads, both the build system and the source code were changed. Firstly, the CMake build system was modified to ignore searching for pthreads if and only if building with Emscripten. This change was limited to the Emscripten-build to ensure that if pthreads was not found on other platforms where it *should* be available, then it would cause a build-time failure. After the changes in CMake, code within the Csound codebase was modified with pre-processor conditional checks to optionally compile certain code when building with Emscripten.

The result was that libcsound could be compiled without thread support. This allowed for single-threaded use of Csound, with all of the same built-in opcodes and features as is found on all other libcsound builds (i.e., Desktop, Mobile). For the purpose of using Csound with WebAudio, being single-threaded was sufficient.

After libcsound and libsndfile were compiled, Costello then wrote an implementation of CsoundObj in Javascript. This version of CsoundObj, like

the Android and iOS versions, comes pre-configured to handle Csound and platform-specific I/O to work with the native audio system (i.e., WebAudio). It does this by creating a WebAudio ScriptProcessorNode [125] that manages and runs an instance of a Csound engine. The ScriptProcessorNode is responsible for transferring incoming samples to Csound, running the Csound engine for n number of buffers, and transferring samples back from Csound to WebAudio.

The current version of Emscripten CsoundObj does not provide the CsoundObj Binding system found in Android and iOS.[15] Instead, users read values from and write values to Csound using its channel system. Whether Bindings will be implemented is currently on hold as the future of WebAudio is moving away from the ScriptProcessorNode and towards AudioWorkers. With the the current state of AudioWorker design, AudioWorkers will run in a separate thread and can not share memory with the main JS thread. This would make it impossible to implement a synchronous Bindings system. It is unknown whether Bindings can be implemented for Emscripten Csound until the final design of AudioWorkers is complete.

Like iOS and Android, a Csound Emscripten SDK is now released together with every new version of Csound. The SDK provides a pre-compiled version of libcsound.js (generated from libcsound and libsndfile), CsoundObj.js, and a set of HTML-based examples that demonstrate usage of Emscripten Csound and also act as a manual for the project.

[15]Version 6.05.0 of Csound, as of the time of this writing.

PNaCl

Another technology, Google's Pepper API (PPAPI) and Portable Native Client (PNaCl), offers a different path to developing cross-platform, web-based audio applications. With PNaCl, C and C++ code can be compiled into a portable bytecode that is in turn ahead-of-time compiled just before run-time. Developers compile applications into portable .pexe files, deploy them over the web together with a web page, and run them across browsers that support PNaCl. Additionally, web pages and PNaCl applications can communicate with each other using the PPAPI. This API provides a standardised way for C code to receive messages and data from Javascript, and, in turn, send messages back to Javascript.

Google provides a cross-platform compiler toolchain for compiling C and C++ code into PNaCl binaries. As Google creates both PNaCl and Android, it is unremarkable that the PNaCl toolchain is very similar to the one provided for Android's NDK. As a result, the build for PNaCl Csound is setup similarly to the one used for Android Csound, with a custom Makefile used instead of an `Application.mk` and `Android.mk` file.

Because PNaCl supports pthreads and the toolchain is very similar to ones provided for platforms like Linux, there were few changes necessary to build libcsound and libsndfile. After doing the initial build, Lazzarini then filled in the parts normally implemented within a CsoundObj implementation though, in this case, it is a Javascript and C++ class called Csound.[16] This Csound C++ class uses the Pepper API to register callbacks and handle audio

[16]The PNaCl Csound class could easily be renamed to CsoundObj to provide closer parity to CsoundObj implementations on other platforms, as it functions in the same ways, wrapping portable libcsound API usage with platform-specific code for PPAPI.

communication between a Csound engine instance and the MediaTrack audio system provided by Pepper. The C++ class also handles receiving messages to perform actions such as starting, running, and stopping a Csound engine instance. The Javascript class in turn uses the Csound C++ class via the PPAPI and wraps the functionality of the C++ class, so that end users can write their web applications using Javascript to work with Csound.

Csound PNaCl releases are now released together with every new version of Csound. The SDK follows the norms of other Csound-based SDKs and provides a pre-compiled library, example code, and a manual.

Comparing Emscripten and PNaCl

While both Emscripten and PNaCl versions of Csound provide the same libcsound-based features and are both capable of compiling and running the same Csound code, the two implementations differ in significant ways in terms of performance and availability. In terms of performance, as shown in [108], the Emscripten build does not run as fast as the PNaCl one in terms of raw speed. Additionally, and perhaps more importantly, the WebAudio ScriptProcessorNode's design is not optimal for real-time audio. Consequently, breakups in audio are much more likely to occur with the Emscripten build than in the PNaCl build. For PNaCl, the audio system interaction between Csound and Pepper API's audio system is very similar in design to how Csound interacts with desktop audio systems. As a result, for the best real-time audio performance, the PNaCl version is much preferred to the Emscripten version.

However, in terms of availability, while PNaCl and the Pepper API is a cross-platform technology, it is only implemented by a single browser vendor

– Google – in their Chrome, Chromium, and ChromeOS browsers. On the other hand, since the Emscripten build of Csound only requires Javascript and WebAudio, it is capable of running everywhere that the PNaCl build is able to be run (i.e., the previously mentioned Chrome browsers), as well as all of the other browsers that provide WebAudio (i.e., Firefox, Safari). For users who are less concerned with performance and more concerned with availability, the Emscipten version would be much preferred over the PNaCl version.

With either version of Csound for the Web, the end-user does not require separately installing any applications or plugins prior to loading the web page. The user can just load the page and everything will download and run. From the perspective of dependency management, one can view the Web Csound builds as having all of their library dependencies statically compiled into their builds, such that they will always be satisfied when running the web page. The result is that existing Csound-based projects can be preserved using the Web, together with the exact version of Csound used to create the project. Also, new Csound-based projects will have a high degree of long-term viability to execute and run in the future.

4.5.4 Impact of Csound on New Platforms

Extending Csound to new platforms – iOS, Android, and the Web – has extended both the durability and value of Csound as whole. With more platforms to run Csound, users can have more confidence that their Csound-based projects will continue to operate in the future. Also, with the new features that each platform provides, users can reuse their existing Csound knowledge and experience when they begin new works and more quickly

develop projects on these new platforms. Furthermore, with the introduction of the CsoundObj API, users have clear guidance on how to translate their projects from one platform to the next.

4.6 Case Studies

The following will discuss various case studies that use the iOS, Android, and Web versions of Csound discussed in Section 4.5. The case studies will be divided into two categories depending on their approach to using Csound: *Csound Exposed* and *Csound Inside*. For *Csound Exposed* programs, the program will expose the Csound language to the user of the program. The target user of these programs will be one who knows how to program using Csound's Orchestra and Score languages. An example of a Csound Exposed application on the desktop would be CsoundQT [5], an environment where users program in Csound but have other features provided by the application. The typical users of these kinds of programs will use Csound on their desktop systems and look to use Csound on other platforms, and they may want to move Csound CSD projects between programs and platforms. From a higher-level view, we can say that both the developer and user of the program require knowledge of Csound.

For *Csound Inside* programs, the program does not expose Csound to the end user. An example of this on the desktop would be AVSynthesis [112, 137], a program where the user works entirely with a graphical user interface. For these kinds of programs, knowledge of Csound is only required for the developer of the program, and it is for their benefit.

There are also programs that both expose and hide aspects of Csound, such as the author's own program *Blue* (see Chapter 5). For the use cases

below, the examples will be categorised according to what aspect they most exemplify. Notes will be given describing features that venture away from their category.

4.6.1 Csound Exposed

Csound Notebook

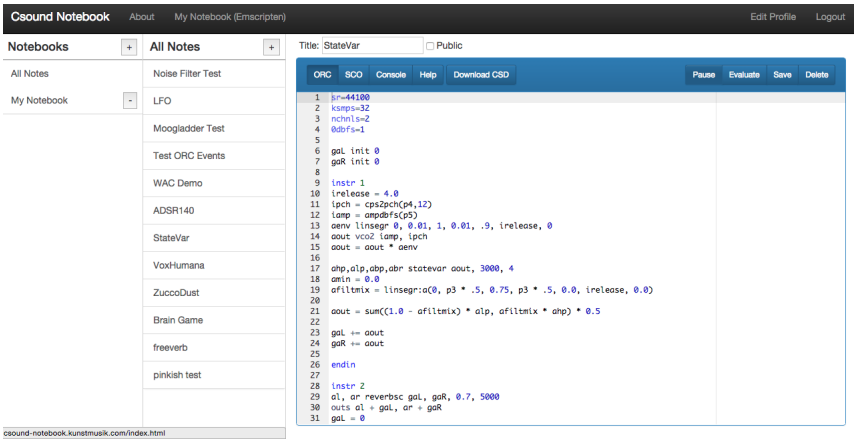

Figure 4.15: Csound Notebook

The Csound Notebook [196] is an open-source [197] website for creating Csound projects, called *Notes*, and organising them into collections, called *Notebooks*. The client-side is written using HTML and Javascript, using the AngularJS 1 [77] framework. The server-side is written using Ruby on Rails [9].

Users using the Csound Notebook must first create an account. Once an account is registered, users can login and open up their notebooks. Figure 4.15 shows the notebook user interface. The interface is organised into three columns. The first column shows the list of notebooks, including a special "All Notes" notebook. Once a note book is selected, the available notes for

that notebook are shown in the second column. Once a note is selected, the contents of the note are shown in the note editor, found in the third column.

The note editor is set up with three primary tabs: Orchestra, Score, and Console. Controls for playing and pausing Csound as well as evaluating code is available above the tabs. Users can write Csound Orchestra and Score code separately, and view the status of Csound output in the console tab. Notes may also be marked as public, which allows the note to be viewed by any user. Otherwise, notes default to private status and can only be viewed or used by the note's owner. Finally, notes may be exported as standard Csound CSD files for use with other Csound versions (e.g., desktop).

The Csound Notebook functions as an online workspace for working with Csound. The application supports use of both Emscripten and PNaCl Csound builds. Users can choose to use PNaCl when using Chrome-based browsers (Chrome, Chromium, Chrome OS) and get near native performance, or use the Emscripten build if they are using any other browser that supports the WebAudio API. Neither of these options require the user to have Csound installed on their system.

The Csound Notebook functions as a useful tool for users to work on musical computing online using Csound. The application also functions as a testing ground for using both Emscripten and PNaCl builds. As the application is open-source, it can serve as a reference for users building their own web-based, music computing systems.

The Csound Notebook exemplifies one particular use case where the end product exposes Csound to the user directly. The users of the Notebook will work with Csound in a classical way, programming in Csound code. Other use cases for Web Csound are described further below.

Csound6 Android Application

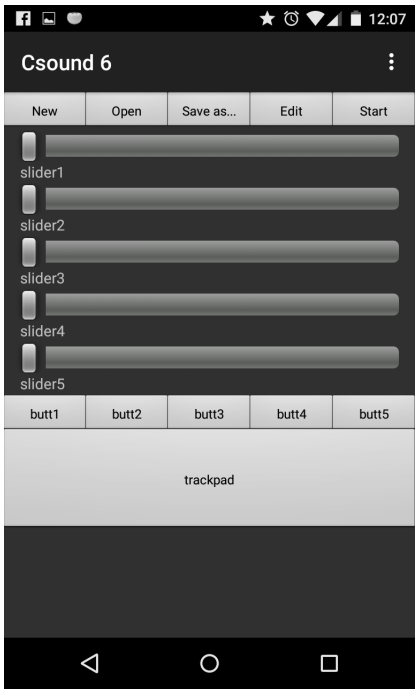

Figure 4.16: Csound6 Android Application

The Csound6 Android Application is a program developed for devices running the Android operating system. Device types where Android is found include phones, music players, tablets, laptops, and desktops. The program was originally named CSDPlayer and was written by Victor Lazzarini.[17] It was released with the initial Csound for Android SDK as an example application for running Csound on Android. The project was further developed by Michael Gogins and renamed as Csound6.

The Csound6 application allows loading and running of Csound CSD projects. A screenshot of the application running on a Nexus 4 device is shown in Figure 4.16. The application does provide an edit button but does

[17]An article describing usage of the earlier CSDPlayer is available at [151]

not provide its own editor. Instead, the application delegates to an external editor application.

The application provides both a standard, fixed set of graphical user interface widgets and user-defined widgets written using HTML5. For the fixed widget interface, each widget is assigned a pre-defined channel that they use to send values to Csound. Users wanting to use the widgets can simply add calls to the *chnget* opcode in their Csound code and receive the values from the widgets. A benefit here is that if the project is moved to another system, the project may still run even in the absence of those widgets (using the default 0.0 value for Csound's control-rate channels). For the HTML5 interface, users can use Javascript to bind the values from widgets to channels in the running Csound instance.

The Csound6 application is an example of a *Csound Exposed* application. Typical workflows include on-device development and execution of Csound CSD projects, as well as off-device development and on-device performance. By extending Csound to the Android platform, Csound users can now extend their own workflow and embrace new performance possibilities using the Csound6 application.

4.6.2 Csound Inside

ProcessingJS

The ProcessingJS Csound example [200] is a demonstration project using PNaCl Csound for its audio engine and ProcessingJS [145] for its user-interface. The project is a client-side project and has no server-side dependencies.

The user interface is a simple interactive canvas. When a user presses down with a mouse, a note string is sent from ProcessingJS to Csound to

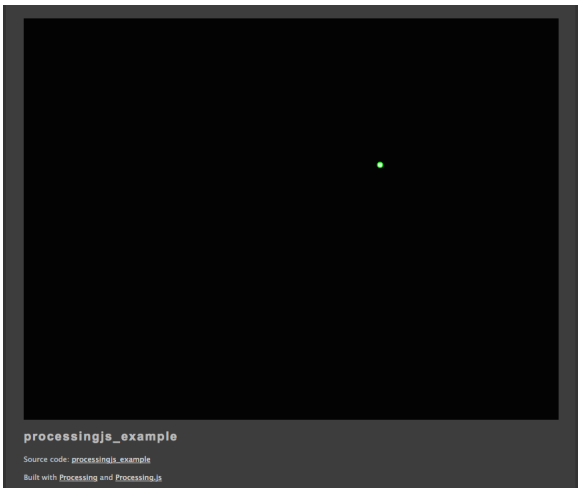

Figure 4.17: ProcessingJS + PNaCl Csound Example

start a note. The x- and y- mouse coordinates are tracked as the user moves the mouse, and the values are mapped to the instrument's frequency and amplitude respectively. Additionally, the ProcessingJS sketch's code will render a green circle that follows the mouse pointer while the user has the mouse button down. On mouse up, the note is ended and the green circle disappears.

This project is an example of one where the end product is a musical application that does not expose Csound to the end user. The use of Csound is purely for the benefit of the developer. Note also that the Csound CSD project is the *exact same* CSD used within the MultiTouchXY examples found in the Csound for Android and Csound for iOS SDKs. This demonstrates one of the larger goals of extending the platform reach of Csound, that of simplified cross-platform application development. One can imagine a developer using the same Csound engine and CSD project as the basis of a musical application across multiple platforms.

AudioKit

AudioKit (versions 1 and 2) [1] were an open-source Objective-C programming framework for OSX and iOS. Both versions were built upon the Csound for iOS and Csound for OSX SDKs.[18] AudioKit provided a high-level library for building musical applications.

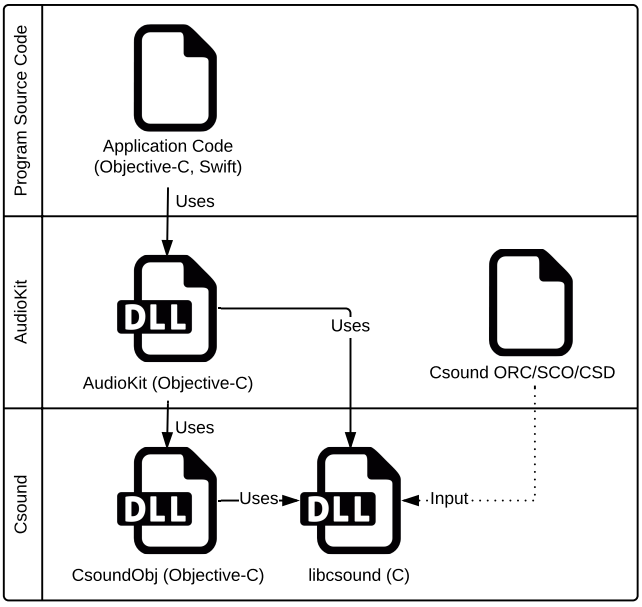

Figure 4.18: AudioKit Architecture Diagram

Figure 4.18 illustrates the relationship between application code, AudioKit, and the Csound for iOS SDK. The target user of AudioKit was an Objective-C or Swift developer looking to integrate sound and music into their applications. No knowledge of Csound was required by users using AudioKit, though Csound knowledge was required to develop and extend AudioKit itself. AudioKit provided its own API to its users and itself used

[18]Csound for OSX is a desktop-focused SDK that contains an implementation of CsoundObj appropriate for OSX. Application code written for CsoundObj in iOS can be re-compiled in OSX using the Csound for OSX SDK, largely without change.

the Csound API to implement the features required by the higher-level API. AudioKit provided some abstractions that mapped closely to Csound abstractions (e.g., `Instruments` and `Operations` mapped closely to Csound's instruments and opcodes) but also provided its own abstractions such as `Phrases` and `Properties`.[19].

AudioKit is an example of a *Csound Inside* program where one wants to build a higher level musical library based upon Csound. Using Csound this way, the developers of AudioKit build upon a foundation of well-tested audio engine code that they do not have to implement themselves. The depth of AudioKit demonstrates the possibilities of what can be developed using CsoundObj and Csound.

csGrain

csGrain [36] is a commercial iOS application produced by Boulanger Labs. It was the first iOS application released on the App Store to use the Csound for iOS SDK. The application uses Csound internally for its audio engine, while its own application code focuses on areas such as the graphical user interface, configuration storage, and inter-app audio communication. The manual for csGrain describes the application's capacities as follows:

> csGrain is a stereo granular sound processor with 10 post-processing effects – all realized through a single Csound orchestra that is rendering, processing, sampling, resampling, synthesizing, resynthesising, playing, reversing, delaying, triggering, gating, compressing, limiting, chorusing, flanging, echoing, filtering, pitch-shifting,

[19]Further information about AudioKit's abstractions can be found in the documentation provided with AudioKit releases.

> harmonizing, granulizing, and recording – in any combination, or simultaneously; all in real-time, and all 100% being done in Csound! csGrain is the first of Boulanger Lab's innovative and cutting edge Csound Touch apps that are built with the latest version of the Csound for iOS SDK by Victor Lazzarini and Steven Yi. [37]

csGrain is an example of a *Csound Inside* application that employs Csound on a mobile platform. The application takes advantage of native coding practices and blends in with other applications on the platform. It employs hardware specific to iOS devices, uses operating system specific features, and uses the Csound engine with cross-platform Csound Orchestra code. csGrain demonstrates how CsoundObj and Csound can be used to make applications that are well-integrated into mobile platforms and do not require users to know Csound.

4.7 Conclusions

Developing programs for platform-extensibility requires understanding what are all of its dependencies. From there, the process of porting software is one of trying to satisfy all dependencies on each target platform. This may reveal new dependencies that were the result of assumptions becoming invalid.

The work in porting Csound to iOS, Android, and the Web has shown that even software that has long been cross-platform can still present new dependencies. Issues such as temporary file writing on Android and the absence of plugin loading on iOS and the Web were challenges that required

new software designs, both internally (e.g., replacing temporary score file writing) as well as externally (e.g., CsoundObj).

However, by employing build-time configuration and compile-time conditional code, as well introducing the CsoundObj API, Csound was successfully ported to all three platforms. Users can now employ their Csound knowledge in new places, take advantage of the unique features each platform offers, and rest assured that Csound can continue to grow and adapt to the changes in computing.

Looking towards the future, the Csound code base should continue to support new platforms as they arise and attempt to do so in ways that embrace the unique qualities of each platform. It should continue to use the same source code on all platforms so that new developments are shared and projects built for one version of Csound can expect to run equally as well on another. By developing for platform extensibility, Csound continues to support users' existing work and offer new ways to leverage their existing knowledge and experiences.

Chapter 5

Modular Software Design and Blue

This chapter will discuss modular software design and run-time module-based systems as a foundation for extensible computer music software. It will begin with a look at different music system archetypes and discuss their features and drawbacks for users and developers. Next, it will look at modular systems and how they leverage features found in the two prior types of system. Finally, original work for this thesis will show applications of modular design in Blue's [195] Modular Score timeline. This provides a developer-extensible way to add new score layer groups and types to Blue.

5.1 Introduction

Computer music software is designed for various modes of operation. Some software is independent and designed to stand alone, whereas other types of software are independent but designed to interoperate with other software,

and still other types are completely dependent on other software to function. Users will pick and choose from the software that is available to them and use it to create their music. Developers will work to extend the systems by the means available, whether that is directly extending a software or indirectly through creating plugins or separate applications. The means by which a software is made extensible for developers directly impacts the way users will assemble software for their musical work.

Taking a step back and looking at the larger picture of a musical project, one can see a network of dependencies develop based on all of the software that a user employs to create a work.[1] From the start of a musical project, each software introduced becomes an extension to the state of the system. Understanding how the total system assembled by a user for their work is organised and how the pieces communicate and interoperate with one another can help to understand the robustness of the work over time. The ways that software is made extensible thus not only affects what operations are possible with the system but also how fragile the work may be.

The following will begin by analysing various music system designs and discussing their strengths and weaknesses. Next, it will consider the issue of extensibility in existing music software systems. Finally, the chapter will present the original work to develop Blue's Modular Score timeline. This presentation will include a review of timelines in other software, the design of the modular timeline, and case studies of new layer group types implemented for Blue.

[1]For a further discussion on dependency analysis, see Section 4.2.3.

5.2 Music System Designs

Music programs can be systems unto themselves or they can be a part of a larger set of programs that together make up the system used for a work. In this section, I will discuss the primary building blocks of music systems: executables and plugins. From there, I will look at single and multi-executable systems. Finally, I will look at modular systems and how they relate to and differ from other systems.

5.2.1 Executables and Plugins

Executables

Executables are standalone applications that users use to perform some set of operations. They may be non-interactive programs that operate on given inputs and return an output. They may also be long-running processes that are interactively operated by a user. They may have graphical user interfaces or be designed to operate in a terminal. Executables form the starting point of computer music systems.

Executables may or may not offer extensibility through plugins. If they do, they will first look at their *registry* of plugins to discover what plugins are available. The registry may be *explicitly* defined, such as having a text file that lists what plugins to load, or it may be *implicitly* defined, such as having a directory where any files found that follow a naming convention may be assumed to be a plugin for the system. Once the registry is consulted, plugins will be loaded and the life cycle of the plugin will begin. Throughout the life cycle of the plugin, the host may search for values or call functions provided by the plugin, and the plugin may in turn do the same for values

and functions provided by the host. The host and plugin will communicate via the plugin/host Application Programming Interface (API).

While an executable may offer a plugin point as a means to extend the system, executables may also be designed to interoperate with other executables. Executables may communicate explicitly with each other using some form of interprocess communication (i.e., sending and receiving binary or text data via pipes or network sockets). They may also communicate with each other implicitly using an intermediary data file in a known format (i.e., one application writes a MIDI file to disk, a separate application reads the MIDI file and renders it).

Plugins

Plugins are extensions to a system that require a host application to use. Plugins do not stand-alone and can not be executed directly by a user. Instead, an executable application loads plugins to offer additional features provided by the plugin.

Plugins are often packaged in the form of *dynamically-loaded* shared libraries. Unlike *dynamically-linked* shared libraries, these are unknown to an application until run-time. Programs use system-provided functions to explicitly load libraries and search them for symbols to use as data or functions. These are used to install one or more plugins and extend the program's capabilities.[2]

While there are many different types of plugins, plugins as a whole do share some general properties. Plugins must adhere to a *convention* or *format*. Plugins must also be *registered* in some way with the host application so that

[2]For further discussion on dynamic linking and loading, see [113, Chapter 10. Dynamic Linking and Loading].

the host can *discover* the plugin and know how to load that plugin. Finally, plugins also have a *life cycle*, where the plugins are used or perform certain actions at certain times, determined by the code using the plugin.

For example, the music program Pure Data (PD) works with a plugin format it calls *externals*. As the documentation for PD notes, it loads plugins on demand:

> Pd looks first in the directory containing the patch, then in directories in its "path." Pd will then add whatever object is defined there to its "class list," which is the set of all Pd classes you can use. [146, chapter 4: writing PD objects in C]

This convention serves as the means by which PD *discovers* plugins, with the registry of available plugins being implicitly defined by the available libraries found in the folder for the patch or the program-wide path-list for externals. To load an external, the library must have a public `void xxx_setup(void)` function that can be found to initialise the plugin. The xxx part of the name must match the name of the plugin itself. [206, 2.4 generation of a new class] The setup function for the plugin also serves as the initial point of entry into the external and is part of the *life cycle* of the plugin. Finally, the plugin uses the API defined in `m_pd.h` to *interact* with the PD system and register new object classes.

Plugins offer a means by which to extend the functionality of a software. Once loaded, their features become a part of the running application. Extension by plugins may be offered not only by executables, but also by libraries and other plugins. The scope for where a plugin can be used is determined by what host applications support the format used by that plugin.

Analysis

Executables and plugins form the base upon which computer music systems are developed. Users may use one or many executables and each executable may have zero to many plugin possibilities. Plugins in turn may also support plugins themselves. The following will discuss application archetypes for music systems based on their usage of executables and plugin.

5.2.2 Single-Executable Systems

In single-executable systems, a user uses a single top-level executable as the primary system for their work. For example, this could be a command-line program like Csound, or a program with a graphical interface such as Ableton Live [13]. These programs may use projects and resource files (e.g., audio files, data files) to perform or render the musical work, or else operate as a musical instrument to be used in real-time.

Single-executable systems may be entirely self-contained or allow for extension through plugins. For example, a program like Xenakis's UPIC [118] was a self-contained, graphical application that was not extensible by third-party developers[3]. The application was used to create original content and to render the final audio output.

On the other hand, a sequencer program, such as Apple's Logic Pro [94], does allow users to augment the available instruments and effects within the program by installing additional third-party plugins. The application

[3]The Lohner article describes the system in 1986; I had an opportunity to use UPIC in 1999 during the summer course at Les Ateliers UPIC and did not find any means to extend the system at that time as well.

provides a host environment for creating musical work but participates in a larger ecosystem of music software through the AudioUnit plugin format.

From the point of view of the developer, a single-executable system represents a complete vision for a music system. That vision of software may be the work of an individual developer or the shared work of many developers. Developers can extend a single-executable system either by modifying the source code of the executable or by creating plugins for known plugin points. Extending the executable requires tight coordination amongst developers and consensus on acceptance of changes, if the extension is to become a part of the canonical executable application. Extension by plugin requires loose coordination between the core application's developers and plugin developers via the specification of the plugin format. However, given a plugin point and its format, any developer can extend that point as they wish without coordination with the core application's developers.

Analysis

Single-executable systems are the smallest operational unit for computer music making. The executable may be singular in purpose or be very deep in features to accommodate many musical tasks. When plugin support is available, third-party developers can extend these systems by adhering to the specification and format of published plugin points.

However, when a feature is desired that would extend the executable in ways not possible by plugins, or if plugin support is not available at all, third-party developers have limited options. If the system is open-source, they can work with primary developers and contribute code to the original project, or otherwise create a fork of the application with their own changes

applied. The former involves a risk of the change not being accepted; the latter requires upkeep if the fork is to remain current with the upstream source.

If neither of those options work for a developer, or if a system is closed-source, then third-party developers may be able to create a separate executable application that can communicate with the primary application. This requires that there be a communication protocol in place that both applications support. If a protocol is not present then there may be no way to extend the primary application for the desired feature.

5.2.3 Multi-Executable Systems

Multi-executable systems are built upon single-executable systems. With multi-executable systems, users assemble the total system for their work using available single-executable systems. Each application within the work may or may not communicate with another application (i.e., a user runs the two programs simultaneously but the programs are unaware of each other). When there is communication between applications, it may be *direct* or *indirect*.

Figure 5.1 shows a diagram of a simple two-application multi-executable system. One application supports plugins and the other does not. The arrows represent the communication between these programs.

Applications that run concurrently may directly communicate with each other in real-time. This communication can take the form of some inter-process communication system and format, such as using pipes or sockets to send formatted binary or textual data. Applications may also communicate with each other indirectly. Here, one program may generate data that is stored on disk, then another program later opens and reads that data. The

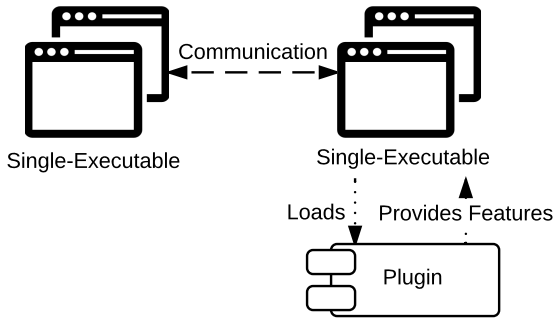

Figure 5.1: Multi-executable system

data format may be the same as what is used in direct communication but how the information is communicated differs. Indirect communication also allows for the data to be distributed separately from the application that generated it and for the information to be processed at a later time than when it was created.

Developers may design music systems from the outset to be made from many executables. A primary example of this is the CARL [119] system, made up of many small executables designed to work together through Unix Pipes. (Further analysis of CARL is given in Section 5.3.2). In this case, the parts of the core system are distributed together and third-party developers can extend the system by creating new executables that follow the communication protocols set out by the system.

Another example is SuperCollider 3 (SC3) [123] as a dual-executable system. As discussed in Section 4.3.1, SC3 employs two programs: the scsynth server, run in one process, and the sclang language interpreter, run in another process. They communicate with each other using the OSC format over network sockets (TCP or UDP).

Users may also create their own multi-executable systems by assembling various other systems together. The parts in these ad-hoc systems have

means to communicate and work with other systems, though they may not be primarily designed to do so. For example, a user might use one music system to compose and render music out to sets of audio files. From there the audio files are then loaded into a second program for mastering work to generate the final audio product. In this scenario, two separate programs were used to create the final output and the means by which they communicated with each other was through audio files.

Analysis

For third-party developers, multi-executable systems provide a high degree of flexibility to extend an existing system. Given that an existing system adheres to a known protocol, developers can use whatever technologies they wish to create a new executable to work with that protocol. For example, if Program A is written in C++ and is able to render AIFF audio files to disk, a third-party developer could write a Program B in any language and GUI toolkit to process AIFF audio files. This would allow a user to use both Program A and B together to create a work, using features from both programs.

Multi-executable systems can empower both developers and users. Developers benefit by being able to create new executables using technologies that may be very different from those used by other executables in the system. Users benefit by choosing from a variety of software options to assemble a system that uniquely satisfies their needs in ways that single programs may not alone.

However, while multi-executable systems provide a lot of freedom, they are not without drawbacks. Firstly, the dependencies for a work are made up

of each application used by the work as well as any plugins used by the user within each application. Each dependency introduces greater complexity in satisfying the overall dependencies to recreate that work. As each executable can have its own unique network of library dependencies, the complexity of the overall dependency graph can be come quite large. With single-executable applications with plugins, the network of dependencies is resolved within the confines of a single system and features can be shared between the host and plugins. The tendency within single-executable systems then is to have smaller dependency networks – a benefit to users.

Using multi-executable systems can also obscure the dependencies of applications upon each other. They may also make problems with changes in communication protocols (i.e., the data format) harder to detect. For example, a user uses Program A_0 and B and they use file format X_0 to communicate with each other. Later, Program A_0 is modified as A_1 to produce data in format X_1, and a user uses Program A_1 with Program C in a newer work, which is also designed to use format X_1. All is well until the user returns to the first project after some time and attempts to use Program A_1 with Program B again. Program B however was not updated to use format X_1 and only now is the problem discovered. This error may be a problem that the developer of Program B might not be aware of and that a user may not be able to diagnose easily.

If, instead of individual executables, program A was a single-executable program and programs B and C were plugins, then modifying the data format or communication protocol could have caused a compilation error or plugin load error to be reported for program B. In this case, the user may very well get an error message from program A that program B is no longer compatible

with program A and the developer of program B may learn there is an issue with their program.

In addition to the problems of dependencies and communication formats is the problem of communication graphs. Loy discusses this point in [119], indicating that the communication graphs for the CARL system could only reach certain levels of complexity. One-to-one communication for applications would be fine but one-to-many and many-to-one relationships were not well supported. While this may not be a problem in systems based on communication servers like JACK (discussed further in Section 5.3.3), the issue is one that can present itself when other communication protocols – such as MIDI or OSC – are used.

Furthermore, for real-time multi-executable systems, another problem is *session management*. A session involves the state of all applications used in a work. Session management then is the management of the state of applications and involves saving and loading of the state of all programs participating in the session. This not only requires knowing the state of each application but also restoring the state of each program in the correct order. For example, if software A routes audio to software B in real-time, reproducing the session state requires that software B load first, then software A, then the connection made between A and B. Session management may be an ad hoc process managed by the user, or may be automated by a session management system, which tracks dependency graphs between applications.[4]

From a high-level perspective, multi-executable systems empower users to use the best applications for each part of their work and empowers developers to extend systems with tools of their own design using whatever technologies

[4]Session management has been an ongoing issue in the Linux Audio community, with multiple approaches having been explored. See [136] for further information.

that best suit them. However, the benefits must be weighed heavily against the potential fragility of the work due to the larger number of dependencies, the complexity of communication between applications, and the difficulties in recreating the state of the total system.

5.2.4 Module-based Systems

Module-based systems, as defined here, are an extension of single-executable systems design where the application is made up almost entirely of plugins loaded at runtime. With typical single-executable systems, many if not all of the features of the application are built into the executable itself. However, with module-based systems the executable contains very few features, primarily dealing with discovering, ordering, loading, and unloading of plugins. In turn, all application-specific code is delegated to functionality provided in plugins.

Module systems – such as OSGi-compliant containers [85] and the Netbeans Module system [33, Chapter 3: The Netbeans Module System] – use a very generic module format. Modules have callbacks that are called at different points of the application's life cycle. These include things such as when a module is first discovered and installed, when it is loaded, when it is unloaded, and when it is uninstalled. Modules also list dependencies upon other modules as well as public classes available for other modules to depend upon. In other words, modules are used both at compile-time, like a library, and at run-time. Module containers have not only the responsibility to load specified plugins but also to do so in the correct order using the dependency graph of the modules.

At run-time, module systems provide a means for modules to advertise service or plugin class implementations as well as to search for and discover those services and plugins. This mechanism is a primary feature of module systems and it promotes application architectures with many points of extensibility for third-party developers to customise. It also helps core developers of an application to use the same plugin mechanism internally as it promotes clarity in design.

Unlike typical single-executable systems, applications using the same module system all share the same top-level executable. The only difference between applications are what modules are provided and the configuration information. Therefore different applications may use the same executable and module system but produce very different systems. Additionally, users can install modules from different developers in the same container, effectively having multiple top-level "applications" in the same single-executable application.

Analysis

Module-based systems offer features to the application developer that bring together some of the best aspects of both single-executable and multi-executable systems. However, module-based systems also have their own unique drawbacks and concerns.

Single-executable system designs can be implemented using modular systems. For self-contained single-executable systems, a comparable design would be to create a single-module program that contains the entire application codebase. For single-executable systems with plugins, a core module or set of modules can be created that maps to the features of the single executable.

The core module or set of modules may expose interface classes for other modules to implement as plugins. From there, third-party developers could create modules that depend on the plugin interfaces within the core module. They would package their plugin implementations within their own modules. The plugin modules could then be presented for users to install into their module container application and the plugins would become available for the user to use.

Module-based systems can also operate similarly to multi-executable systems. For example, given a known communication protocol, two developers can develop their own set of modules as applications and the two can be loaded into the same module container. The two "applications" could then work directly or indirectly with each other, with the same conditions found in multi-executable systems. The separate set of developers can work independently, yet participate within the same application container. While module-based systems can provide this kind of architecture, it still has the same kinds of drawbacks as multi-executable systems and is probably not the most effective use of modular-programming technologies.

The real advantage of module-based systems is that they make designing plugin points and using plugins into a fundamental part of application architecture design. The result is that application developers using module-based systems tend to make many more features of their program extensible as plugins than non-module-based systems. As a result, there are more opportunities for third-party developers to extend a system without having to create their own separate executable.

This too may help users who might not find all features within an application adequate. Instead of reaching for another executable application and thus

introducing another potentially large set of dependencies, they may be able to find a module that can address their needs. If the module container can install a module into its container then the module system has verified and properly satisfied the dependencies of that module. Consequently, the burden of verifying the total system's dependencies is managed by the application itself, rather than by the user.

However, module-based systems do have their drawbacks. Firstly, if an application is small in scope, the features of using a module system may not be worth using. Secondly, modular programming frameworks are mostly found in use on the Java Virtual Machine (JVM). This limits the programming languages for developers to those that work on the JVM. Although there are many languages that work on the JVM – Java, JRuby, Scheme, Clojure, Jython, Scala, and Groovy to name a few – it may not satisfy those developers wanting to work in a language that operates closer to the machine level (i.e., C, C++, Objective-C). Modular programming systems in other languages do exist, such as Celix [22] – written in C – and CTK [6] – written in C++. However, they do not appear to be as widely used as Java modular programming frameworks. Finally, module-based applications may require the use of specific GUI toolkits. If a developer does not wish to use that GUI toolkit, they may decide not to create plugins for that application.[5]

Module-based systems provide a great deal of flexibility in designing applications and make extensibility a high-level concern when designing the architecture of a program. It extends the monolithic single-executable with plugins model by simplifying and generalising plugin loading. This in turn

[5]The primary Java runtime has evolved to provide three different UI Toolkits: AWT, Swing, and JavaFX. There are opportunities to use all three within an application. However, a separate UI toolkit, such as SWT, may not interoperate well with the built-in UI toolkits.

provides more opportunities for extension by third-parties, providing some of the same freedoms afforded by multi-executable systems. While multi-executable systems will still develop – whether designed by developers or assembled ad hoc by users – module-based systems can provide a base where the need to create a separate executable is diminished. In the end, the musical software ecosystems that can arise around a module-based system can help manage a work's dependencies for users, yet also continue to grow to meet out users' needs in the future.

5.2.5 Summary

Music systems are built from executables and plugins. A system may be as small as a single executable or grow larger through plugins and other executables. Executables and plugins all have dependencies and all of the dependencies become a part of the total system's dependency graph. This in turn affects the robustness of a user's work.

Module-based, single-executable systems provide an alternative to single-executable and multi-executable systems. It provides a foundation to build music programs open to extension at many levels, yet carefully managed in terms of dependencies. This allows an ecosystem to develop around a music application that can serve the needs of users and developers but also minimises the risks for long-term viability of projects and works.

5.3 Computer Music Systems and Extensibility

In this section, I will look at a number of existing computer music systems, examining their overall designs and comparing them to the application

archetypes presented in the previous section. I will also frame the systems in terms of how extensibility is accounted for and the impact that system's extensibility design has on the developer and user.

5.3.1 Digital Audio Workstations

Digital Audio Workstations (DAWs) are a common type of graphical music application that allow users to work with visual timelines to organise and perform musical material. Originally the term DAW applied to programs that strictly worked with digital audio content and the term *sequencers* applied to programs that worked with organising MIDI content. Today, most programs that supported digital audio now support organising MIDI material and vice versa, and the term DAW can be used to apply generally to these class of programs. Examples of DAWs include open-source programs such as Ardour [55] and QTractor [42] and commercial closed-source programs such as Steinberg's Cubase [74], Apple's Logic Pro X [94], and Cakewalk's Sonar [95].

DAWs are generally single-executable, monolithic applications that support plugin formats such as AudioUnit, VST, LV2, and LADSPA. Plugins are available to extend the system at known points – instruments, effects, and MIDI processors – but extending the primary host application requires introducing changes to the primary source code. This must be done by a trusted developer with access to the code for the application.

There are a few options available to the third-party interested in extending a DAW application. One could make a request for a change to be performed by one of the core developers. Another option available for open-source programs would be to perform the change to a local version of the application's source

code and submit a patch to the core developers, whom would inspect and either apply the change to the main source or reject it. In single-executable systems like DAWs, modifications to the primary application are generally closely audited. Changes deemed not globally useful may not make it into the canonical application. This is good for consistency of user experience but prevents integration of novel features that may be extremely beneficial for some users, but not for all.

For open-source programs, one alternative is that a third-party developer or set of developers could create a custom version of the application and distribute the changed source and/or the modified version of the application. The differences between the primary and modified sources could then be maintained as patches. These patches could then be applied whenever there are changes to the primary code base. Also, the code may become a completely forked version of the code, with changes from the original repository applied to the fork so that the fork can receive new features and bug fixes from the original. However, this may require a lot of work, especially if the original or fork diverges enough where patches cannot be applied easily and manual intervention is required.

Another approach to dealing with extensibility by third parties is to provide more plugin formats by which to extend the primary system. If the format is stable, third party developers could create plugins to add new menu options, add actions, add behavior, and customise the UI. The choice then to use the features provided by these plugins becomes one a user can opt to use or not. The most useful features would still likely be made as part of the primary system but less globally valuable features still have a way to be a part of the application.

Also, the means to extend the system need not necessarily be by binary plugin; it may also be some form of scripting language. For example, Cakewalk's Sonar provides a custom scripting language called CAL (Cakewalk Application Language); Cockos's Reaper [96] provides ReaScript [43], which allows programming using EEL2 (a custom open-source language), Lua, or Python programming languages. With both of these systems, users can extend the behavior of the primary application by writing custom scripts. This provides at least a limited way by which novel and useful features could be introduced by third-parties to the existing system.

User may use a DAW alone but they may also use one as a part of a multi-executable system. Using DAWs this way increases the complexity of the total system but the complexity may be well understood and accepted by a user. An analysis of JACK-based multi-executable systems, discussed below in Section 5.3.3, will explore these complexities further.

DAWs generally provide a consistent and stable user experience by implementing many features within their application code. They support limited extensibility through a fixed set of plugin formats for features such as instruments, effects, and MIDI processing. They may also offer support for scripting as a way for users to extend the system in novel ways, but this support is in itself novel for DAWs and not a common trait for these kinds of programs. Users working with DAWs may have all of their needs met by the features provided by the program; however, when the user requires a novel feature, DAWs can be more limited than other kinds of programs in ways to extend the application.

5.3.2 CARL

The CARL System [119] was a multi-executable system designed for Unix operating systems. Each executable within CARL was used to perform a single operation and users programmed their work by connecting executables together and streaming data from one application to another. Users could use the programs by themselves, execute a series of applications together on a command-line, or use shell scripts to program a work or batch process using the CARL applications.

CARL's design followed classic UNIX programming philosophy[6], providing multiple single-purpose executables that followed a well-defined interface: this design accounts for extensibility from the start. CARL provided a number of features that would be generally useful that generically worked with binary data over pipes. Users could combine the provided applications as they wished. If a feature was desirable but not provided by CARL, a third-party could develop a new executable in any language available and have it work with any of the other CARL executables, provided it used the same communication protocols as the rest of CARL.

CARL's design worked well for its initial purposes, but it would have drawbacks over time. As Loy points out in [119], some of the primary issues with CARL's design have to do with the focus on "linear processing chains" and "avoidance of anything resembling dynamic real-time operation". This could be stated more generally that the system of communication used between the executables within CARL was not capable enough to satisfy the requirements of computer musicians over time.

[6]For more information about UNIX programming, see Raymond's *The Art of UNIX Programming* [150].

The design of CARL provides a great deal of extensibility and flexibility for users. However, the system of communication between executables ultimately limited the capabilities of the total system. It may be that a different system of communication or different granularity of features per executable might have solved some of these issues but the system did not explore those options and eventually fell out of use.

5.3.3 JACK-based Systems

Another multi-executable system are those developed to work with the JACK audio server. [56] The JACK audio server was originally designed as a way to connect audio programs to hardware. Later, the JACK audio server became a hub by which audio from different programs could be routed to each other and real-time multi-executable systems could develop. Developers could program their JACK-compatible applications using their preferred languages and GUI toolkits and each application could participate within a larger network of applications.

JACK-based systems provide many similarities to CARL and offer the same freedoms to arbitrarily extend the system with third-party applications. The system of communication – the JACK audio server – allows for much more flexible routing and real-time communication than Unix pipes. This addresses some of the problems mentioned by Loy regarding CARL.

JACK has also gone on to inspire other inter-application audio systems. On the iOS platform, Audiobus [27] has become a popular audio routing and session saving system for connecting audio applications. Apple later added their own inter-app audio system in iOS 7 as part of their Core Audio library [24, Working with Inter-App Audio]. On Android, the Patchfield [79]

system was developed to achieve the same kinds of capabilities as JACK for inter-app audio. From a high-level perspective, all of these systems – including JACK – share traits as communication technologies for building multi-executable systems.

However, while many freedoms for developers and users are granted with such a system, as mentioned earlier, multi-executable systems can be fragile. Compared to a work created using single-executable applications, a work that depends on multiple executables can have a much larger dependency graph that must be satisfied if the work is to be recreated. Also, session management becomes a concern with JACK-based systems. The freedoms and capabilities must be weighed against the risks to the degradation of the system over time and portability of a work.

5.3.4 Summary

The designs of DAWs, CARL, and JACK-based systems present different architectures based on the number of executables and system of plugins. Ideally, a music system would provide the stability and consistency that Digital Audio Workstations provide as single-executable applications. Also, a system would ideally have the flexibility to extend the system as one has in systems like CARL and those built around inter-application audio systems like JACK.

5.4 Blue: Modular Score Timeline

This section will look at original work for this thesis for Blue's Modular Score timeline. It will begin with an introduction to Blue. Next, it will

review existing score timelines and discuss their properties. Motivations for developing a developer-extensible, modular score timeline will follow. Afterwards, the design and implementation of the new timeline will be explored. Finally, two new Score Layer groups and types will be presented.

5.4.1 Introduction to Blue

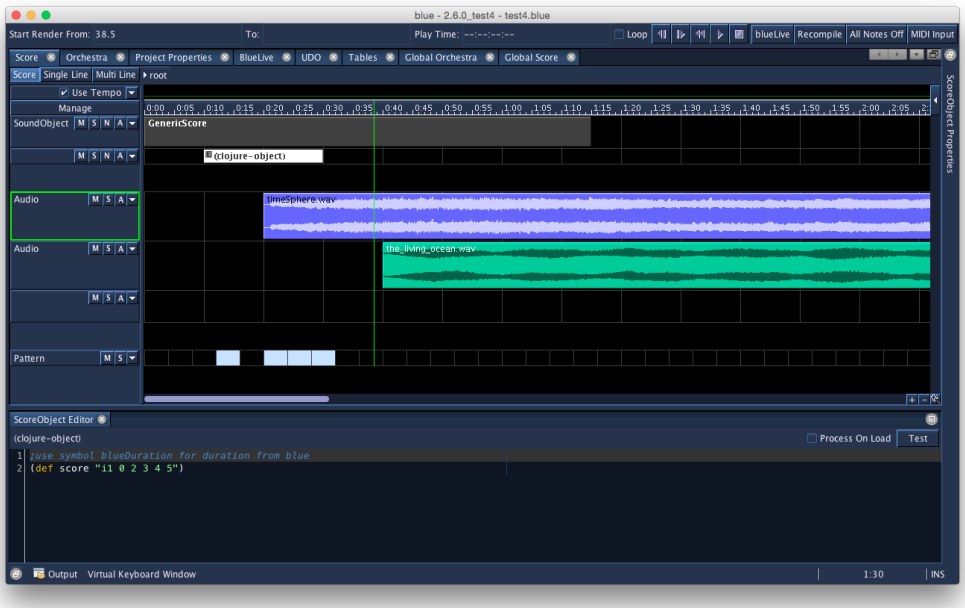

Figure 5.2: Blue: Modular Score Timeline

Blue [194] is an integrated music environment for composing and performing computer music. It is written in Java and is built upon the Netbeans Rich Client Platform (RCP) [47]. Blue also uses Csound as its audio engine, either indirectly through executing the Csound command-line executable or directly through Csound's Java API.[7]

[7]This makes it either a single-executable or multi-executable system, depending on how the user decides to operate the program.

Blue is both a "Csound Inside" and "Csound Exposed" application.[8] It features a score timeline, graphical instruments, effects, and mixer system, as well as a number of high-level features. These parts act as a framework in which to use Csound programming to develop works. The program scales such that the user can use mostly Csound code for their work, or use mostly visual tools and almost no code at all.

In terms of classification, Blue is primarily a single-executable, module-based software program. Using the Netbeans RCP, it provides many opportunities for developer extension. Internally, Blue is developed as a set of modules and the application is released together with other modules provided by the RCP. As is the practice with RCP applications, many parts of a program are developed either to implement a plugin point or to consume plugins. As external modules can be installed into an application, developers have a well-defined mechanism provided by the RCP to create new modules for Blue to extend its features and introduce new ones.

5.4.2 Review of Score Timelines

Score timelines present users with a visual way to coordinate musical material in time. Time can be measured in a number of ways – beats, seconds, measures, SMPTE time code, etc. – and different systems may show time in one or more ways. Timelines are often divided into *layers* or *tracks*, and each layer may be of a different type. For each layer type, there are different kinds of objects that can exist; for example, audio layers may contain audio clip objects, and MIDI layers may contain MIDI clip objects.

[8] Section 4.6.

The following will look at score timelines in various music software. The software will be classified according to whether they are homogeneous or heterogeneous in terms of both their layer types and layer object types.

Ardour

Ardour is an open-source Digital Audio Workstation (DAW) and sequencer. Prior to 3.0, Ardour's timeline contained one layer type: audio layers. Audio layers provided one type of layer object called *Regions*, which map to portions of an audio file. Multiple regions may map to the same audio file and each region in turn may have individually unique properties. For example, if two regions map to the same audio file and use the same start time and duration of audio to play, they may differ in their fade-in and fade-out times.

From 3.0, MIDI layers were introduced into Ardour. MIDI layers allow MIDI clips to be organised on the timeline. MIDI clips are used to drive instruments – either using software synthesisers or hardware devices.

The Score timeline in Ardour prior to 3.0 could be classified as having homogeneous layers of type Audio with homogeneous objects of type Region. From 3.0 onwards, Ardour could be classified as having heterogeneous layers of type Audio or MIDI with each layer type as being homogeneous in terms of allowed sub-objects – Regions and MIDI clips respectively. Through the current version, Ardour's layers and sub-objects are not extensible by third-party developers.[9]

Ardour's score timeline is characteristic of most commercial and open-source DAWs and sequencers. Heterogeneous layer types with homogeneous

[9]The current version of Ardour is 4.1 as of the time of this writing.

objects are also found in QTractor [42], Steinberg's Cubase [74], Apple's Logic Pro X [94], and Cakewalk's Sonar [95].

While Audio and MIDI layer types cover a large set of use cases, developing new layer types or layer objects must be done within the core software application code. This puts a burden of implementation and support on the core developers. If these systems allowed for plugin layer types, third parties could freely extend the system without intervention by core developers and the burden of support would move to outside the core development team. Also, as noted in Section 5.3.1, the introduction of novel features that may have a limited audience is hindered when options for third-party extensibility are not available.

Kyma

Kyma [155] is a hardware/software music and audio system produced by Symbolic Sound. Kyma's timeline contains a single layer type that contains a single, but user-extensible, object type. From the product page for Kyma X:

> Layer and sequence your sounds by dragging them into the timeline. Each bar in the timeline represents a synthesis or processing algorithm – program running on the Pacarana, starting at a particular time, perhaps running in parallel with other programs, and stopping at a specified time. You could think of the timeline as a "process scheduler". For example, you could create a timeline where each bar represented a different effects-processing algorithm applied to the microphone input – with each effect starting at a different time, some of them running in parallel and routed to

different outputs, and some of them fading out before others. [168, Timeline]

Kyma's homogeneous layer type provides a single, consistent way to organise material in time. Since Kyma's layer objects are user-extensible, users can create objects with different processing algorithms and user interfaces. This allows one to not only use pre-made objects, but also modify existing ones and create new ones from scratch. Thus the system provides many ways to express musical ideas within the contexts of bars on a timeline.

While a system like Kyma is very powerful, it does not allow for different representations of and interactions with music on the timeline itself. Instead, variations in musical ideas are expressed within the editor for each bar in a separate editing area outside of the timeline. Users have many options for individual object variety but, within the context of the timeline, they are limited to the visualisations and interactions provided by the software's developers. Like Ardour, any new layer types or object types would require development and support by the core developers.

Duration

Duration, developed by James George and YCAMInterlab, is an open-source project that describes itself as a "Timeline for creative code." [73] The program is a score timeline made up of tracks. Each track can be one of a number of pre-made types: bangs, flags, switches, curves, LFOs, colours, and audio. Each track type has its own user-interface and each type generates OSC data in its own unique way. The application is designed to be used with a separate receiving program that can communicate over OSC. Thus, it is designed for

developing multi-executable systems and works and has all the benefits and drawbacks associated with that class of system design.

As the README.md file notes:

> Timelines are used in so many different scenarios there is no way that one application could solve them all, with this in mind Duration was built to be extended. [72, Hacking on Duration]

Duration can be classified as having heterogeneous layer types and either none or one object type per layer (in other words, some layers do have sub-objects while others do not). While the code base supports extending the system by adding new layer types, it does not appear to expose this capability to third-party developers via plugins. The source code does not have plugin loading facilities nor are there any mention of such things in its documentation. It is assumed then that the path to introducing new layers is to modify the source code for Duration and request merging if one wants to extend the system for all users.

Duration has the internal architecture to allow arbitrary extension of its timeline at the layer level. It would certainly be possible to expose these capabilities to third-parties with the addition of a plugin system. However, the state of the program is uncertain, as its last release was February 26th, 2013 (Alpha 004), and there are very few commits since then [71]. It may well be that the program is "complete" and satisfies the goals of its creators, or it may be that it is no longer being maintained or used. Regardless, the system is open-source and open to extension to support plugins if its community of users and developers wants to pursue that path.

Analysis

Ardour, Kyma, and Duration present various approaches to score timelines. They differ in their types of layers and in the types of objects each layer supports. They also represent a range in terms of their designs for extensibility of their timelines. Kyma's user-extensible objects present extensibility at the level of the layer object; Duration's design presents extensibility at the layer level; and Ardour presents a fixed layer and object design. None of these programs provide extensibility of their timelines by third-party developers.

5.4.3 Motivations

Prior to the work in this thesis, Blue's score timeline offered a single layer type – called Sound Layers – that contained heterogeneous musical material – called SoundObjects. Like Kyma, Blue's timeline interface provided a consistent and uniform interface for organising SoundObjects in time. However, unlike Kyma, SoundObjects were both user-extensible as well as developer-extensible as plugins.

With a single layer type, Blue's score interface was consistent from layer to layer for editing objects in time. With SoundObjects being a plugin-point, developers could create new SoundObject types to extend the available objects in Blue. In addition, users could further customise certain SoundObjects with their own scripting code (i.e., PythonObject, ClojureObject) and user interface designs (i.e., ObjectBuilder), providing a high degree of customisation and extensibility[10]. Conceptually, this allowed users to introduce new objects, similarly to how Kyma provides user-extensible objects.

[10]More information about various SoundObjects are available in [195, 3. Reference - SoundObjects].

The design of a single layer type with heterogeneous objects can be very flexible within the confines of the objects themselves. However, there were two problems with this design in Blue. Firstly, coordinating and editing material within objects in relation to the internal contents of other objects may be difficult. Even with extensive visualisation of the object on the timeline, one still has to work outside the timeline and use the object's edit panel to modify its contents.

One possible solution would be to replace the use of bar renderers with bar panels. This would allow SoundObject developers to provide custom bar interfaces that might allow user editing of SoundObject contents directly on the timeline. This kind of interface has been implemented in Ardour for its MIDI layers and objects. This approach was considered for Blue but it did not seem to fit well at the time with the kinds of objects that currently exist in Blue. However, this approach may be revisited in the future.

Secondly, some forms of musical ideas do not fit in well with the paradigm of objects and bars on a timeline. Using bar panels may solve the first problem but does not address the second. One example would be timelines with western music notation, where time is organised into measures by meter, and notes and other markings are written within measures. Another example is a music program like UPIC, where one works by drawing lines directly on a timeline using digital pen and tablet. Both of these types of programs work with music in time in ways very different from the objects and bars model.

In looking at DAWs and sequencer applications, there is a precedent in having heterogeneous layer types for Audio and MIDI. Each layer type has unique interfaces and operations that are specific to that kind of layer. For example, in a MIDI layer, overlapping clips might play both the clips

concurrently, while in an Audio layer, it may create a cross-fade between the clips. In this scenario, while both audio and MIDI layers share an objects and bar model, they visualise and perform them differently, giving different behavior between layer types.

The goal for Blue's Modular Score timeline was to extend Blue's existing homogeneous layer model to become a heterogeneous model. When interacting with the timeline, the timeline should accommodate not only operations specific to each layer but also operations that are global to the score as a whole. In addition, the system should be developer-extensible, such that new layer types could be introduced by third-parties.

5.4.4 Implementation

Blue's modular score timeline was first introduced in version 2.3.0 and further refined over time.[11] The general design of Blue follows the Model-View-Controller (MVC) [102] object pattern but uses the convention found in the Swing GUI toolkit of combining the View and Controller within the same GUI object. The following will discuss changes introduced to Blue's *data model* design, followed by changes for the *graphical user interface*. Next, it will discuss how these changes are exposed to third-party developers as plugin interfaces. Finally, a summary will be provided.

Data Model Changes

Figure 5.3 shows a diagram of the previous Blue Score data model. In this model, a `Score` class was used that contained all data for the score timeline.

[11] The current version Blue is 2.6.0 as of the time of this writing. Versions 2.3.0 through 2.6.0 were developed as part of this thesis.

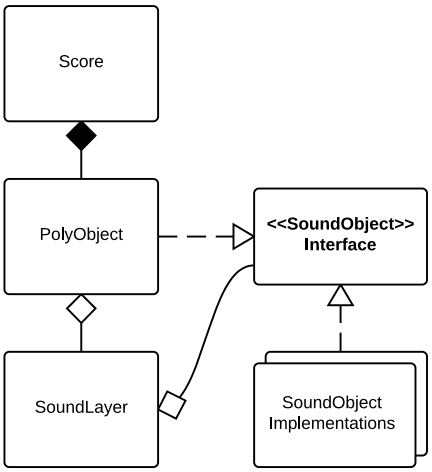

Figure 5.3: Blue Score: Old Data Model

The `Score` class contained a single top-level `SoundObject` called a `PolyObject`. The `PolyObject` class is a Composite [70] container class made up of a list of `SoundLayers`, each of which contained `SoundObjects`. As `PolyObjects` also implemented the `SoundObject` interface, this allowed adding of `PolyObjects` to `SoundLayers`, thus allowing timelines to be embedded within timelines. `SoundLayers` were presented in the user interface as vertically laid out rows, and `SoundObjects` were presented as bars within each layer.

Figure 5.4 shows a diagram of the new Blue Score data model. The modular score model introduces a new `LayerGroup` interface that acts as a container for `Layer` interface objects. Score has been modified from holding a single `PolyObject` to now act as a container for `LayerGroups`. The model also introduces two new sub-interfaces, `ScoreObjectLayer` and `ScoreObject`, which express that certain kinds of layers can provide `ScoreObjects`. By using the sub-interfaces, the system can distinguish what layers may or may not support layer objects.

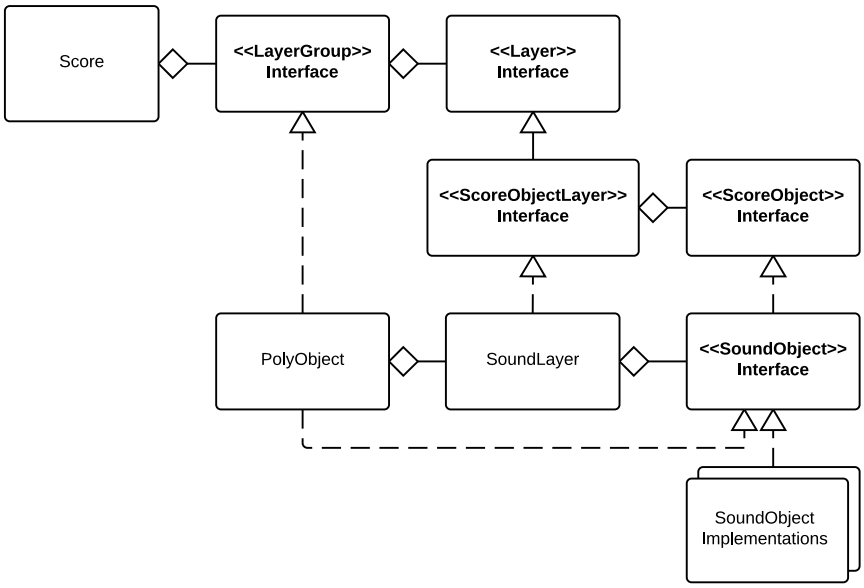

Figure 5.4: Blue Score: New Data Model

The diagram also shows how the previous concrete classes, `PolyObject` and `SoundLayer`, are now implementations of `LayerGroup` and `ScoreObject-Layer`. Also, `SoundObject` is now a sub-interface of `ScoreObject`. Making these classes implementations of the new interfaces allows the previous model to act as a subset of the new model. As a result, there is a clear path of migration for older projects and no data is lost in translation.

User-Interface Changes

In Blue's modular score, the user interface class design was modified to work with the new data model. Instead of working with a single `LayerGroup` (`PolyObject`), the program now must work with multiple `Layergroups`.

Figure 5.5 shows a diagram of the previous Blue Score UI model. In this model, the `ScoreTopComponent` held the editor for the `Score`. It used a hard-coded `ScoreTimeCanvas` component to render out the main timeline

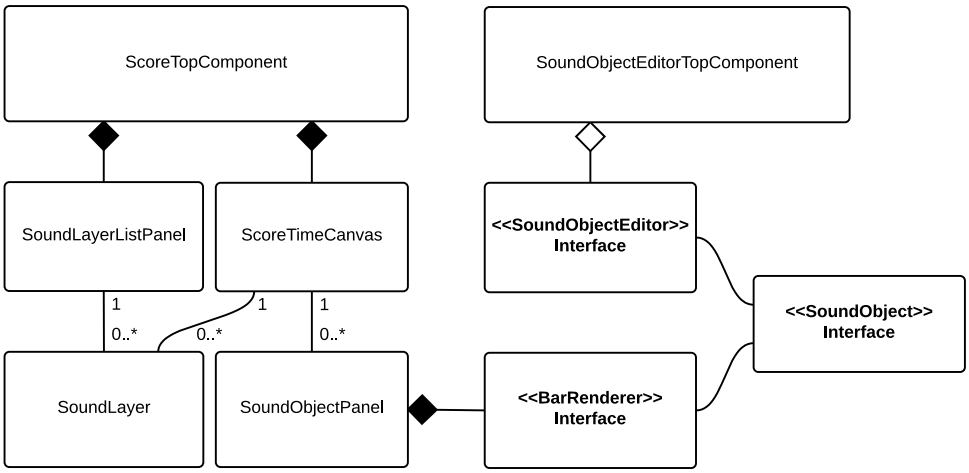

Figure 5.5: Blue Score UI: Old Class Design

of a `PolyObject`. A `SoundLayerListPanel` header component was used for the left-hand side of the score. The `ScoreTimeCanvas` renders each layer by reading their `SoundObjects` and creating `SoundObjectPanels` for each one. Each panel used the associated `BarRenderer` for a SoundObject for visualising the object on the timeline. The `SoundLayerListPanel` would create `SoundLayerPanels` for each `SoundLayer` for editing properties for the layer.

Figure 5.6 shows a diagram of the new Blue Score UI model. The ScoreTopComponent now uses a fixed `ScorePanel` and `LayerGroupHeaderListPanel`. Each of these new components works as containers for multiple `LayerGroup` headers and panels.

A new `LayerGroupUIProvider` Abstract Factory [70] interface was introduced that is associated with `LayerGroups`. When the `ScoreTopComponent` receives a `Score` to edit, it will use the the associated provider for each `LayerGroup` to create a header and panel. These are then added to the `LayerGroupHeaderListPanel` and `ScorePanel`.

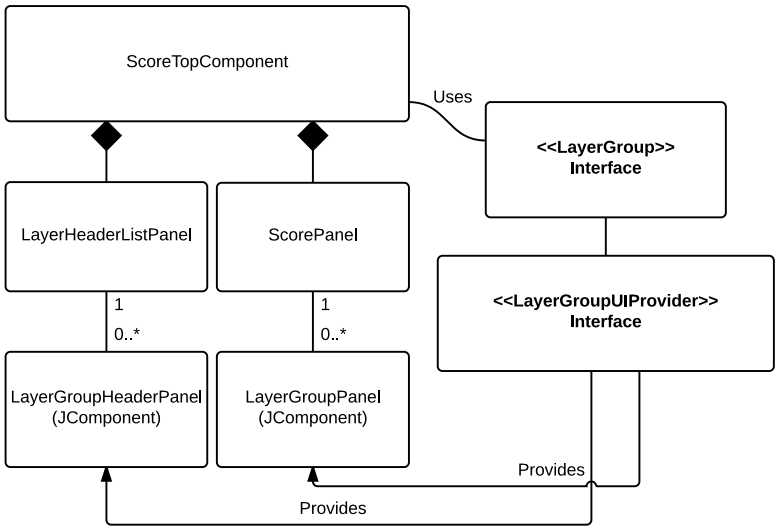

Figure 5.6: Blue Score UI: New Class Design

Figure 5.2 shows an example of the modular score. In the score are three LayerGroups: SoundObject (i.e., PolyObject), Audio (see Section 5.4.5), and Pattern (see Section 5.4.5). The header list on the left and the timeline on the right both show the corresponding panels provided by the `LayerGroupUI-Providers`. Space is introduced between panels for `LayerGroups`, similar to how orchestra scores in Western music notation group instrument families together and provide space between groups.

To use the modular score, a new Score Manager dialog was introduced (shown in Figure 5.7). This dialog is used to add, reorder, and remove LayerGroups to the Score. User can also add, reorder, and remove Layers for LayerGroups, as well as edit properties for the LayerGroup using property panels provided by `LayerGroupUIProviders`.

Finally, mouse handling for the timeline was redesigned. Each LayerGroup panel can implement its own mouse handling code. However, some mouse operations should be global to the entire score. A new `BlueMouseAdapter`

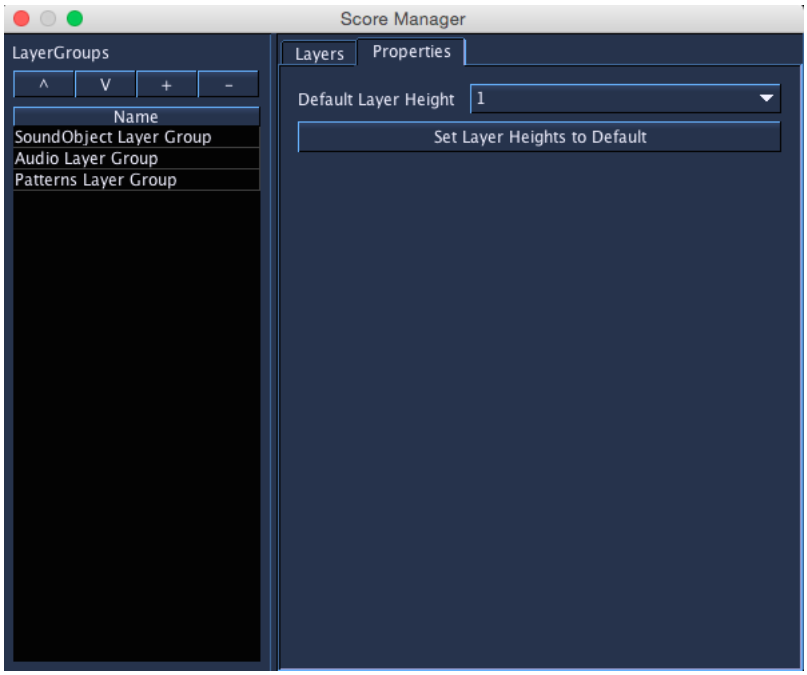

Figure 5.7: Score Manager Dialog

plugin was introduced that allows new mouse handlers to be used on the top-level of the Score timeline. If a LayerGroup panel does not mark a mouse event as being `consumed`, the top-level mouse handlers get an opportunity to handle the mouse events.

When migrating to the modular score design, the former `ScoreTimeCanvas` and `SoundLayerListPanel` were updated to be components returned from the `PolyObjectUIProvider`. In addition, much of the mouse handling was updated and moved from the `ScoreTimeCanvas` to a `BlueMouseAdapter`. The mouse code was modified to work with `ScoreObjects`, allowing operations like object selection, moving, and resizing to work with `ScoreObjects` of different types and across different `LayerGroups`.

Plugins

The Blue Modular Score introduces new interfaces to both the data model and UI class model. These interfaces are plugin points that allow new kinds of LayerGroups to be introduced into Blue. This can be done either by core or third-party developers.

Plugins are handled in the standard way for Netbeans RCP applications. The plugin interface is defined in one of Blue's core modules. The package is marked as public in the module's manifest [33, Chapter 3: The Netbeans Module System] so that other modules can use those interfaces. Other modules add a dependency on the Blue core modules and implement the plugin. They then register the plugin using the System Filesystem [33, Chapter 7: Data and Files]. From here, Blue uses the Lookup [33, Chapter 5: Lookup Concept] system or Filesystem API to discover all plugin implementations.

Summary

To implement the new Modular Score, new Java interfaces were introduced for both Blue's data model and its user interface class model. The Score user interface was updated to work with and organise `LayerGroups`. These interfaces are exposed as plugins points and can be implemented in new modules, whether created internally within Blue or externally by third-parties.

5.4.5 Case Studies

This section will discuss two types of `LayerGroups` that were introduced since the implementation of Blue's modular score timeline: Patterns and Audio. These `LayerGroups` will show different approaches to working with material

in time and how their unique interfaces offer something more than what the previous homogeneous score model could offer.

Pattern LayerGroups

Pattern `LayerGroups` were first introduced in Blue 2.3.0. Pattern LayerGroups are split into `PatternLayers`, each of which has a `Pattern` and a `SoundObject`. The `SoundObject` for a `PatternLayer` is used as source material for the layer, and the `Pattern` tracks where in time to perform the `SoundObject`. For each location of the pattern selected, the `SoundObject`'s generated score is used to play for the duration of the pattern's box. The length in time of each pattern box is configured in the `PatternLayerGroup` and shared by all `PatternLayers` for the group.

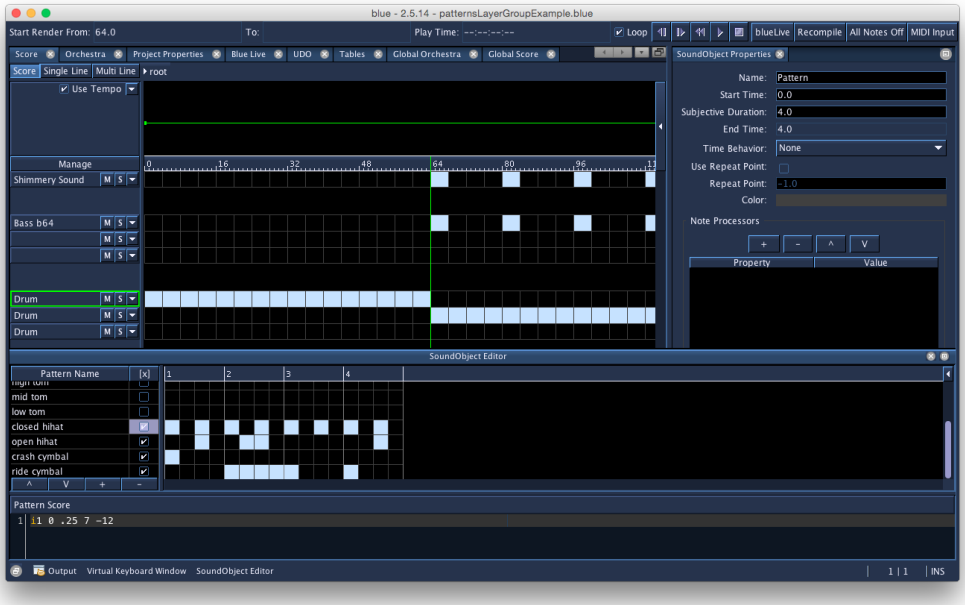

Figure 5.8: Pattern Layers

`PatternLayers` are useful for music which repeats material. Figure 5.8 shows a `PatternLayer` used together with a `Pattern SoundObject` to draw in a drum pattern. Once the drum pattern is created, the user can then select each place on the timeline where they would like that pattern to play. Because the pattern is both drawn and edited on the timeline itself, the user can be very quickly fill in locations to play the pattern. As Figure 5.8 shows, `Pattern` and `SoundObject LayerGroups` can co-exist on the timeline together, allowing the user to choose which representation best suits the musical idea they want to represent and work with.

The user interface for `PatternLayers` uses custom mouse actions that differ from `ScoreObject`-based layers. For example, a user can press down with the mouse in one box, drag the mouse over multiple locations, then release the mouse. In that one gesture, the user would have filled in multiple boxes within the pattern. Afterwards, they might then click to deselect some locations. Using `ScoreObjectLayers`, the closest interface gesture would be to copy a `ScoreObject`, then command-click to place copies of that object in multiple places on the timeline. For those working with pattern-oriented music-making, the `PatternLayers` interface may be considered to be more efficient and optimal to the kind of music being represented than that of using `ScoreObjects`.

Audio LayerGroups

Audio `LayerGroups` were introduced in Blue 2.6.0. They offer the same kinds of functionality as one would find in DAW software, allowing the user to organise audio clips in time. Blue's audio layer functionality is modeled upon

the features and behaviors found in other existing DAW software. Figure 5.8 shows an example of using Audio layers.

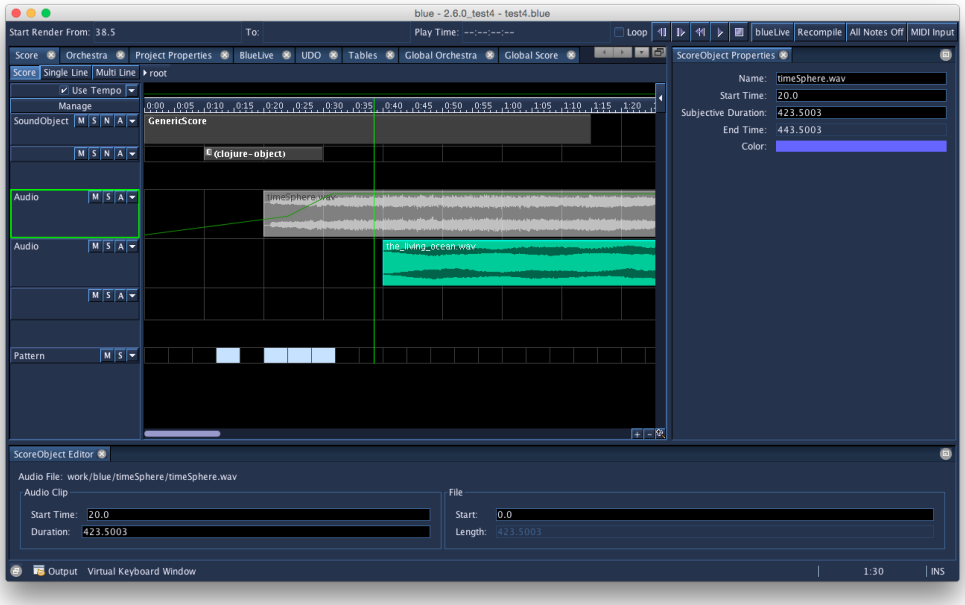

Figure 5.9: Audio Layers

With `AudioLayers`, users can drag and drop audio files on to the timeline to create `AudioClips`. The mouse interactions for `AudioClips` should be familiar to those who have experience with other DAWs. User can select and move clips as well as resize from the left- and right-hand sides of objects to adjust both the clip's start/end time and audio file's start/end time.[12] Fade-in and fade-out times may also be adjusted.

`AudioLayer` is implemented as an instance of the `ScoreObjectLayer` interface. Also, `AudioClip` objects implement the `ScoreObject` interface. This allows mouse handling to work across Audio and `SoundObject LayerGroups`

[12]The `AudioClip` has properties such as when to start, but also where to start playing within the audio file the clip refers to.

and general mouse gestures for selection and movement to work with material from either type of layer.

`AudioLayers` also work with Blue's Mixer and Effects system. Each `AudioLayer` maps to one auto-generated channel within the mixer. All audio from the layer is routed through the mixer, where effects can be added. Users can automate parameters for effects set on the layer's channel directly on the layer's timeline panel.

5.4.6 Summary

Blue's modular timeline meets the goal of providing heterogeneous layers in an extensible way. By making `LayerGroups` within Blue a plugin, developers can extend the Score timeline with new kinds of layers. The introduction of Pattern and Audio `LayerGroups` shows that layers with very different and very similar interfaces to the pre-existing `SoundObject` `LayerGroup` can be implemented as plugins.

5.5 Conclusions

Module-based systems and modular programming techniques provide an extensible foundation for computer music software. These systems and techniques extend the monolithic single-executable with plugins model to make the entire application a set of plugins organised into modules. Module-based systems simplify exposing and handing plugin points, encouraging the same kinds of freedom to extend the system by third-parties that are offered by multi-executable systems.

For this thesis, the module-based Blue music system was extended to implement heterogeneous layer types in its score timeline. This was performed by introducing new plugin points for `LayerGroups` and `LayerGroupUIProviders` and exposing them for third-party implementation. Two new `LayerGroups` were implemented: Audio and Patterns. As a result, users now have options from which to choose to best represent their musical ideas, and they can organise various representations of music together in time.

Chapter 6

Music Systems as Libraries: Pink and Score

This chapter will look at music systems as libraries. This kind of system exists when libraries are developed for use by users within general-purpose programming languages (GPLs).[1] I will compare this with music systems using domain-specific languages (DSLs) and look at what each design offers to developers and users in terms of extensibility and robustness of works.

After the analysis of language-based systems, I will look at two music libraries written in the programming language Clojure – *Pink* and *Score* – that I have developed for creating library-based musical works. Pink is a library for sound and music computing, and Score is a high-level library for working with symbolic representations of musical events. In the discussion of each project, I will explore related systems, discuss the design and implementation of the library, and look at how they address extensibility. Although each

[1]For this chapter, I will use GPL to refer to general-purpose programming languages. This is not to be confused with the GNU Public License, a commonly used license for open-source programs.

library is designed to function alone, their use together enables additional features that will be explored later in this chapter.

6.1 Introduction

Language-based music systems are the foundation of computer music. Starting with Max Mathews' *Music* series of programs and moving through software such as Csound and SuperCollider 3 today, music systems that offered their own languages have been a fundamental part of the history of computer music software.

Over time, as computing developed and the expectations for the target user changed, music software evolved into different forms. Some moved more towards technical users – users who are themselves developers – while others moved more towards non-technical users – users with no knowledge of programming. Programming with text and manipulating graphical user interfaces represent the extremes by which users develop their works today.

The following will focus on the more technical end of the computer music spectrum: using libraries as music systems. In this approach, music systems developers create libraries for use within an existing general-purpose programming language. This is in contrast with building language-based music systems where a domain-specific language and interpreter are implemented by the system itself.

I will begin by comparing language-based systems using DSLs and GPLS. I will then discuss the differences in architectures between the two and the impact that library-based designs have on extensibility and the user's work. I will then discuss two new libraries for music making, Pink and Score, that are written in Clojure and designed with extensibility in mind.

6.2 Language-based Systems

Language-based computer music systems are those where the user writes text to program musical ideas. The languages used for music systems may be *domain-specific* or *general-purpose*. In the former, a system provides its own custom programming language tailored to the problem domain of musical computing. In the latter, a music system is written in an existing general-purpose programming language, with the purpose of being used by users within that language. Both kinds of system offer users means of extensibility by user code, plugin, or programming library. They both also require users to be or become programmers.

Computer music has long had a history of language-based systems, starting with the original computer music software, Max Mathews' Music-N series of programs. Early computer music was rooted in a culture where learning to program in a DSL or GPL was a requirement for making music with the computer. Later systems – such as Csound, SuperCollider 3, and Common Lisp Music – continue the tradition of offering language-based systems. While today's landscape of computer-based music making has diversified to include many other kinds of programs, language-based systems continue to be attractive to users who find expressing musical ideas through text aligns with their way of thinking about and working with music.

In the following, I will look at the designs of language-based systems using domain-specific languages and general-purpose languages. I will look at how each relates to the user's work in terms of dependency management and user control.

6.2.1 Domain-specific Languages

Music systems using DSLs are in control of the language and the work. Users write code then execute the system with their code. The music system initialises, compiles the code using its built-in interpreter, and then runs the engine to render the project. Further code compilation may occur if the system supports compilation at runtime.

An example of a DSL-based system is Csound. Csound's Orchestra language is used to define instruments, opcodes, and data, and can also define top-level commands to execute.[2] This language is tailored specifically to the domain of Csound's music programming model.

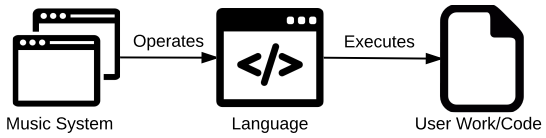

Figure 6.1: Control Graph for DSL-based systems

Figure 6.1 shows the control graph for DSL-based systems. With a system like Csound, the system controls (i.e., operates) the language, which in turn controls (i.e., executes) the work. The top-level point of entry is thus the system. The user then works within the context of both the language and system to develop their work.

6.2.2 General-purpose Languages

Music systems designed for use with GPLs operate within the context of the language. In these systems, the language and its runtime is the top-level point of entry. The music system's status then is just another library for use

[2]In Csound parlance, top-level commands are written in instrument 0 space.

within the language. The user's work, which employs the music system as a library, controls the system.

An example of a GPL-based system is Common Lisp Music (CLM) [160]. CLM is written in Common Lisp and provided as a library. Users write their works in Common Lisp and use features provided by CLM. The system then operates within the context of the work. (CLM is discussed further below in in Section 6.4.1.)

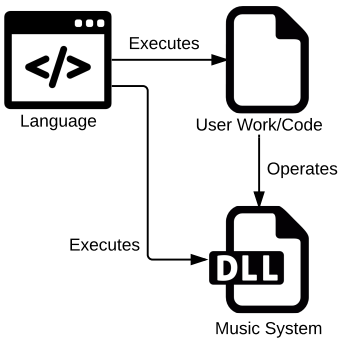

Figure 6.2: Control Graph for GPL-based systems

Figure 6.2 shows the control graph for GPL-based systems. With a system like CLM, the language is the top-level point of entry. The language executes the user's work and system, and the work controls (i.e., operates) the system. Other examples of GPL systems and languages include CMix [104] and C; PyO [30] and Python; and Incudine [105] and Lisp.

6.2.3 Discussion

With DSL-based systems, the developers of the music software design, implement, and maintain both the DSL language specification as well as the interpreter. This is in addition to the musical aspects of the software. DSL-based software then requires developers to understand at least two domains

of knowledge: firstly, music systems, and secondly, programming language design and implementation. This is in contrast to GPL-based systems, where developers are dependent upon a third-party implementation of a language. In this case, the developer must know how to use the language but does not have to implement or maintain it. Developers then will only have to worry about a single domain of knowledge – music systems – for their implementation.

Implementing and maintaining a DSL is non-trivial. Doing so has an impact on both user extensibility and robustness of the software. In terms of extensibility, the developer must not only implement parts of a system but also *expose* parts of the system to the user through the DSL. Beyond exposure, the DSL must also be expressive enough to be able to extend the parts of the system. For example, in Csound, instruments are a part of the system that is exposed to the user both to use and extend. The Orchestra language is designed for the problem domain of musical programming, of which defining instruments is an integral part. However, the processing order of instruments is a part of the Csound system that is not exposed for users to work with in the DSL. Even if it was exposed to the user, the Orchestra language would not be well suited to the kind of programming required to extend that part of the Csound system.

In terms of robustness, implementing a language requires a skill set outside of the domain of music systems. A DSL-based system then requires more knowledge to maintain the system than a GPL-based system. Finding developers with the requisite skills is thus a factor in measuring the robustness of a program over time.

With GPL-based systems, the issues of language design and maintenance and its impact on the system's robustness are delegated to third-party language

developers. This is largely a positive factor, as the language developers are likely experts in their domain and can well-support the system. This frees up the developers of GPL-based music systems to focus on their domain.

Also, in regards to extensibility, as users and developers work within the same language, the work of exposing parts of the system are minimised. Developers need only focus on their decisions on what to expose to users without having to worry about how to do so. If a part of the system is exposed to users, the developer will not have to do extra work to address language differences between the system language and the user language.

Aside from the issues in working with and supporting the language implementation, the issue of control is an interesting one in terms of how it relates to extensibility. In a DSL-based system, the user's work is processed within the life cycle of the system and may have little ability to control the system. In a GPL-based system, as the system is a library, the user's work is in complete control over the life cycle of the system. For example, the user can execute code before the system is even instantiated and started.

In summary, DSL-based systems offer users a language that is customised to a problem domain. The cost of using a DSL for developers is the time required for development and maintenance of a language implementation. The cost for users is potentially a loss in extensibility and control of the system. GPL-based systems use an expressive language that is not domain-specific. It may then be more difficult to learn and use for music. However, there is generally less work for the developer to maintain these systems, and users can more easily extend and control the system.

6.3 Introduction to Pink and Score

Pink and Score are music libraries designed for use in the general-purpose programming language Clojure. They are developed as stand-alone projects that may also be used together. (This design was inspired by Common Lisp Music and Common Music, which will be further described below.) Each has features that target the domain in which it is designed to serve: audio synthesis and processing for Pink, and higher-level symbolic representations of musical events for Score. Because Pink and Score are both written in the same language, new and interesting musical possibilities emerge when they are used together.

The following will discuss common aspects of Pink and Score. Section 6.4 will explore the design and usage of Pink as well as look at related systems. Section 6.5 will do the same for Score.

6.3.1 Clojure

Clojure, created by Rich Hickey and made publicly available in 2009, is a general purpose programming language. Regarding the language, the official website states:

> Clojure is a dialect of Lisp, and shares with Lisp the code-as-data philosophy and a powerful macro system. Clojure is predominantly a functional programming language, and features a rich set of immutable, persistent data structures. When mutable state is needed, Clojure offers a software transactional memory system and reactive Agent system that ensure clean, correct, multi-threaded designs. [89]

Additionally, Clojure is open-source and developed as a hosted-language that operates in conjunction with a known platform. Currently, three primary Clojure implementations exist: Clojure, Clojurescript, and ClojureCLR. Each of these implements the Clojure programming language and they are hosted on Java, Javascript, and C# languages and platforms respectively. Further information about the language can be found in [86], [64], and [69], as well as on the official website.

For Pink and Score, I chose to use Clojure as I found the language appealing to use. In particular, I found that functional programming practices suited the problem spaces well for both audio and high-level musical descriptions.[3] I also found Clojure's operation on the JVM to be beneficial as the JVM is open-source and provides platform extensibility. This also opened up the possibility to use Pink and Score with other JVM-based music systems, such as my own music program Blue.

6.3.2 Open Source Software Stack

An important aspect of choosing Clojure for Pink and Score is that one can develop works using a completely open-source software stack. This means that all software used for a work – from the lowest-level, the operating system, to the highest level, Clojure itself – can be open-source software. Figure 6.3 shows a possible system dependency diagram where all dependencies are open-source.

[3]Desain describes the benefits of using Lisp [122] in [58] for symbolic processing in music. I think since the time of Desain's article, computing has increased in performance enough that it is well suited for signal processing in real-time as well.

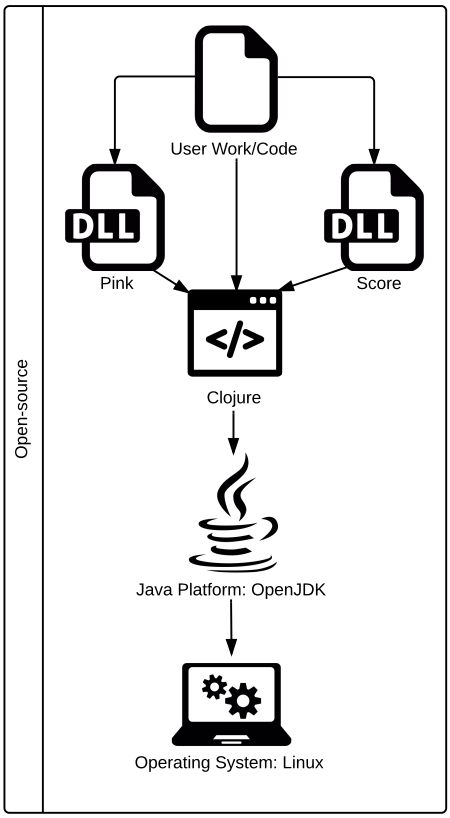

Figure 6.3: Pink/Score Dependency Graph: Open Source

While the software stack for Pink and Score can be open-source, it is not a requirement. Pink and Score function equally well using closed-source dependencies. Figure 6.4 shows a possible system dependency diagram where some dependencies – the operating system and Java Platform – are closed-source. This allows for a larger range of dependency configurations to be used and promotes cross-platform development and use.

Having the possibility to use a completely open-source software stack was an important design requirement for Pink and Score. Open-source platforms provide options for maintenance (i.e., development can continue by third-parties if the original developers discontinue work) that closed-source

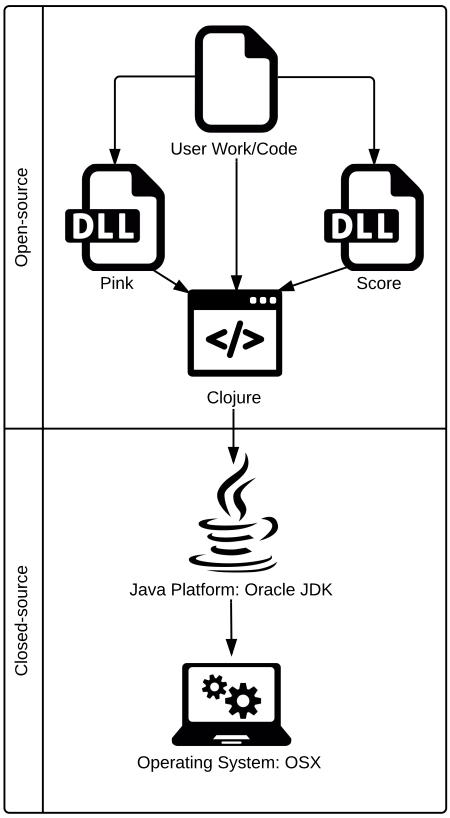

Figure 6.4: Pink/Score Dependency Graph: Closed Source

platforms do not offer. This in turn increases the potential robustness of the dependencies and the work. Even if a user decides to work with a closed-source configuration, they can rest assured that their work will continue to function if support for those dependencies ends.

6.3.3 Cross-Platform

Another important concern when designing Pink and Score is that they should be cross-platform. Pink and Score are designed specifically for the Java-platform version of Clojure. As a result, they inherit the availability of platforms that is provided by the JVM.

The JVM is available on a number of operating systems. This includes the three major OSs – Windows, OSX, and Linux – as well as other platforms like the BSD-family of operating systems. This includes different versions of these operating systems as well as variants (e.g., Windows 7 32-bit (i386), Windows 8 64-bit (x86_64), OSX 10.10 64-bit (x86_64), Linux 64-bit (amd64), etc.).

It is important to note that the JVM protects the developer from differences and changes in hardware. For example, a developer can depend upon the byte order representation of data as big endian in Java, whether or not they are running on hardware that is natively big endian or little endian. Another example is that the same application may run the same whether it is run on a 32-bit CPU or 64-bit CPU (and potentially whatever CPU architectures arrive in the future). As long as the JVM can be ported to a platform and it complies with the Java Virtual Machine specification [116], a JVM-based application will run on those platforms.

Making Pink and Score work across platforms was a requirement set out at the beginning of their designs. The cross-platform support of the JVM and first-class interoperability with the Java platform were primary factors in determining the applicability of Clojure for this work. With protection of changes in hardware and potential porting of the JVM to new operating systems, it is expected that Pink and Score can work well not only across existing platforms available today but also upon new platforms into the future.

6.3.4 Design Practices and Goals

I designed Pink and Score using features and practices commonly found in the Clojure programming world. This includes not only functional programming practices in general but also Clojure-specific idioms. It was a goal to write both

libraries such that those familiar with functional programming languages could understand the design, and those familiar with Clojure would understand the implementation.

In particular, both projects make use of *closures* and *higher-order programming*. With closures, functions in Pink and Score can return functions that *enclose* over data and capture them for use as part of their processing. The returned function can then reference and use the enclosed data, even though they are not passed in as arguments to the function. This allows for creating stateful functions and was particularly useful for implementing the concept of *unit generators* in Pink and score parameter generators in Score. Also, a number of key parts of each library are designed to generically work with functions as arguments. These aspects of higher-order programming are employed liberally in Pink and Score. For further information about closures and higher-order programming, see [93].

Macros are used for compile-time programming. Clojure – and Lisp in general – is a *homoiconic* language. Homoiconicity refers to the quality of a language where "their internal and external representations are essentially the same" [97]. One writes Clojure code using the syntax of lists, maps, numbers, and so on, and internally the code is read in and represented using the same data structures before it is evaluated.

This allows one to write macros, which are functions that operate on code as data structures, most often for the purpose of transforming the code. In essence, macros are code that operates on code. When the Clojure compiler goes to compile code and encounters a macro, the macro is allowed to process the containing code and generate new code. The final code is then what is compiled by the compiler.

Pink and Score use macros judiciously and as minimally as possible. In general, macros allow one to extend the language itself and introduce new abstractions and shapes to code. With Pink and Score, I wanted to rely upon standard abstractions and concepts as much as possible. Users could then focus on learning and using the features of each library, without having to additionally learn new high-level language constructs.

Other aspects of design include *reusability* and *dependencies*. For reusability, I wanted to make as much of each library reusable, meaning that the functions are designed for use not only within the context of the library but also on their own outside of the library. For example, in Pink, the audio engine uses a scheduler system. In some music systems, the user might be able to use the scheduler through the engine but may not be able to reuse that part on its own. This may be because the scheduler is designed specifically for that system, or the developer simply did not expose that part for public use. However in Pink, the scheduler and other parts of the engine are designed for stand-alone use and are publicly available for users to use. This allows users to reuse parts of Pink so that they can develop their own engines and have greater flexibility in modifying the system for their work.

Finally, for dependencies, I wanted to create libraries that had as few dependencies as possible. As a result, besides what is provided by the JVM, Pink and Score only depend upon Clojure and no other libraries. Minimising the dependencies provides the smallest baseline possible for users when choosing to use these libraries. When they opt to add additional libraries to their work, they can expect that Pink and Score will not add a number of other dependencies that may conflict with the new libraries or

their dependencies. This then simplifies the dependency graph of works that depend on Pink and Score.

6.3.5 Libraries and Versioning

One important aspect of developing libraries for use within a general-purpose programming language is versioning. As user's works are themselves programs, the build system for the work can take into account the version of the library required for the project. This provides a great freedom for the library developer as incompatible changes may be introduced without fear of endangering the functioning of existing projects. The user also benefits in knowing that a prior work may continue to function even when new versions of dependencies become available, as they can continue to use a specified version.

In the Clojure world, the Leiningen [84] build tool is often used when creating projects. It uses a Maven-compatible dependency resolution system for downloading libraries used by a project.[4] Users specify the *identifier* and *version* of the library they wish to use in their `project.clj` file. Leiningen in turn searches known Maven repositories for the specified versions of those libraries.

```
(defproject my-music-project "0.1.0-SNAPSHOT"
  :description "Example music project"
  :url "http://some-url.com"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.7.0"]
```

[4]Maven is a build tool based around a concept of a project object model. One specifies dependencies and they are retrieved from a repository. It is a popular tool and the Maven Central repository [23] contains many libraries. For more information on Maven, see [127].

```
    [kunstmusik/pink "0.2.1"]
    [kunstmusik/score "0.3.0"]])
```

Listing 6.1: Example Leiningen project.clj file

Listing 6.1 shows an example Leiningen project file. It defines a project called my-music-project that depends upon version 1.7.0 of Clojure, version 0.2.1 of Pink, and version 0.3.0 of Score. When this project is first built or run, Leiningen will check to see if all of those dependencies are available locally and, if not, search for them online. As long as the dependencies are available online, the project will be able to download and satisfy all dependencies and operate. If a new version of library is published, it will not affect the project as it will still use the specified version.

Pink and Score are packaged and deployed to the Clojars [2] repository. This repository is popularly used by the Clojure community for releasing and locating libraries, and supports housing multiple versions of libraries. In addition, for safety, I retain a backup copy of all versions of Pink and Score libraries. If for some reason Clojars was to cease operation, I am able to create a Leiningen-compatible repository with the backups.

Versioned libraries together with build tools can greatly simplify management of dependencies for a musical work. By specifying a specific version of a library, the project can isolate itself against changes to dependencies. This liberates the system developer to continue developing the system without fear of breaking users' existing works, while also providing stability for users' works to continue to function over time.

6.3.6 Summary

Pink and Score are music system libraries developed using functional programming techniques, employing features such as closures, higher-order programming, and macros. They are designed to maximise reuse of their parts and to empower the user to extend the system. These systems inherit Clojure and the JVM's cross-platform and open-source properties. Finally, users' works using Pink and Score can depend upon specific versions without fear of changes in new versions, thus increasing the robustness of the work over time.

6.4 Pink

Pink is an audio engine library for building music systems and works. It includes functions for building engines, signal graphs, audio functions (similar to *unit generators*), and other utility code. It provides a pre-made system to extend and use, as well as exposes the core set of functions that users can use to assemble their own engines and systems. The library is designed to have the least amount of abstractions necessary to implement the largest number of musical use cases.

The following section will discuss the design and architecture of Pink. I will begin by looking at related systems and how they influenced Pink's design. I will then discuss the general design of the system, followed by the implementation of each part of Pink. Use of the library will be shown in the full project example in Section 6.6.

6.4.1 Related Work

The design of Pink has been influenced by many other systems. The following will begin with discusssion of CLM, Nyquist, and Extempore, as their use of Lisp is directly relevant to Pink. Other systems and their influences will follow.

Common Lisp Music

Common Lisp Music (CLM) [160] is a Music-V based system written by Bill Schottstaedt for use in the Common Lisp (CL) language.[5] A portion of the signal processing functions of CLM are written in C and exposed to the Lisp side of the system using the host CL interpreter's Foreign Function Interface (FFI). CLM employs a GPL-based design, in contrast to the DSL-based design of Music-V which it is based on.

CLM's processing model first ahead-of-time compiles each note as audio to disk, then coalesces the audio from notes into the final audio file. It then plays back the generated sound file in real-time. CLM is suitable for composed works and audio processing utilities but less so for real-time composition where changes to the work will occur during rendering.

CLM's architecture is rooted in the design of Music-V and similarly employs concepts of *instruments* and *note lists*. Instruments are generally written in an imperative way, where the body of the code is a loop that writes a calculated signal to the output stream using the `outa` or `out-any` functions. The `with-sound` macro, used to organise and render note lists, does allow for adding reverb to its child instrument instances using the `:reverb`

[5]This document will discuss the Common Lisp version of CLM. Other variants of CLM exist in C, Scheme, Ruby, and Forth. For more information, see [159].

keyword argument and a supplied reverb function. Within `with-sound` calls are a list of calls to instrument functions with a start time, a duration, and other arguments. As `with-sound` can be embedded within other `with-sound` instances, a tree of audio signal processing can form.

CLM is designed for user-extensibility at the level of instruments and unit generators using Lisp code. Note list writing can be done by explicitly writing each event or using Lisp code to generate events. CLM has also long been used in conjunction with Heinrich Taube's Common Music (CM) [175]. CLM's capacity to use Lisp at all levels – from generators to note lists – is appealing, as is its interoperability with other libraries. That CLM has its own event system, but can also work with other libraries like CM, was also inspiring.

However, for Pink, I wanted to make a system that better supports development of real-time musical works, as well as move away from the Music-N paradigm of instruments and generators to a more generic, functional audio graph. Additionally, I wanted to expose more of the engine's parts to the user than is offered in CLM, so that they could explore modifying the existing engine or create new engines should they desire to do so.

Nyquist

Nyquist [52], written by Roger Dannenberg, is a music system written in C that offers both a Lisp and SAL language front end. It differs in design from Music-N systems, using a more functional programming approach where audio functions return their signal data when they are called. Instead of having a separate concept of instruments and generators, as is in Music-N, Nyquist users will instead compose and use a new audio function built up

of other audio functions. This allows signal routing to be determined by the caller, rather than the callee as is done in Music-N systems. This "dual nature of a patch", such that a patch acts as both an object (unit generator) as well as a signal, was further developed by Dannenberg in [53] in discussing his music language Serpent, which later became a part of his real-time music system Aura [54].

Additionally, Nyquist does not differentiate between scores and orchestras as is found in Music-N systems. Instead, it uses *temporal control constructs* [52] together with function applications to have a comparable system of timed functions. Nyquist provides a number of essential time transformations that allows one to easily build up larger score material from smaller ones.

Nyquist's processing model is block-based, similarly to most Music-N systems. One unique aspect, as discussed in [51], is that signal generating functions can compute ahead of the current block and cache those results. This allows for a signal to be realised only once, regardless of how many other functions read from that signal. This also allows for one to calculate values ahead of time for the total signal, such as the maximum amplitude of a note. However, the drawback is that realizing ahead-of-time signals is not particularly suitable for real-time processing.

Nyquist's approach to audio functions and instruments as aggregate audio functions was influential in Pink's design of its unit generators. This provides a great deal of extensibility and reuse of audio functions for users. While Nyquist's breaking down of the orchestra and score offers unique features, this design was not used directly in Pink. Nyquist was also not designed for real-time use, which was an important requirement for Pink's design. Finally, while Nyquist provides a Lisp front-end, it is not designed for use in general-

purpose Lisp interpreters. This limits interoperability with third-party Lisp libraries.

Extempore

Extempore [166], by Andrew Sorenson, is a music system designed for "cyber-physical programming" [165], where the user participates as part of the system by live-coding. Extempore develops upon the work done in an earlier system, Impromptu [164], and introduces its own Scheme-like language, *xtlang*, that is "designed to mix the high-level expressiveness of Lisp with the low-level expressiveness of C." [166]

One of the key influences of Extempore and Impromptu is the focus on what Sorenson calls *temporal recursion* [167], which allows an event function to schedule another event using the same function later in time. As Sorenson points out in [167], this idea was explored earlier by Dannenberg in the CMU MIDI Toolkit [48], which includes MoxC, which was itself based on Collinge's MOXIE system [44]. This is also explored in Csound by Lazzarini in [106].

Extempore is well-designed for real-time usage, a trait I wanted in my own system. Also, temporal recursion as a style of programming was something I wanted to support in Pink. However, unlike Extempore and Nyquist, I wanted to build a system that would operate in the context of a general-purpose language.

Other Systems

ChucK [188] is an audio programming system. It implements its own programming language, virtual machine, and audio engine. It employs a single-sample processing model. ChucK's use of *shreds* as concurrent processing code was

an influence on the goals of what could be done with Pink's control functions, though the design and implementation differ.

Faust [133] is a domain-specific language that uses a purely functional programming style for DSP. The user writes code in the Faust language, then compiles it for use within other music systems. Faust does not provide a full music system, as it does not have support for events. However, Faust's language excels at providing users the ability to express a lot of DSP programming with very little code. Faust's design was influential on the functional programming approach to audio functions taken by Pink.

Overtone [12] is a music composition system written in Clojure. It employs SuperCollider 3 as its audio engine and functions as a replacement for sclang as a frontend language. It inherits all signal processing and engine properties from SC3, such as block-based processing and a client/server model. It uses its own scheduler for processing of event functions that is run in a separate thread from the SC3 engine. The design of Overtone is largely oriented towards live coding and real-time performance. It is a mature system that shows how Clojure can excel for musical programming, but its dependence on SC3 limits one from writing low-level signal processing functions and exploring audio engine research within the Clojure language itself.

Kyma [155] is a commercial object-oriented music system written in Smalltalk that employs a graphical user interface and custom signal processing hardware. The original design discussed in [155] describes a class hierarchy to categorise the sound processing objects within the system. The `Sound` class acts as the basic unit generator interface that `SoundAtoms` (source signal generators) and `SoundTransforms` (signal processing units) implement. Pink adopts a similar philosophy to Kyma's unit generator design by considering

audio functions as a "stream of samples", audio function interfaces as "uniform" where any function can substituted for any other, and edges between nodes (i.e., audio functions) in the graph as representing an "is-a-function-of" relationship. [156] However, Pink uses coding conventions rather than class types for its definition of audio functions due to the dynamically-typed nature of the Clojure programming language. Pink is also designed as a software-only system that is employed as a programming language library.

The Create Signal Library (CSL) [143] is a "low-level synthesis and processing engine" [143] written in C++. CSL3 used an object-oriented design based on `Buffers` as signals, `FrameStreams` as signal generators and modifiers, and `IO` as driver abstraction for network and hardware communication. CSL3 also provided an `Instrument` utility class to simplify building signal processing graphs and exposing control parameters. CSL4 [142] adopted the MetaModel for MultiMedia Processing Systems (4MPS) [16, 15] that originated in CLAM [15, 17, 18]. 4MPS uses *Processing Data* as the abstraction for representing signals, *Processing Objects* for signal generators and modifiers, *Ports* and *Connections* for synchronous signal connections between Processing Objects, and *Controls* and *Links* for asynchronous event data connections.

Pink's design has both similarities and differences to CSL and CLAM. Pink, like CSL, is not a music representation language (MRL) but is designed to work together with MRLs. Pink does include a scheduler and event system, which CSL does not. Pink can support developing not only the same kinds of realtime audio applications that CSL and CLAM are capable of supporting but also music works employing classic computer music composition practices developed around an event system that operates synchronously with the audio engine. Pink also includes a control function system for synchronous

processing of non-signal processing functions, a feature neither modeled directly within 4MPS nor provided by CSL or CLAM.

Pink's design for its signal processing graph is simpler than 4MPS and does not directly model Ports, Connections, Controls, or Links. Instead of modeling these concepts directly, Pink focuses on the minimal abstraction of the audio function with most node connections of the graph created statically through node references passed as arguments to audio functions at initialisation time. Pink does support the dynamic connection features of Ports and Controls and their synchronous and asynchronous processing features by providing higher-level features (e.g., Nodes, discussed in Section 6.4.3) built on top of audio functions. The result is that Pink users not only can avoid the layers of indirection that a Proxy [70] object like Ports and Controls introduce but also can choose to use Proxy-like functions only where necessary.

Finally, Pink's processing graph does not support Observers [70] on audio functions as is found in CSL's `UnitGenerators`. To achieve the scoping and IO behaviors associated with CSL's Observer system, Pink users can use pass-through audio functions inserted into the graph to add additional behavior. In OO design, the use of pass-through functions would be equivalent to the use of Decorators [70] rather than Observers to add additional behavior. Looking at systems more broadly, Music-N systems (e.g., Music V, CLM, Csound, SuperCollider 3) share concepts of instruments (called Synths in SC3), unit generators, and events (i.e. notes). Instruments are made up of unit generators and can be scheduled for activation through events. Unit generators can neither operate on their own outside of instruments nor can they be scheduled. Within Music-N instruments, unit generators are loosely coupled: connections between unit generators are not made directly but rather

are made through shared variable memory. Like unit generators, instruments that communicate values to each other do so indirectly by writing to and reading from shared memory (e.g., Csound global variables, SC3 bus channels). Finally, Music-N event systems are generally limited to expressing a fixed set of operations and only permit certain kinds of values as event arguments.

Pink was inspired by Music-N systems to include an event system as a fundamental part of its design. However, Pink's event system is generic in terms of both its event functions and argument types. Pink also discards the separation of instruments and unit generators and unifies the two in its concept of audio functions. Audio functions can be composed together from other audio functions to achieve the same features as instruments in Music-N systems. Any audio function, whether it is a Composite or a standalone function, may be scheduled and activated within Pink's signal processing graph. Finally, audio function connections are made by using references to other audio functions directly rather than through intermediary variables.

The concept and implementation of unit generators from Music-N were further extended by numerous software synthesis systems [140]. The concept of a reusable signal processing object remains at the heart of what defines unit generators, but requirements for real-time applications — particularly dynamic connections and post-initialisation messaging — have largely driven authors to extend the core definitions of unit generators within their systems. For example, in graphical patching systems in the Max family (e.g., Max [147], Pure Data (PD) [148]), the processing object not only executes signal processing synchronously with the engine but also allows connections to be made dynamically and accepts asynchronously posted messages to affect state and behavior outside of the signal graph. The dynamic modification

of graphs and event handling features of Max-style objects is a departure from the implementation of unit generators in Music-N systems, where signal processing graphs are assembled statically within the confines of an instrument and generators may only react to their signal inputs. Dynamic connection capabilities appear in object models such as the 4MPS model (CSL, CLAM) through its concepts of Ports and Controls and the post-initialisation messaging appears as additional methods on processing objects outside of the primary signal processing method.

Many OO DSP systems (e.g., CSL, CLAM, Max, PD, JSyn) take the dynamic processing requirements and define them as a basic fundamental part of their unit generator classes. However, in looking at these systems, I personally found the code for implementing unit generators to be more verbose and harder to understand than what I was familiar with in a Music-N system like Csound. Also, from my experiences in wrapping Csound opcodes using Adapter [70] objects for use in Aura [204] — which features a dynamic connection model — I realised that a static connection design could be made to operate dynamically. These observations lead to my decision to design Pink with as simple a model as possible that could both satisfy the requirements for creating signal processing graphs and supporting building higher-level object models. As a result, Pink, by default, takes a more Music-N approach to unit generator design where audio functions are assembled into graphs at initialisation time and may only react to their inputs. Other unit generator features, such as dynamic connection capabilities, are expected to be developed as higher-level features built upon Pink's basic model.

Systems developed to operate within general-purpose programing languages (e.g., CSL, CLAM, Synthesis Toolkit (STK) [46], Cmix [104], JSyn [40])

served as models for library-based system design. Observing how these systems were packaged as versioned libraries and how they integrated into users' works, as well as studying the degree of openness of each system in terms of the reusability of their parts, influenced design decisions when developing Pink.

Systems that offer callback-based APIs, found mostly in hardware interaction libraries (e.g., RtAudio [157], RtMidi [158], PortAudio [31], JACK [56]) provide users a way to register a callback function and data pointer. The callback function is later executed by the system, passing the data pointer to the callback for processing. Callback-based APIs operate with state data and behavior as separate entities and are most commonly found in systems programmed in languages, such as C, that are not object-oriented. Pink uses a similar pull-based processing model to callback-based systems but works with stateful functions (i.e. closures) instead of separate state data and functions. (Object-oriented systems implementing this model would take a similar approach but use objects instead of closures.)

Multimedia frameworks provide support for not only audio programming but also other media-related development needs (e.g., graphical user interfaces, video and sensor data processing). JUCE [153] and OpenFrameworks [130] are both cross-platform, object-oriented C++ multimedia frameworks. JUCE primarily targets audio application and plugin development while OpenFrameworks targets building multimedia applications. Gibber [152] is a browser-based live coding environment and multimedia framework written in Javascript. In contrast to these frameworks, Pink's design focuses on the audio programming domain alone and it assumes users requiring non-audio programming features will employ other libraries to fulfill their needs. Also,

while there are no plans to make Pink into a framework itself, Pink could serve as a part of a larger framework in the same way that Gibberish.js is an audio-specific library that is a part of the Gibber framework.

6.4.2 Overview of Pink's Design

The core of Pink is separated into the following parts: *engine*, *signals*, *nodes*, *audio functions*, *control functions*, *context*, *events*, and *utility code*. The engine is used to process events and run audio and control function graphs. Events are time-tagged objects that are used to call functions with supplied argument values; events are most often used to activate new audio functions and dynamically extend the audio graph, but they are also generic and may be used for whatever purpose the user desires. Audio and control graphs are made up of zero-argument functions that are composed together at initialisation time; the function graphs in Pink follow a pull-model where the engine calls the graphs and processes their results. Signals are data structures (e.g., arrays of 64-bit floating point numbers) that are returned by functions and further processed by other functions within the graph. Nodes are data structures used together with processing functions to create dynamically-modifiable points within audio and control function graphs where sub-graphs can be attached; Nodes behave similarly to *Composite* [70] objects in object-oriented languages and can serve as dynamically-connectable, fan-in *ports* for audio functions. (Further discussion on static and dynamic graph connections and comparison to ports in other systems is provided in Section 6.4.3.) Context provides audio and control functions information about the processing context, such as the sample rate, buffer size, or current buffer number. Finally, utility code is provided to help users implement their code to work with the engine.

The design of Pink aims to provide not only all of the low-level parts necessary to create a music system but also a working system for the user to use. These high-level functions, found in the `pink.simple` namespace, provide a new user a default setup that works out of the box for music-making. However, by also exposing the lower-level functions that make up all of the parts of the engine, a user can create and customise engines, adding and removing parts as desired.

The goal for Pink is to support development of both ahead-of-time and real-time systems. The design is made to be extensible by the user at all levels. Ultimately, Pink should not only serve as a system to explore music composition, performance, and audio processing, but also act as a framework for exploring music system design.

6.4.3 Implementation

The following will discuss the implementation of Pink. Each part will be discussed on its own, and an example usage of the full system, together with the Score library, will be shown in Section 6.6. I will then conclude with a discussion of the system's design for extensibility and its impact on users.

Engine

Pink's engine provides the basic core of a music system. The implementation of the engine is found in the `pink.engine` namespace. The engine runs audio and control function graphs as well as processes events. The engine's design supports both real-time and ahead-of-time use and is thus appropriate for creating real-time music systems as well as pre-composed works.

Pink's engine employs a single-threaded design, where one thread drives all of the processing in the system. The engine state may be read at any time by other threads, but mutations to the engine state may only be done by the engine's thread. Users wanting to mutate the state of the engine (i.e., add a new event, add a new audio function, etc.) can use utility functions that schedule messages to various message inboxes in the engine. These messages in turn will be processed by the engine thread. By using message passing in conjunction with Clojure's atomic operations, the engine's design is completely lock-free and wait-free.

Figure 6.5 shows the four main parts of Pink engine loop: processing of scheduled events, the pre-audio control graph, the audio graph, and the post-audio control graphs. The engine delegates event scheduling and processing to the `EventList` and `event-list-processor` (described in Section 6.4.3).The three graphs are instances of Nodes (described in Section 6.4.3), and their arrangement allows for doing control processing before and after the audio graph. These control graphs may be used to retrieve and write values before and after the audio graph is processed, as might be used for audio, network, or user interface I/O. The use of multiple graphs allows for flexibility in controlling the order of processing.

Pink's engine executes but does not process the results of the control function graphs. The results of control functions are processed by their parent functions and most often will be processed by a Node control processing function to determine if the function is complete and should be removed from the graph (further details about Node processing is given in Section 6.4.3).

Pink's engine both executes and processes the results of the audio function graph. The audio function graph follows a pull processing model where results

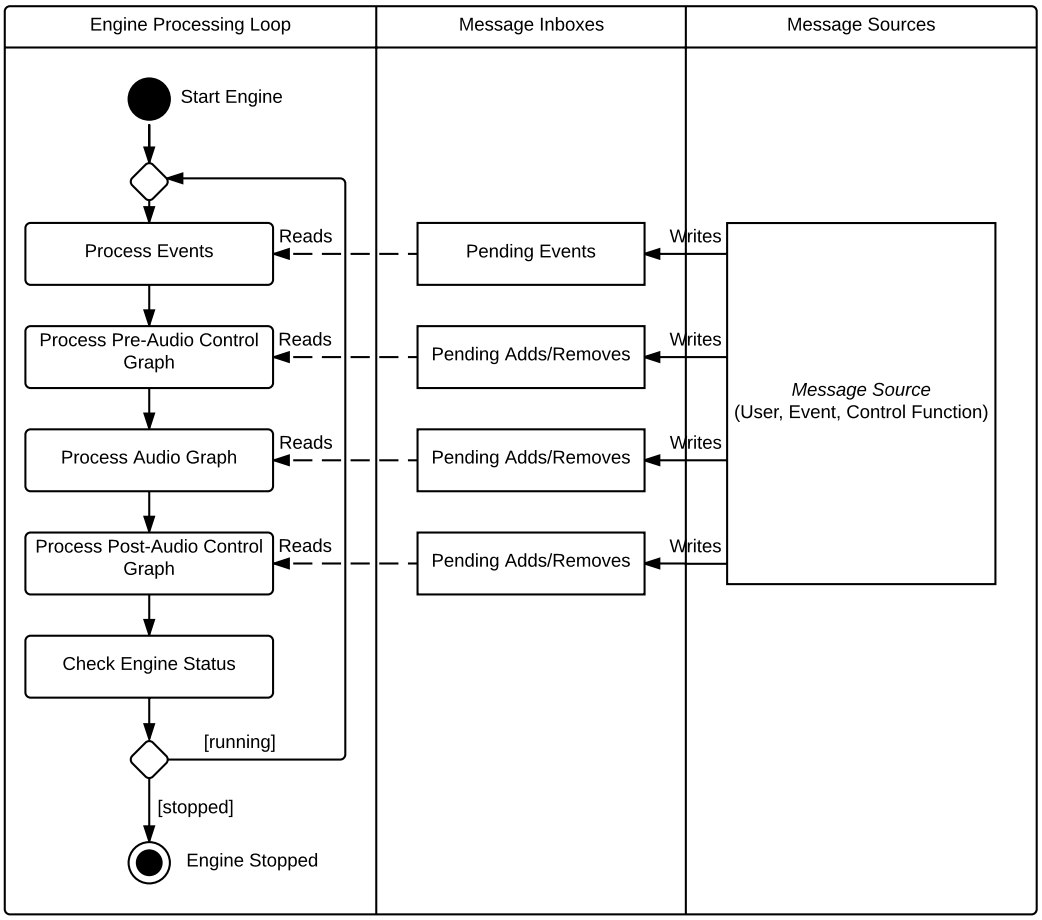

Figure 6.5: Pink Engine Architecture

of signal processing nodes (i.e., audio functions) in the graph are returned to their callers for further processing. Pink calls the top-level audio function (a Node audio processing function) that performs a depth-first traversal of the audio function graph. The results of each audio function is processed by its caller and ultimately returned to the engine. The engine takes the audio graph results and uses the JavaSound API to push audio samples either to the soundcard or to disk.

The code that handles each processing step will first handle incoming messages prior to main processing for that step. These messages are generally used to modify the target graph by adding or removing functions or events. The source of these messages may be from anywhere, including not only the user but also from events and control functions. This means that events may create events, control functions may create control functions, and so on.

The general use of an engine involves creating, starting, stopping, and resetting an engine. Other operations involve scheduling events as well as adding and removing functions to the audio and control graphs. Beyond this, all other operations for the music system are handled by subsystems in Pink.

Pink provides two separate processing functions for the engine. `engine-start` takes in an engine and creates a thread to run the engine in real-time. It handles writing audio output to the sound card. `engine->disk` takes in an engine and will run the engine to completion, defined as when all function graphs are complete and empty, as well as when all events have been processed. This functions handles writing audio output to disk.

The same engine and architecture is used whether the system is run to disk or real-time. If the user would like to modify how the engine processes, they are free to reuse the engine state design and develop their own processing function.

Signals

Signals are modeled in Pink using simple data structures. For example, an mono audio signal would be implemented by using an array of double precision (i.e. 64-bit) floating point numbers and a stereo signal would be implemented

by using a two-dimensions array. Audio functions in Pink generate signals as well as consume signals from other audio functions.

Currently, Pink's signal processing functions support only audio signal processing of single- or multi-channel audio signals. Other signal functions may be developed in the future to support scalar floating point numbers (similar to Csound's control rate variables), FFT analysis frames, and other signal types. Processing functions for new signals would follow the same pattern used for audio functions (discussed in Section 6.4.3) with the exception of returning the new signal type instead of audio signals.

Nodes

Nodes are points in Pink's function graphs where other functions may be dynamically attached. They are used similarly to Composite objects in OO programming where they function both as collections of audio or control functions as well are themselves audio and control functions. Nodes may also be used as Proxy[70] functions if used with a single child function. When used as an input to an audio function, Nodes act like *Ports* for the audio function that can both fan-in signals from other audio functions as well as provide dynamic connection capabilities.

Implementation The implementation of Nodes is found in the `pink.node` namespace. Nodes operate by iterating over a list of active function instances and calling them for processing. Nodes also contain a list for pending adds and removes of functions to and from the active list, a running status, and the number of audio channels. Note that the the number of audio channels is used only with audio node processing and is ignored for control node processing.

Nodes are implemented in two parts. The *state data* contains all of the current state. Listing 6.2 shows the `create-node` function that creates the state data for a Node. Nodes are represented using plain Clojure map data structures.

```
(defn create-node
  [& { :keys [channels]
       :or {channels *nchnls*}
       }]
  { :funcs (atom [])
    :pending-adds (atom [])
    :pending-removes (atom [])
    :status (atom nil)
    :channels channels
   })
```

Listing 6.2: Code for create-node

The second part of Nodes is their processing function. `pink.nodes` provides two processing function generators, `node-processor` and `control-node-processor`. Each of these functions take in a Node and returns a function that is used for processing of the node for the duration of one block of audio.

The two processing functions have similar processing code but differ both in how they are used and how results are returned. For `node-processor`, the returned function follows the Pink audio function convention. When this function is executed, it will first process any pending add and remove messages to update the list of active audio functions contained in the `:funcs` list. Next, each function in the `:funcs` list is executed. If a function returns an audio buffer, the results are summed into a results buffer. If a function returns `nil`, the function is considered done and removed from the active list.

After processing child audio functions, the `node-processor` function returns the summed audio buffer.

For `control-node-processor`, processing follows the same model as `node-processor`, but the returned processing function follows the Pink control function convention. The function will begin by processing all messages and handle updates to the active `:func` list. Next, functions are executed. Pink control functions are expected to return `true` or `false` as their results, which signals whether they are done processing. The `control-node-processor` will remove any function that signals it is done from its active `:func` list. The `control-node-processor` function will also return `true` or `false` to signal if it itself is done.

As the functions returned from `node-processor` and `control-node--processor` themselves conform to Pink's audio and control function conventions, they can be used as inputs to other functions, including other Node processing functions.

Active Functions Updates to a Node's active function list are not done directly by users. Instead, when adding or removing a new audio or control function, the function is added to the appropriate `:pending-adds` or `:pending-removes` list for the Node. In turn, the audio or control node processing function will handle inserting and removing those functions. This guarantees that the state of the `:func` list is only mutated at audio block boundaries by the processing function and not while processing occurs.

Users may add functions to the `:pending-adds` or `:pending-removes` lists by using the `node-add-func` and `node-remove-func` utility functions. These take in a Node and a function and atomically add the function to the appropriate pending list. These utility functions may be used by the user

directly while live-coding or indirectly through events. The use of events with these functions to dynamically add new audio functions to the graph effectively simulates the concept of *notes* found in Music-N systems.

Summary Pink's Node system is defined using a single data structure and a set of related functions for working with instances of that structure. The system provides safe functions for mutating the active function list through the use of atomically-protected pending lists. The pending lists are then safely processed at audio-block boundaries by Node processing functions. Two function generators are provided that produce processing functions that conform to Pink's audio and control function conventions and operate upon the state held within the Node data structure. Users can use the provided Node functions to create dynamically modifiable points within a function graph as well as build upon the Node data structure to implement their own custom processing functions.

Audio Functions (Unit Generators)

Audio Functions in Pink are roughly equivalent to the classic concept of *unit generators* and are composable units of audio signal processing. The system of unit generators has had a great influence on computer music software since its introduction. The design and implementation of unit generators differs between systems. Beyond their primary signal processing methods, unit generators may have additional features such as the ability to accept messages (e.g., PD objects), exposure of properties for external modification (e.g., ChucK UGens), and re-initialisation (e.g., Csound opcodes). The design of audio functions in Pink is largely focused on the the signal processing

method alone, relying upon standard functional programming techniques to implement the features found in other systems.

Life Cycle of Unit Generators The following section discusses the basic life cycle of a unit generator. This includes *allocation*, *initialisation*, *performance*, and *deallocation*. These properties of unit generators are common to all unit generator implementations and has similarities to the life cycle of objects in object-oriented (OO) languages. Some OO languages (e.g., Java) tie together allocation and initialisation through the concept of a *constructor*, but these concepts will be discussed separately here as some music systems (e.g., Csound) operate with the two phases separately. This analysis of unit generators was also presented in [204].

Allocation *Allocation* is the process of acquiring the memory required to represent a unit generator. This aspect is the beginning of a unit generator's life cycle. Memory allocation is generally carefully handled in audio systems. Approaches include implementing custom real-time memory allocators (e.g., SuperCollider 3) and/or garbage collectors (e.g., Aura), as well as reuse via memory resource pooling (e.g., Csound). The extra care regarding memory allocation is done to prevent breakups in real-time audio, where the time required to allocate or free memory may interfere with the delivery of audio samples to the sound card.

For Pink, memory allocation is done using the standard mechanisms found in the Java Virtual Machine. The JVM uses a garbage collector (GC) and pre-allocates the heap memory for an application at the start of the system. Object allocations are done using sub-regions of memory from the heap. This

makes memory allocation very fast compared to `malloc` as the memory is already allocated.

While this provides fast allocation, it presents two drawbacks. Firstly, the garbage collector thread can cause a full GC pause of all other JVM threads to take care of GC tasks (also known as *stop-the-world* time). The exact performance characteristics depends upon the GC algorithm chosen when starting the JVM.[6] Care must be taken to limit both the *frequency* and *duration* of GC pauses so as not to interfere with the audio thread.

The audio thread is a natively-managed thread that is not affected by JVM GC operations. The audio thread is setup to read from a ring buffer; the Java audio system is setup to write to that ring buffer. There are two buffers so the system is double-buffered.

The frequency of the GC is proportional to the rate at which garbage is generated. Pink's unit generators and engine reuse memory as much as possible and do not allocate memory after initialisation. The worst case scenario would be if the GC pause frequency was greater than the buffer processing frequency such that more than one GC pause could occur in the processing of one buffer.

With Pink and the default buffer size of 256 and sample rate of 44100 Hz, the duration of the buffer equates to 5.8 ms of time. The target then would be to reduce GC pauses to less than once every 5.8 ms. Informally, turning on GC diagnostic information (using the `-XX:+PrintGCTimeStamps -XX:+PrintGCApplicationStoppedTime` flags when starting the JVM), GC

[6] At the time of this writing, the G1 algorithm [59] is the target algorithm for Pink.

pauses occurred every 1.0-3.0 seconds, well safe from having multiple pauses per buffer.[7]

The duration of the GC pause limits the amount of work that can safely be done in the time to calculate one buffer. The worst case scenario is that it takes longer than one buffer's worth of time to generate and deliver the buffer to the audio system. To safely deliver the buffer in time, the buffer must be generated and delivered in less time than the duration of the buffer, minus the duration of the GC pause. Informally, GC stop-the-world times were viewed in the from 1.1 to 3.3 ms. With a buffer size of 256 size, the worst case scenario for the observed GC pause times would require generating each buffer within 2.5 ms to operate without risk of having audio breakups.

Here, the user has a few options. They can limit their CPU usage in their project, they can try modifying the JVM settings to lower the stop-the-world pause time, or they can increase the buffer size to minimise the cost of the GC pause.

Secondly, if an application tries to allocate more memory than is available in the heap, the application will start throwing `OutOfMemoryExceptions`. The default maximum memory used for heaps depends upon the JVM used (client or server) as well as the platform.[8] For synthesised sound, the default heap size may be enough but it may be limited for sample-based audio processing. To mitigate this, one can set a larger maximum heap size when starting the JVM using the `-Xmx` flag. For my own Pink-based projects, I default to using 512 megabytes and adjust according to the project.

[7]This was observed on a Macbook Pro 13-inch early-2011, 2.7ghz Intel Core i7-2620m CPU machine.

[8]For more information on heap size defaults, see [132]

Note that all audio systems that use the JVM receive the benefits of fast object allocation and issues due to GC pauses. This would include programs like Beads [38] and JSyn [40].[9] As one can not implement his or her own memory allocator in Java, Pink then optimises what it can for GC frequency and follows the common practice for how to handle GC pause times.

Initialisation An initialisation pass for a Unit Generator is done to configure state variables and to calculate constants that will be used during each performance call. The initialisation pass may be done in the constructor for a class in an object-based system (e.g., SC3) or may be an explicitly called function (e.g., Csound, PD). Additional memory may be allocated at this time and must be handled with the same care as the initial unit generator allocation. This is especially important if the initialisation is done while on the main audio thread.

Performance At performance, a unit generator is responsible for generating x number of samples for a given time n. The number of samples and how time is measured depends upon the system. For example, in systems with a single-sample processing model, time would be measured in the number of samples since the start of the engine, and the number of samples to produce would be 1. In systems employing a block-based processing model, time would be measured in the number of blocks since the start of the engine, and the number of samples to produce may be something like 64 (the exact number depends on how the user or developer has configured the system).

[9] This refers to newer, pure-Java versions of JSyn, which differs from earlier versions that used a natively programmed C synthesis engine.

For block-based systems, a unit generator will largely follow the following pattern:

1. Read state values from the previous pass into local variables.

2. Process and generate samples up to the block-size, using a loop. Processing here uses stack and local variables to improve performance over the duration of the loop calculation.

3. When the block-size number of samples has been generated, the current local state is then written into to the unit generator's state. These values are used the next time the unit generator is called for processing.

4. The unit generator may then write the generated values to some location in memory, or return them to the caller, depending on the design of the system.

Note that most state values stored in a unit generator's memory that are loaded into and saved from local variables are used strictly to preserve the state of the generator's computation between calls.

Deallocation When a unit generator completes processing, its memory is available for deallocation. This may happen when an audio sub-graph is expired or done (i.e., when a note ends) but may also be delayed until a non-critical time (i.e., when a piece is finished rendering). What happens at deallocation time is dependent on what memory system is implemented. It could mean a call to `free()`, a marking that memory is garbage and free for collection, or a decrement of a reference count. For Pink, the system follows JVM standard practice and the unit generator will be garbage collected when it no longer has any references to it from live objects.

Pink Unit Generators In Pink, unit generators are split up into an outer-function that returns an inner-function. The outer function will do initialisation, calculating and storing values that the inner-function will enclose over and have access to while processing. The inner-function is expected to be a function of zero arguments that returns a signal value. This higher-order programming style provides the same initialisation and performance time separation as found in other systems based on unit generators, such as Csound and SuperCollider.

```
(defn some-ugen
  [arg0 arg1 arg2]
  (let [x (some-calculation arg1)
        out (create-buffer)]
    (fn []
      (do-processing-loop x arg0 arg1 out)
      out)))
```

Listing 6.3: Basic code shape of Pink unit generator

Listing 6.3 shows the basic shape of a Pink unit generator in Clojure code. In this example, the `some-ugen` function returns an anonymous audio function with zero arguments. The outer function is where allocation and initialisation is done (shown in the let-block), and the returned function is used for performance. The returned anonymous function is a *lexical closure* that will close over both the arguments to the outer function as well as the values within the let-binding. At performance time, the audio function will return a *signal* – such as a mono or multi-channel audio buffer – or `nil`. Returning `nil` signifies that the audio function is done processing.

An audio function in Pink must be sure to check whether any audio function it depends on is done (i.e., returns `nil`). If a `nil` is found, the audio

function must short-circuit and return `nil` itself. When an audio function is used as a child function of a Node, the Node's processing function will check if an audio function is done and, if so, remove that function from its active list to prevent further processing.

Most unit generators yield stateful functions with mutable data. State is generally used only for storing and restoring values that are used in the processing loop and are scoped only to the function which closes over it. This state should therefore not be allowed to escape its scope and thus be shared outside of the function. Also, the only code that should be allowed to write to the unit generators state is the unit generator itself. Following these rules provides for safe use of mutable state by the function. (This follows the same logic for safety as used in Clojure's transient data structures [91].)

The basic pattern for Pink unit generators operates similarly to how *constructors* and *factory methods* [70] function in object-oriented programming. Constructors provide programmers a way to allocate, initialise, and return an object of an exact type associated with the constructor. Factory methods function similarly to constructors but are not bound to an exact type: they can programmatically choose amongst subclasses to return to the caller. Hoyte calls the function pattern used for Pink's audio functions as "lambda over let over lambda" and notes the similarities to objects and classes in [93].

```
public interface AudioFunction {
  // can return double[] or double[][]
  public Object perform();
}

public class MyAudioFunction implements AudioFunction {
  // Constructor
```

```
  public MyAudioFunction(double arg0, double arg1) {
    // Perform initialisation based on constructor arguments
  }

  public Object perform() {
    // processing code
    ...
    return result;
  }
}

public class AFnFactory {
  // factory method
  public static AudioFunction createAFn(boolean useMyAFn) {
    if(useMyAFn) {
      return new MyAudioFunction(0.0, 2.0);
    }
    return new SomeOtherAFn();
  }
}
```

Listing 6.4: Constructors and Factory Methods

Listing 6.4 provides an object-oriented interpretation of Pink's audio functions using the Java programming language. The `AudioFunction` interface represents the polymorphic type that the audio functions returned from a Pink unit generator function implement. The performance code within Pink audio functions can be interpreted as the implementation of the `perform()` interface method. The `MyAudioFunction` class and its constructor maps closely to Pink audio functions that return only a single kind of function. The static `createAFn()` factory method of the `AFnFactory` class maps closely to

Pink unit generators that process their input arguments and choose amongst various function implementations to return to the caller.

Function Graphs and Connections Audio functions are designed such that all of their dependencies for their calculations are known at initialisation time. When one audio function is passed as an argument to the initialisation function for a second audio function, the second audio function keeps a reference to the first to call for signal generation. The function reference is itself the connection made between two audio functions and represents the edge between two nodes in a directed acyclic graph. The function graph is thus formed at initialisation time through the passing of functions as arguments.

```
;; Define a Port as a mutable atom with a nil audio function
(def port (atom nil))

;; Proxy audio function for Ports
(defn port-processor [port]
  (fn []
    (let [afn @port]
      (if afn
        (afn)
        EMPTY-BUFFER)) ;; global empty buffer

;; set port to use some-audio-function
(reset! port some-audio-function)
```

Listing 6.5: Implementing a 4MPS-style Port using audio functions

For cases where an exact dependency is not known at initialisation time, such as in the case where audio functions are dynamically connected at runtime,

proxy functions (equivalent to Proxy objects [70]) may be used. Listing 6.5 shows a simple implementation of a proxy function called `port-processor`. The proxy function takes in a Clojure `atom` that is used as a mutable container for audio functions. The function returned from `port-processor` is an audio function that checks if an audio function has been set in the `port` container and either executes the function and returns its results or returns a default `empty-buffer` value. The `port` may then be modified at runtime to support making dynamic connections between audio functions.

```
;; Define a Control as a mutable atom with 0 value
(def control (atom 0))

;; Control audio function
(defn control-processor [control]
  (let [out (create-buffer)]
    (fn []
      (Arrays/fill out @control)
      out)))

;; set control to some value
(reset! control 1.0)
```

Listing 6.6: Implementing a 4MPS-style Control using audio functions

For cases where an asynchronous connection is desired, a function and data pattern similar to proxy functions can be used where a mutable data container is exposed for writing values and an audio function is used for reading values. Listing 6.6 shows an `atom` called `control` that is used to hold a double floating point value. The `control-processor` function takes in an `atom` and returns an audio function that, at runtime, will fill the `out` array with the current value of `control`. Users may modify the value of `control`

asynchronously from any other thread (e.g., GUI, MIDI, OSC) and the most current value will be read by the `control-processor` when it generates its signal value.

Pink's audio function model is a lower-level model than the 4MPS object model found in CSL and CLAM. Pink does not model concepts of Ports or Controls as a base part of the signal graph design. However, Listing 6.5 and Listing 6.6 demonstrate how those concepts could be implemented on top of Pink's model to add dynamic synchronous and asynchronous connection capabilities to a Pink signal processing graph. A full implementation of a higher-level Processing Object model could also be developed on top of audio functions that includes all of the features from 4MPS's model (i.e. metadata lookup for what Ports and Controls are available, support for dynamic connections by default). The lower-level design was chosen for Pink to simplify coding responsibilities for audio function authors and to allow users the flexibility to explore their own higher-level designs. A higher-level system like Processing Objects may be offered as an optional part of Pink in the future, but it remains outside of the goals of Pink's design at this time.

```
(sum (sine2 440)
     (sine2 880))
```

Listing 6.7: Example usage of audio functions

Listing 6.7 shows a simple use of audio functions. The `sum` unit generator is called given two `sine2` unit generators as arguments, one with frequency 440Hz, the other with 880Hz. Each call to `sine2` returns an audio function. When `sum` is called, it uses the two `sine2` audio functions and itself returns an audio function. When the returned function goes to calculate values, it will call the audio functions returned from `sine2`, check if their results are

`nil` and, if so, return `nil` itself. If the audio functions return audio signal values, the `sum` audio function will mix the values into its out buffer and return that as the result.

Instruments and Effects Pink's engine processing model is built up upon directed acyclic graphs of functions. The graphs are generally organised into stable and dynamic subgraphs organised by Nodes. Stable parts of the graph would be used for always-on processing, such as for mixing and effects, and dynamic subgraphs would be used for temporary audio processing, such as for instrument notes.

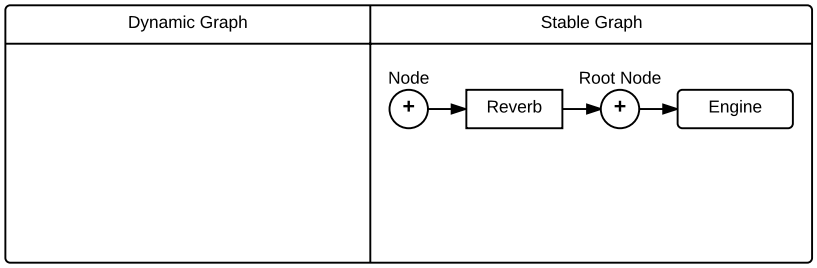

Figure 6.6: Example Pink Audio Graph: Stable

Figure 6.6 shows a simple stable graph. The root Node is the starting point of the graph and is used by the engine to pull audio from the graph. Connected to the root Node is a reverb audio function that itself has a Node as its source. Using a Node as the source for the reverb allows audio functions to be dynamically attached to that part of the graph.

Figure 6.7 shows the graph after a new note instance (i.e., audio function) is created. The *Audio Function* (AFn) box here represents a Composite audio function made up of other audio functions and used to represent an instrument in the classic Music-N sense. The AFn is shared and attached to

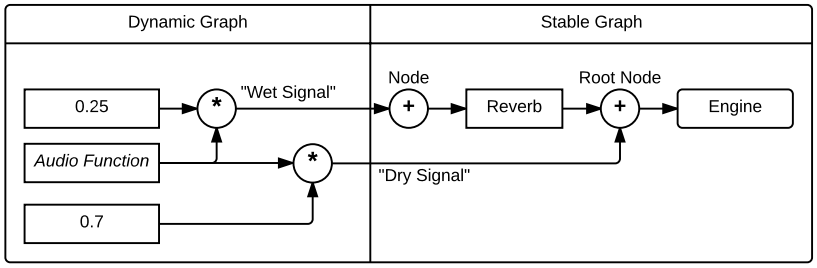

Figure 6.7: Example Pink Audio Graph: Dynamic

the graph in two ways. Firstly, the AFn is used as the source to a multiplier audio function, and that AFn is in turn attached to the root Node. Secondly, the AFn is used as the source to a second multiplier audio function, and that AFn is in turn attached to the reverb Node.

When processing occurs for Figure 6.7, the audio results for the Audio Function box would be routed through both the reverb and root Node parts of the graph. The use of the multiplier AFNs then controls the amount of *wet* and *dry* signal being used in the graph. When the Audio Function is complete – such as would happen if the duration of the note is complete or the note is turned off – it would return a `nil` value. That nil would cause the multiplier AFns to short-circuit and themselves return `nil`. From there, the `nil` would cause both the reverb's source Node and the root Node to remove the multiplier AFNs from their active `:funcs` list.

The above shows that the concept of effects and instruments can be implemented using audio functions and Nodes. Ports and Controls may be added to Pink-style effects and instruments by implementing them as audio functions as shown in Listing 6.5 and Listing 6.6. Adding these features to an audio function graph would then make the instruments and effects amenable for manipulation by GUI, MIDI, or OSC. A higher-level instrument abstraction that provides metadata and access to Ports and Controls, such

as the one provided in CSL, is not provided at this time. However, future versions of Pink may offer such an abstraction as an optional feature.

Generator Macro Because Unit Generator code has a number of common requirements, the `generator` macro, found in the `pink.util` namespace, was developed to ease writing of unit generators. The generator macro uses four parts and has the general code shape shown in Listing 6.8.

```
(generator
  [cur-state state] ;; 1. State pairs
  [sig sig-fn]      ;; 2. Signal in sig function pairs
                    ;; 3. Calculation for current sample
  (let [some-value (calculation cur-state sig)]
    (aset out int-index some-value)
    (recur (unchecked-inc indx) (update cur-state)))
  (yield out))      ;; 4. Value to yield
```

Listing 6.8: `generator` macro basic code shape

The above reads as "For each value of the current state and each signal value returned from sig-fn's, process each sample in a loop until `*buffer-size*`, and yield out as a result." The expanded macro would create a function that will:

1. Initialize the cur-state to the provided initial state.

2. Restore cur-state from its last value in the processing loop (i.e., cur-state).

3. Call sig-fn and assign the value to a temporary value. If the value is `nil`, immediately short-circuit and return `nil`.

4. In a loop, using the values from the state and signals sections (sections 1 and 2), call the section 3 body until `indx` is >= `*buffer-size*`.

5. When `indx` is >= `*buffer-size*`, save the local loop values to cur-state, then return the out value.

The macro code in Listing 6.8 would macroexpand to the code shown in Listing 6.9.

```
(let* [state1890 (double-array 1 state)
       buffer-size1891 (clojure.core/long
   pink.config/*buffer-size*)]
   (fn*
     ([]
     (let* [buffer1889 (sig-fn)]
       (if buffer1889
         (do
           (loop* [indx 0
                   cur-state (clojure.core/aget state1890 0)]
             (if (clojure.core/< indx buffer-size1891)
               (let* [int-indx (clojure.core/int indx)
                      sig (aget buffer1 889 indx)]
                 (let* [some-value
                        (calculation cur-state sig)]
                   (aset out int-index some-value)
                   (recur (unchecked-inc indx)
                          (update cur-state))))
               (do
                  (clojure.core/aset state1890 0 cur-state)
                  out)))))))))
```

Listing 6.9: `generator` macro basic code expanded

Note that the macro does synthesise some common values, such as `indx` and `int-indx`, which refers to the current index when iterating through the processing loop. It is recommended to use the generator macro where possible, as doing so can lead to code whose handling of state makes it easier to read as well as safer to write. However, the generator macro does not currently work for all unit generator use cases, as some unit generators require more logic than just resuming from the previous state. In those cases, one can simply write audio functions directly.

Example: Phasor The following is the source for the Phasor unit generator:

```
(defn phasor
  "Phasor with fixed frequency and starting phase"
  [^double freq ^double phase]
  (let [phase-incr ^double (/ freq (double *sr*))
        out ^doubles (create-buffer)]
    (generator
      [cur-phase phase]
      []
      (do
        (aset out int-indx cur-phase)
        (recur (unchecked-inc indx)
               (rem (+ phase-incr cur-phase) 1.0)))
      (yield out))))
```

The `phasor` function will, given a frequency and starting phase, return a function that will generate audio signals from 0.0 to 1.0 over and over again, repeating at the given frequency and offset by the given initial phase. Note, `phasor` does not use the signals section of the macro (section 2). Use

of section 1 and 2 of the generator macro is optional. Examples of further generator macro use can be found in the Pink codebase.

Shared Audio Functions When using audio functions, situations arise where the user may want to use the result of one audio function as the input to multiple other audio functions. In general for functional programming, a let-block is used to assign the result of an expression to a local variable, and the variable is then further used as arguments to other functions. Using let-blocks with audio functions addresses sharing of the audio function, but it does not take into account that audio functions are often stateful and should only calculate values once per unit of time within the engine.

```
(defn shared
  "Decorates an audio function with another AFn that ensures
  calculations are done only once per block. Calculates,
  saves, and returns an audio buffer the first time called
  per buffer number. Subsequent calls during the same buffer
  number will return the saved value."
  [afn]
  (let [my-buf-num (long-array 1 -1)
        buffer (atom nil) ]
    (fn []
      (let [cur-buf (long *current-buffer-num*)]
        (if (not== (getl my-buf-num) cur-buf )
          (do
            (aset my-buf-num 0 cur-buf)
            (reset! buffer (afn)))
          @buffer)))))

(defn- decorate-shared
  "Utility function for let-s macro to wrap AFn symbols as
```

```
  shared."
  [args]
  (reduce
      (fn [a [b c]]
        (conj (conj a b) (list `shared c)))
        []
      (partition 2 args)))


(defmacro let-s
  "Macro for defining let-like bindings. Wraps AFn functions
  with shared and allows using audio functions as inputs to
  multiple other audio functions."
  [bindings & body]
  `(let ~(decorate-shared bindings)
     ~@body))
```

Listing 6.10: Code for `let-s` and `shared`

Pink provides the `shared` audio function decorator and the `let-s` macro – short for "let shared" – to address sharing of audio functions within a graph. The `shared` function takes in a source audio function as an argument and returns its own audio function. When this audio function is executed, it will check the engine's current buffer number and compare it to the last stored buffer number. If they differ, the source audio function is called to process one buffer of audio. The returned value is cached then returned to the caller. If the buffer numbers match, it means that the function is being called again within the same graph. In this case, the cached value from the source audio function is returned. Users can use the `shared` function to decorate any audio function and make it safe for use by multiple other audio functions. However, using `shared` with multiple functions may be "noisy" to write and read in the

code. Instead, the `let-s` macro can be used. This takes a set of bindings and a body of code in the same way that let-blocks do. In turn, the macro simply decorates all of the second arguments in the bindings vector with `shared`.

```
;; Use shared function directly
(let [amp-env (shared (adsr 0.1 0.1 0.95 0.5))]
  (mul amp-env
    (moogladder (blit-saw 440.0)
      (sum 2000 (mul 2000 amp-env)) 0.5)))

;; Use let-s macro
(let-s [amp-env (adsr 0.1 0.1 0.95 0.5)]
  (mul amp-env
    (moogladder (blit-saw 440.0)
      (sum 2000 (mul 2000 amp-env)) 0.5)))
```

Listing 6.11: Example use of `let-s` and `shared`

Listing 6.11 shows example usage of `shared` within a let-block and an equivalent code that uses `let-s`. In both of these examples, `amp-env` is assigned an `adsr` envelope audio function that has been decorated by `shared`. `amp-env` is then used both to multiply the signal to affect amplitude and to modulate the cutoff frequency of the `moogladder` filter.

Sharing audio functions in Pink requires users to use either the `shared` function or `let-s` macro. This design requires that users ensure that their functions are properly protected using one of the above options. An alternative design would move the burden of buffer time checking and caching from users to audio function authors. One benefit of the current design is that the audio function implementation code is focused only on the audio processing code, making it easier both to read and to write than code that also handles audio function sharing. Another benefit is that the design limits use of the `shared`

processing code only to the locations where it is essential, reducing both processing time for the graph and memory costs for caching. The drawback to this design is that it adds additional work for the user when designing their audio function graph. However, since Clojure requires using let-blocks to share values, the use of `let-s` seemed like a reasonable solution that satisfies requirements for both protection of shared audio function usage and idiomatic programming patterns for the language.

Using an alternative design that requires audio function authors to handle shared function usage complicates authoring, but simplifies using, of those functions. The costs of this design are, as noted above, extra processing and memory when functions are not shared as well as extra code unrelated to the core audio processing algorithm. The benefits to this design are that users do not have to worry about sharing and they can employ standard let-blocks to share audio functions. While the benefit of the alternative design were very appealing, my own concerns over the costs ultimately lead me to choose to implement Pink's current design.

Summary Pink uses functions that return stateful functions to implement the concept of unit generators. As functions can be used as arguments to other functions, the initialisation pass can be seen as a way to assemble static graphs of signal processing functions. Dynamic graph manipulation is possible by employing audio functions with separately mutable data. Using a functional programming approach allows for smaller functions to be composed into larger ones. This allows audio functions to act as both a unit generator and an instrument, and for new unit generators to be created by reusing other unit generators.

Processing Context

Unit generators in music systems process within a *context*. The context holds information such as the sampling rate, the block size, and other values. Unit generators and control functions in turn use this information to calculate coefficients and derive other values such as the current time.

Local and Global Contexts For Music-N-based systems, contexts are generally found in two forms: a global context that is shared engine-wide, and a local context that is shared by unit generators within a bounding body – such as Csound's instruments or SuperCollider 3's Synths. The variables for the contexts are generally found collected within a data structure. For example, in Csound's opcodes, global context information can be retrieved from the `CSOUND` data structure, which is passed in to every opcode's initialisation and performance functions. Additionally, local context values can be found through the opcode's `INSDS` data member, which is a back-pointer to a data structure shared by all opcode instances for an instrument. The `INSDS` struct holds information relevant only to that instrument instance such as if it is in a release state, the p-field values used to create that instrument instance, and more. It also contains local overrides of global contextual values, such as overriding `ksmps` so that a local block-size is used.

Pink and Dynamic Variables For Pink, processing context is handled using *dynamically-scoped variables*. In Clojure, a *var* is a "a mechanism to refer to a mutable storage location that can be dynamically rebound (to a new storage location) on a per-thread basis" [92]. By default, vars are *static*, meaning they have a single *root binding* to a value, or may be bound to nothing. The value of the var can be redefined but can not be dynamically

rebound to another value when static. Vars can be made dynamically-scoped if they are marked as `:dynamic`. When a var is dynamic, its value can be temporarily rebound on a per-thread basis, using the `binding` function. The value of the var is restored once a binding goes out of scope.

```
(def ^:dynamic *sr* 44100)
(def ^:dynamic *buffer-size* 64)
(def ^:dynamic *nchnls* 1)
(def ^:dynamic *current-buffer-num* 0)
(def ^:dynamic *duration* nil)
(def ^:dynamic *done* nil)
(def ^:dynamic *tempo* 60.0)
```

Listing 6.12: Processing context variables in `pink.config`

Listing 6.12 shows the processing context variables defined in the `pink.config` namespace, together with their root bindings. The values of these variables are rebound in various areas of the code base. For example, Listing 6.13 shows how Pink configures variables for a global context. When a Pink engine starts, it takes values it has configured and rebinds `*sr*`, `*buffer-size*`, and `*nchnls*`, then runs the engine. All code that is run within the engine thread will then see the values configured by the engine, rather than the default values. Once the code within the binding is complete, the value in each var is restored.

```
(binding [*sr* sr *buffer-size* buffer-size *nchnls* nchnls]
...
)
```

Listing 6.13: Rebinding of context variables in Pink's engine

Dynamically-scoped variables are used to implement contexts within Pink. As bindings can be nested, this allows Pink code to use these variables for both

global and local processing contexts. For example, the `*duration*` variable is `nil` by default and is not used by the engine. If it is set, certain audio functions – such as envelope generators – may use the value in `*duration*` to calculate values such as segment durations. If `*duration*` is not set, the envelope generator knows that it is running in a context where there is no duration and may adjust its calculations to work in a real-time setting. In that case, the envelope may then check if the `*done*` variable is set, which would contain a boolean flag used to signal done-ness (i.e., a MIDI keyboard key release). If a boolean flag is found, the audio function may check that value while processing and adjust computations accordingly.

Another example is the the `with-buffer-size` macro, found in the `pink.util` namespace. This macro allows for running a group of audio-functions with a different `*buffer-size*` than the current one. This is used similarly to how Csound's `setksmps` works, which changes the buffer size (`ksmps`) for the scope of the instrument or user-defined opcode. The `with-buffer-size` macro generates the code to rebind `*buffer-size*` locally, handle running the contained audio function enough times to fill the size of `*buffer-size*` outside of the macro, and restore the original `*buffer-size*`. For example, if the outer `*buffer-size*` is 64 and `with-buffer-size` has a value of 1, the audio functions within the `with-buffer-size` will run one sample at a time and they will be run 64 times to fill the size of the outer buffer.

Analysis Using dynamic variables for processing context provides unique benefits and drawbacks. Users can modify existing context variables to affect processing for local portions of code. Users may also extend the processing context by introducing their own dynamic variables. They can define and

use them within their audio and control function. Introducing new context variables then requires no modification to Pink's core engine code, making Pink's context extensible by the user.

The drawback to using dynamic variables is that debugging code may be difficult. Code that seems to work fine in one setting may not function well in another setting if the user forgets to set the dynamic variables correctly. This is a real problem, but I would argue that the benefits of extensible processing contexts outweigh the drawbacks of handling the context variables.

Pink's use of dynamic variables provides a means to define a processing context for audio and control functions. By using dynamic variables, subgraphs of functions can operate within a local processing context. As bindings can be nested, multiple layers of contexts can form, not just global and local contexts. Additionally, users can extend the processing context freely in their own code, without requiring change to the Pink engine.

Control Functions

Control functions in Pink have similar characteristics to audio functions but are generally used for non-audio purposes. The basic shape of a Pink control function is:

```
(defn control-func
  [arg0 arg1 arg2]
  (let [x (some-calculation arg1)]
    (fn []
      (do-processing x arg0 arg1 out)
      true)))
```

Like audio functions, control functions are attached to a Node as part of a graph. They are also run once per-block of audio processing, just like

audio functions. However, unlike audio functions, their return values are not used to pass signals to callers. Instead, the functions return `true` or `false`, to notify the caller if the function is done or not.

Control functions are called synchronously with the audio engine. Users can implement and add control functions to act similarly to coroutines or threads. Some uses include implementing algorithmic composition routines that fire new events (i.e., notes) at calculated times, implementing sample-accurate clocks for event generation, and processing user-interface values.

Like audio functions, control functions have access to the audio engine's current buffer number and sample rate. By using these values, control functions can keep track of elapsed engine time in number of buffers and samples. By tracking time and running synchronously with the engine, control functions can both run in real-time and ahead-of-time and produce identical results.

Pink's control functions have similarities to ChucK's *shreds* [187] and would be used for similar purposes. However, instead of being *sample-synchronous* as in ChucK, they are run *block-synchronous* as Pink's engine is block-based rather than sample-based. If desired, one can set Pink's block size to 1 to attain sample-synchronous behavior.

It is also noteworthy that Music-N systems often have a model for unit generators that behave more like Pink's control functions rather than audio functions. By this I mean that Music-N unit generators are often those that perform side effects and return a boolean result to signal success or failure. For example, in Csound, an opcode will process and write results to various memory locations then return either a success or error code. Due to these similarities, if one wanted to mimic a Music-N processing model and use

busses to read and write audio, one could use only control functions to achieve this.

```
(defn tempo-change
  "Change tempo atom value from current value to
  end-tempo over given seconds time."
  [tatom seconds end-tempo]
  (let [cur-buf (atom 0)
        end (/ (* seconds *sr*) *buffer-size*)
        incr (/ (- end-tempo @tatom) end)]
    (fn []
      (when (< @cur-buf end)
        (swap! cur-buf inc)
        (swap! tatom + incr)
        true))))
```

Listing 6.14: Example Control Function

Listing 6.14 shows an example control function. The function has three arguments: a tempo atom that holds the current tempo, the number of seconds to change over time, and the target end-tempo. The values cur-buf, end, and incr are first initialised, then an anonymous function is returned. The returned function will be added to one of Pink's control graph Nodes for processing. When the function is executed, it will first check if the current buffer is less than the end buffer number. If not, the `when` function will return `nil`, which is equivalent to `false` for the calling code. If it is less than the end buffer number, it will update the tempo atom with the next value towards the target tempo then return `true`. By returning `true`, the control function is signaling that it is to continue processing.

Control functions enable users to write non-audio processing code that will run synchronously with an audio engine. They are used primarily for

their side-effects, as the return result is used to signal whether the function is complete or not. It is a tool that can act as the foundation for writing composition and application code.

Events

Pink's event code is found in the `pink.events` namespace. It includes functions for creating events, an `EventList` for holding pending events, and a scheduler function for processing events. In Pink, events are considered timed applications of functions. An event is fired by calling a given function at a given time with given arguments.

```
(event horn 0.0 0.4 440.0)
```

Listing 6.15: Example Pink event

Listing 6.15 shows an example of creating an event. The `event` function is called with four arguments: the `horn` function, which it will use when the event is fired, a start time of 0.0, and 0.4 and 440.0 as arguments to pass to the `horn` function. When the event processor in Pink fires the event, a horn audio function will be created.

Design Pink's event system follows a similar design to classic Music-N systems in providing the user a flat-list system of events. Events are also concrete data structures designed specifically for the single-purpose of firing actions at a given time. Instead of trying to directly accomodate higher-level music representation concerns within Pink's event system (e.g., hierarchical structuring of musical material, type hierarchies for kinds of musical events), the system focuses only on scheduling and firing actions in a generic manner.

The goal in this design is to provide the user the basic tools for timed executions of functions and to leave the decision of how to notate and organise music to the user. The user may decide that the flat-list system is adequate for their tasks as-is, use an available higher-level music representation library (such as Score, discussed in Section 6.5), or develop their own music representation system. When using a separate system from Pink, a translation layer will be required to convert event data from the system's format into Pink's event data structures.

Pink's event code is based around two primary data structures: Event and EventList. Listing 6.16 shows the code that defines both types. Events have three properties: `event-func`, `start`, and `event-args`. The start value designates at what time in beats the the event will be fired. When the event is fired, the `event-func` function will be executed, given the values held in `event-args`.

```
(deftype Event [event-func ^double start event-args ]
  Object
  (toString [_]  (format "\t%s\t%s\t%s\n" event-func start
                         event-args ))
  (hashCode [this] (System/identityHashCode this))
  (equals [this b] (identical? this b))

  Comparable
  (compareTo [this a]
    (let [t1 (.start this)
          t2 (.start ^Event a)]
     (compare t1 t2))))

(deftype EventList [^PriorityQueue events pending-events
                    cur-beat buffer-size sr tempo-atom]
```

```
Object
(toString [_]  (str events))
(hashCode [this] (System/identityHashCode this))
(equals [this b] (identical? this b)))
```

Listing 6.16: Definitions of Events and EventLists

EventList has a number of properties: `events`, `pending-events`, `cur--beat`, `buffer-size`, `sr`, and `tempo-atom`. `events` is an instance of `java.-util.PriorityQueue` and is a priority queue that sorts Events based on their start times. `pending-events` is an atom that contains a list of newly arrived events and acts as a message inbox. These events will be merged into the priority queue when event list processing occurs. `cur-beat`, `buffer-size`, `sr`, and `tempo-atom` are all used for calculating the current time in beats. This value is then used by the event list processing function to determine if any events are ready to fire.

Event Processing The `event-list-processor` function takes in an EventList and returns a Pink control function for processing of events. When the control function is executed, the following will occur:

1. Merge all pending events into the PriorityQueue.
2. Calculate current time in beats for the EventList.
3. In a loop, peek at the head of the PriorityQueue and check if it is time to fire the event. If so, fire the event and then discard it. Continue processing until the first event that is outside of the current time window is found, or until the queue is empty.

This algorithm is roughly equivalent to Dannenberg's "Implementation 2" in [49], with the addition of virtual-time scheduling. Here, virtual-time is

controlled by the tempo held in the `tempo-atom`. Event times are expressed not in seconds but in beats, relative to the current time of the event list. For example, if an event with start value of 1.0 passed to the event list, and the event list's current time is 10.0, then the start time of the event will be adjusted to 11.0 when it is merged into the events queue. If tempo is set to 60 beats per minute, the event will fire 1 second into the future.

Firing Events Pink's event processor is responsible only for firing events, and it has no knowledge of what the function does, nor does it use the function's result. In the case of Listing 6.15, if the `horn` event was fired, the audio function will be created but nothing would be done with it. If the user wants to add the horn audio function to the audio graph of the engine, the user would have to specifically do that in the code for the function argument of the event.

The general responsibility of the action's meaning is inverted from other systems because the event processor does not concern itself with what the function does. For example, in a MIDI processor, the processor would look at incoming data and decide based on the initial byte whether to start a new note or modify some internal state. As a result, there is a fixed set of possible event actions encoded into the MIDI Processor. To expand the kinds of events, one has to modify what kinds of messages the MIDI processor is able to understand as well as change what information is in the event message. This puts the burden of meaning and actions on the processor of events.

Instead, Pink events rely on the message creator to determine what the action will be. The event processor is only concerned with applying a function at a given time and nothing more. For example, given a MIDI note-on message with note number 64 and velocity 127, the MIDI processor might

read the message, determine that the channel maps to synthesiser-a, create a new instance of synthesiser-a, then add it to the engine's list of active audio functions.

In Pink, the responsibility is reversed. Instead of creating a message that maps to an action, the user embeds the action into the event. To achieve the previous example, a Pink event would have an `event-func` argument of `engine-add-afunc`. The `event-args` would include the `sythesiser-a` function and the expected arguments for `synthesiser-a`. When the event is processed, the processor would fire the `engine-add-afunc` function. This in turn might apply the `synthesiser-a` function to the rest of the `event-args` to create an audio function instance. `engine-add-afunc` would then add the audio function to the root audio processing Node's `:pending-adds` message inbox.

Because the user is in control of what happens at a given time, the core engine code can remain very small and simple, while at the same time be extremely expressive. Pink provides the very basic mechanisms of events as well as convenience functions for commonly used actions. However, the user is not limited to any pre-determined notion of what can be done by an event and is free to customise their events as they wish.

Higher-order Events Events in Pink are *higher-order* events, meaning that event arguments may themselves be functions. This capability at the event-level provides the same benefits as passing functions to functions does in higher-order functions. On a musical level, this allows for more flexible designs of audio functions as well as greater reuse.

For example, a violin is a string instrument. It is often used by bowing it with a violin bow. Performers can vary the speed and pressure of a bow

while performing. Performers may also use other techniques, such as plucking the string, hitting the string with the back of the bow, and so on. In all of these cases, the instrument itself has not changed, rather the input into the instrument has changed. Also, inputs to the string time-vary and are not static values.

In Pink, because an event is able to take in other functions, one can design an audio function to take in arguments and pull values during the processing of the audio function. For example, rather than pass in a static value for pitch, such as 440.0, one can pass in an audio function as the frequency argument that will give time-varying values. This allows for an audio-function acting as an instrument to be re-used to perform in a variety of ways, such as for playing a stable pitch or a glissando. It also allows the user to build up a library of audio-functions specifically for modeling performance gestures and reusing them between instruments.

Special Event Notation One problem that occurs with higher-order events is if a set of events was constructed and a user wanted to fire that set of events multiple times, the function instances that were used as arguments in the event may give unexpected results. This would be the case if the argument to an event is itself a stateful function.

```
(event horn 0.0 0.5 (env [0.0 440 0.5 880]))
```

Listing 6.17: Example problematic higher-order event

Listing 6.17 shows a problematic higher-order event. In this event, an `env` unit generator is used to vary the pitch from 440Hz to 880Hz over a 0.5 second period. When the event is created, the function returned from calling `env` is constructed. On the first time an event is called, the `env` instance

would be used when the `horn` function is applied. Everything would render fine the first time as the `env` instance is in its initial state. However, if the event is later reused, the same `env` instance would be used again, which would resume from its previous state.

To mitigate this scenario, Pink uses a special `apply!*!` operator when processing events. If any `IDeref` values are given as arguments, `apply!*!` will first deref the value before applying the function.[10] This differs from the standard `apply` operation, where arguments that are passed in a list would be statically processed at the time of list construction.

```
(def pitch (atom 440))
;; static pitch
(event horn 0.0 0.5 @pitch)
;; dynamic pitch
(event horn 0.0 0.5 pitch)
```

Listing 6.18: Example events using IDeref

Listing 6.18 shows a var called pitch that holds an atom with the value of 440. This is followed by two events. The first event dereferences the atom to get the value 440 when creating the event. The second event uses the atom directly as an argument. For the first event, the call to horn will always use the value 440 each time the event is processed. For the second event, the current value of pitch will be used each time the event is fired, due to the use of `apply!*!` in the event firing code. If the user calls `reset!` to alter the pitch atom to another value, it would not affect the first event but would affect the second event.

[10] `IDeref` is a base interface in Clojure for classes that support dereferencing. This is used for things like atoms and refs, which are used to hold mutable values.

To solve the problem with higher-order events as shown in Listing 6.17, the `!*!` function is provided which wraps the given code in an `IDeref`. This in turn creates an `IDeref` instance that, when dereferenced, calls `apply!*!` on the given function and arguments. For the corrected higher-order event in Listing 6.19, it will call `(env [0.0 440 0.5 880])` each time the event is fired to create a new `env` instance.

```
;; Use special !*! function to ensure a new env
;; instance is used each time this event is fired
(event horn 0.0 0.5 (!*! env [0.0 440 0.5 880]))
```

Listing 6.19: Corrected higher-order event

As a consequence of using `apply!*!`, if you do want to pass in an atom as an argument to the event's function without it being first dereferenced, you must use the `!r!` operator to wrap the atom. (`!r!` reads as a "reference argument".) For example, in Listing 6.20, when `perf-func` is applied, the third argument passed to it will not be the value of `tempo` but the `tempo` atom itself.

```
(def tempo (atom 60.0))
;; Use !r! to ensure the tempo atom is
;; passed as an argument to perf-func and
;; not first dereferenced by apply!*!
(event perf-func 0.0 0.5 (!r! tempo))
```

Listing 6.20: Event with reference argument

In general, if one is using higher-order events, it is likely one will use the `!*!` function. The use of `!r!` will most likely come into play when doing temporal recursion with events. In that scenario, it is useful to pass in things like a tempo or a done value, so that one can affect the recursive event stream elsewhere in the code.

Summary Pink provides a higher-order event system that uses virtual-time processing. This system processes events that contain a function to fire and arguments to supply to the function. This system puts the responsibility of an event's meaning on the creator of the event. The `event-list-processor` function provides a Pink control function that can be scheduled to run synchronously with Pink's engine.

6.4.4 Summary

Pink is an open-source, cross-platform music system written in Clojure. Users work in the Clojure programming language and employ Pink as a library for building musical works and applications. Pink is designed using a small set of abstractions — engine, signals, audio and control functions, higher-order events, and contexts — and it provides a basic set of implementations for those abstractions. These abstractions provide a simple, flexible, and extensible base for users to use and customise for their own works. Higher-level abstractions, such as 4MPS' model of ports and controls, are not currently provided, but, as demonstrated in Section 6.4.3, they may be built upon the existing abstractions.

Pink implements the library-based, general-purpose programming language model of music systems. Releases of Pink are distributed as versioned libraries which users can depend upon to create works with high degrees of stability in the face of change. Overall, Pink has satisfied the primary design goals for user-extensibility, support for both event-based works and realtime systems, use of minimal abstractions, and implementation of a library-based system design. Current plans for future work are to continue to develop the core library of signal processing, control, and utility functions, while also continuing

to develop works using Pink. In addition, as a system to research music systems design, Pink has shown higher-order events to be a useful tool for composing. Future work will apply the research from this project to further develop Csound to also support higher-order events.

6.5 Score

Score is a library of functions for creating musical scores as list data.[11] It is based on the *note* as a list of values, and *scores* – or note lists – as a higher level organisation of notes as lists of lists. Score contains useful musical functions for concepts such as pitch, scales, tunings, and amplitude. It also contains functions for generating, transforming, and organising scores. Because it is based on standard Clojure data structures, the library is designed to interoperate well with other Clojure functions and libraries that also work with lists. Score provides numerous points of extensibility and encourages users to draw upon their Clojure skills to customise their score-writing experience to their own taste.

6.5.1 Related Work

Score has been inspired by many computer music score languages and libraries. The following lists related work and their influences.

Common Music (CM) [174, 175] is an object-oriented score generation and performance library, originally written in Common Lisp. It features a score generation model as well as real-time scheduler for performance. The score

[11] For the purpose of this documentation, the term *list* is used synonymously with Clojure's concept of *sequences* [90].

model generically models musical ideas using its own model, and mappings are used to convert from the internal model to an external target system (i.e., CLM, Csound, MIDI, etc.). CM also includes the concept of *item streams*, which act as generators of values.

CM's item streams and generic music model were of particular influence to Score. Also, CM's earlier design as a library that would work with other CL-based systems like CLM was influential in developing Pink and Score as separate libraries. However, implementing a scheduler as well as using an object-based system were not aspects of the design that were factors for Score. In addition, the move from a library-based approach to an application one in CM3 [176] was a path I did not want to follow due to the benefits of employing versioned libraries for works.

CMask's [28] score model uses *fields* made up of *parameters* to generate note lists. Parameters are generators that take in a time argument and return a value. Parameters may optionally be used with *masks*, *quantizers*, and *accumulators* for further processing. Score reimplements CMask's model in the `score.mask` namespace using higher-order functions and also makes the model extensible for users to implement their own generators and processing functions. Score also provides features for hierarchical score organisation and processing which CMask does not support.

SuperCollider 3's Pattern library [87] provides various *Pattern* objects as item generators that can be used alone or together to generate events. Event generation is oriented around real-time use. Patterns may take into account the time of an event when used for event generation but most Patterns simply produce a stream of values. Score implements a similar score generation

model using Clojure's built-in sequence abstraction rather than employing a custom Pattern abstraction.

JMSL [60] uses a generic container called `MusicShape` for musical information that is further organised into hierarchies. This approach has similaritiese with Score's approach to using Clojure list data structures and score organisation functions with the exception that `MusicShapes` only support numerical values for their fields. However, like CM3, JMSL has a broader design that includes scheduling and performance, which was not a design goal of Score.

Canon's [50] emphasis on "scores as programs" and goal "to combine and transform simple scores to form more complex ones" was a model for Score. Canon provides a number of transformations based on time that depend on its implementation of *notes*, which uses a fixed set of fields. However, an open-ended model for notes was chosen for Score, which prevents providing similar kinds of transformations, as the meaning of fields is determined by the user, not the system. Users can, however, use standard list processing techniques to achieve the same kinds of transformations with Score as found in Canon.

SmOKe (Smalltalk Object Kernel) [139] is a music description language written in Smalltalk. It is a core part of the Siren [141] system and is based on work from Siren's predecessor, MODE [144]. SmOKe uses *Music Magnitudes* objects to represent common musical values (e.g., pitch, amplitude, and temporal values), *Event* dictionary objects as notes made up of values (which could be Magnitudes, Smalltalk function blocks, or any other object), and *EventList* list objects as containers for multiple Events. The EventList is itself an instance of the Event class, which allows EventLists to be considered composite event objects that can be embedded within other

events, thus allowing hierarchical organisation of musical material. SmOKe further provides *EventGenerator* and *EventModifier* objects to generate and transform EventLists and *Voice* objects to interpret SmOKe's Event music representation for use with a target music system.

Score provides many of the same features as SmOKe but differs in its implementation in a number of ways. Firstly, Score handles musical values by means of functions that convert from one value to another rather than using intermediary MusicMagnitude objects. Secondly, events in Score are list data structures rather than dictionaries. The use of lists aligns well with many target systems (e.g., Csound, MIDI, Pink) where event data is sequentially laid out. Writing lists is also less verbose than notating key-value pairs with dictionaries: the verbosity of dictionaries is beneficial for later reading and understanding of code but using lists can be quicker to write and fit more data on screen. Lists also enable use of Clojure's `apply` function to execute another function using the list's values as arguments. These benefits lead to the decision to use lists over dictionaries. Finally, Score's use of lists of lists as note lists is similar to SmOKe's EventLists but does not have the same class-type relationship to Events. SmOKe's Event objects can respond to `play` messages that trigger a walk of the hierarchy of objects; Score requires a separate score organisation function to walk the hierarchical list data. The differences in designs — whether to include behavior with an object or handle it separately in a function that processes data structures — largely reflect the programming practices found in object-oriented and functional programming languages.

Score shares much in common with the design of SmOKe. Each system offers concepts of musical values, events, event lists, generators, processors, and

mapping tools for target systems; allows hierarchical organisation of events; operates within the context of general-purpose programming languages; and provides means for users to extend the systems themselves. SmOKe currently provides more features (i.e., implementations of MusicMagnitudes and EventGenerators) than Score. However, future work will include implementing SmOKe's features currently not present within Score.

Looking at systems more broadly, those employing domain-specific languages can be categorized into notation and score generation systems. Notation systems (e.g., SCORE [163], Scot [75], ABC [186], LilyPond [126]) offer languages for hand-writing notes that are compiled and transformed to operate with a target system (e.g., Music V, Music 11, Csound, MIDI, PDF). Score generation systems (e.g., nGen [103], Score11 [82], CMask) provide DSLs for generating note lists using sets of field generators. The compact syntaxes of notation DSLs provide users with the unique ability to express much with little code. However, score generation DSLs look very much like code in general-purpose programming languages. Score provides comparable designs and features to the above DSL-based score generation systems but does not provide a notation-like system. Support for a short-hand notation system was outside of the scope of Score's current target feature set but may be considered for implementation in future versions.

Finally, GPL-based systems packaged as applications (AC Toolbox [32], Opus Modus [131], CM3) provide extensible programming environments for score generation. These Lisp-based systems provide not only a library of score functions but also text editors, graphics visualisations, schedulers, and other features. Users can extend these systems but must do so while working within the context of the provided application. Score differs from these systems by

packaging itself as a library. Partitioning the system at a smaller level frees users to employ the library within their own applications and use their own preferred tools.

6.5.2 Design

Score uses a functional programming approach to model musical scores using data and functions separately, rather than using objects which combine state and behavior. Notes (i.e. events) are represented using standard Clojure list data structures filled with values. The meaning of each value field of a note is entirely determined by the user. Scores (i.e., note or event lists) are lists of lists (i.e., notes) and they may be hierarchically organised by embedding one within another. Score provides functions for generating and transforming flat note lists as well as organising functions to transform hierarchical lists into a flattened total score. Once a total score is produced, users can use Score's mapping functions to map the data into a format suitable for a target music system. Score also provides a set of music value functions for generating values and transforming values from one format to another. Score's concepts map closely to those found in SmOKe (i.e., Music Magnitudes, Events, EventLists, EventGenerators, EventProcessors, and Voices).

Score is designed for use within Clojure code. Users may explicitly write out each note and organise them into lists, use code to generate and process note lists, or use a combination of the two together. By offering both hand-written notation and score generation, Score allows users to work with both the "score as data" approach found in Music-N systems as well as the "score as program" approach found in Canon.

The model of notes as a list of values, and scores as a list of notes, is very flexible. Users can use standard Clojure functions such `map`, `filter`, `reduce` to process the notes. They can further use functions like `concat` and `mapcat` to join together smaller blocks of notes into larger blocks. This allows a bottom-up composing of scores that has similarities to Western Music concepts of notes, phrases, sections, and movements.

Score's design is backend agnostic, meaning the internal representation is not designed for any specific target music system. The user can take a full list that represents a total score for a work, then do a final processing step to map the notes to a target format. This approach is exemplified in Common Music and SmOKE, where one develops their work with an internal score model and generates output as MIDI, Csound SCO, or other format.

Beyond the internal design, Score, like Pink, is designed as a library and not a system. Users can create works that depend upon a specific version of Score, and their work will continue to function even if newer versions of Score are produced. Also, Score has no other dependencies outside of Clojure itself, which simplifies the introduction of Score into a project. Finally, Score is designed to interoperate well with Pink, and vice versa.

6.5.3 Musical Values

Score provides a number of functions for generating and converting musical values. These functions are useful on their own as well as when generating and processing notes. These value functions are described below.

Amplitude and Frequency

Figure 6.8 lists basic functions provided by Score for conversion between different values for both amplitude and frequency.

db->amp

converts decibels to power ratios.

midi->freq

converts MIDI note numbers to frequency (Hz).

keyword->notenum

convert pitch keywords to MIDI note numbers.

keyword->freq

convert pitch keyword to frequency (Hz).

pch->notenum

convert PCH format to MIDI note number.

pch->freq

convert PCH format to frequency (Hz).

hertz

generic function for converting keyword, MIDI note number, or PCH to frequency (Hz).

Figure 6.8: Basic amplitude and frequency functions

These functions are useful to allow the user to write values in a form they find convenient and transform them into values appropriate for music systems to process. For example, the keyword format uses Clojure keywords to allow

for note pitches to be written using the note names and octave specifications that are common in Western art music notation.

```
user=> (keyword->notenum :C4)
60
user=> (keyword->notenum :C#4)
61
user=> (keyword->notenum :Bb4)
70
```

Listing 6.21: Conversions from keywords to MIDI note numbers

Listing 6.21 shows an example session where keywords are used with the `keyword->notenum` function to generate MIDI note numbers. The keyword `:C4` describes the note name C at octave 4, which corresponds to the MIDI note number 60 and the middle C key on a piano. Note names can be further modified by using `#` and `b` to denote sharps and flats.

Beyond decibels, amplitude, keywords, MIDI, and frequencies is the PCH format. This format is described further below.

PCH notation

Score's PCH notation is based on Csound pch notation.[12] In Csound, pch is a specially formatted number defined using "octave point pitch class". For example, `8.01` means "octave 8, pitch class 1" and is equivalent to the C# above middle C on a piano. Instead of using numbers, Score uses a 2-vector to represent PCH. The equivalent to Csound's `8.01` would be Score's `[8 1]`.

Besides the PCH to MIDI and frequency functions, Score provides additional PCH-related functions.

[12]For further information, see Table 8 in [179].

pch-add

adds an interval to a PCH and returns the new PCH, optionally taking in scale-degrees per octave (defaults to 12).

pch-diff

calculates the interval between two PCHs, optionally taking in scale-degrees per octave (defaults to 12).

pch-interval-seq

given an initial PCH, and list of intervals, generates a sequence of PCHs applying pch-add using the the previous PCH and new interval from the list.

analyze-intervals

given a list of PCHs, calculate the intervals between each PCH.

invert

create a chord inversion using a list of PCHs and inversion number.

Figure 6.9: PCH-related functions

Note, these PCH functions take into account the number of scale degrees per octave and normalize PCHs for overflows and underflows. For example, when `pch-add` is used with `[8 11]`, interval `1`, and scale-degrees 12, rather than return `[8 12]`, the value will be normalised to `[9 0]`. Figure 6.22 shows an example usage of PCH-related functions.

```
user=> (pch-add [8 0] 1)
[8 1]
user=> (pch-add [8 0] 13)
[9 1]
```

```
user=> (pch-add [8 0] -1)
[7 11]
user=> (pch-diff [8 0] [8 7])
7
user=> (pch-diff [8 0] [9 1])
13
user=> (pch-interval-seq [8 0] [2 3 -1])
([8 0] [8 2] [8 5] [8 4])
user=> (analyze-intervals [[8 0] [8 2] [8 5]])
[2 3]
user=> (invert [[8 1] [8 2] [8 3]] 1)
[[8 1] [7 2] [7 3]]
```

Listing 6.22: PCH-related functions usages

These functions provide useful functions for transforming PCH values and working with intervals between PCHs. They allow for common musical operations such as transposition and inversions. Retrogrades and sub-list operations can be achieved using Clojure's `reverse`, `drop`, and `take` functions.

Tunings

`score.tuning` provides functions for working with musical tunings. A tuning is defined using a Clojure map data structure with specific key/value pairs. Listing 6.23 shows an example of the twelve-tone equal temperament tuning, provided by Score.

```
(def ^:const ^{:tag 'double}
  MIDDLE-C 261.6255653005986)

(def TWELVE-TET
  { :description "Twelve-Tone Equal Temperament"
    :base-freq MIDDLE-C
```

```
 :num-scale-degrees 12
 :octave 2.0
 :ratios (map #(Math/pow 2.0 (/ % 12)) (range 12))
})
```

Listing 6.23: Twelve-tone equal temperament

Besides defining tunings by hand, the `create-tuning-from-file` function can be used to load files in the Scala file format [129]. This provides access to over 4000 scale files found in Scala's scale archive [128].

Once a tuning is created, the `pch->freq` function found in `score.tuning` can be used. This function takes in two arguments: a tuning and a PCH. As noted earlier, PCH is a 2-element list that provides an octave and scale degree. The result is the frequency for a given PCH.

Sieves

`score.sieves` provides a complete implementation of Xenakis's sieves, as defined in [193] and [192]. The implementation of `score.sieves` is a translation of the C code from those two sources. Ariza's extensions to sieves and implementation as objects [26] were also consulted, but the extensions to Xenakis's original models were not implemented in Score.

In `score.sieves`, sieves are represented using a 2-element list, made up of a modulo and index. These sieves can be combined using the `U` and `I` functions, which create Union and Intersection sieves respectively. Given a max number of steps and a sieve, the `gen-sieve` function sieves the series of positive numbers starting from 0 and returns the resulting sequence. Listing 6.24 shows an example coding session where four different sieve sequences are

generated. The examples use a simple sieve, a Union sieve, an Intersection sieve, and a complex sieve.

```
user=> (gen-sieve 12 [4 1])
(1 5 9 13 17 21 25 29 33 37 41 45)
user=> (gen-sieve 12 (U [4 1] [3 2]) )
(1 5 9 13 17 21 25 29 33 37 41 45)
user=> (gen-sieve 12 (I [4 1] [3 2]))
(5 17 29 41 53 65 77 89 101 113 125 137)
user=> (gen-sieve 12 (U [3 2] (I [3 2] [2 0])))
(2 5 8 11 14 17 20 23 26 29 32 35)
```

Listing 6.24: Example of generating sieved sequences

Analysis of sieves from a given sequence is also supported, using the `analyze-sieve` function. Listing 6.25 shows an example analysis. The function returns an analysis comprised of 3-vectors of sieves comprised of modulo, index, and number of values covered by that sieve.[13] The analysis also returns a Sieve object ready to use for generating new sequences as well as the period of repetition for the sieve.

```
user=> (analyze-sieve [0 2 3 5 8 11])
{ :analysis [[8 0 2] [3 2 4] [5 3 2]],
  :sieve #score.sieves.Union{:l #score.sieves.Union{:l [8
   0], :r [3 2]}, :r [5 3]},
  :period 120}
```

Listing 6.25: Example of sieve analysis

[13]This matches the design in the C-code.

6.5.4 Score Generation

Score includes two primary ways for generating note lists: `gen-notes` and `gen-notes2`. The first is based on Clojure sequences and is modeled on SC3's Pattern Library. The latter uses higher-order programming and time-based generator functions and is modeled on CMask.

gen-notes

Score's primary tool for generating notes is the `gen-notes` function:

```
(defn- score-arg
  "Utility function used by gen-notes to convert the given
  argument into a sequence if not so already."
  [a]
  (cond (sequential? a) a
    (fn? a) (repeatedly a)
    :default (repeat a)))

(defn gen-notes
  "Generate notes by assembling sequences together into
  notes. If a constant value is given, it will be wrapped
  with (repeat). If a no-arg function is given, it will be
  wrapped with (repeatedly)."
  [& fields]
  (let [pfields  (map score-arg fields)]
    (apply map (fn [& a] (into [] a)) pfields)))
```

Listing 6.26: Implementation of `gen-notes`

Given a set of fields – which may be sequences, functions, or values – `gen-notes` will generate a list of notes, where the value of each note is generated using the value from each field. If the field is a sequence, each item

of the sequence will be used. If the field is a function, it will be wrapped into a sequence using Clojure's `repeatedly` function. Finally, if a single value is given, an infinite list comprised of that value is created using the `repeat` function.

Because `gen-notes` uses `map`, the returned value is a lazy sequence. If all fields given to `gen-notes` are infinite sequence, then the resulting sequence is also infinite. If any of the fields are finite sequences, then the resulting sequence of notes will have a length equal to the shortest field sequence. The user should use the same care when using `gen-notes` as they would with regular Clojure sequences in regards to infinite sequence generation.

```
user=> (gen-notes 1 (range) 1.0 [1 2 3 4 5] (range 6 300))
([1 0 1.0 1 6]
 [1 1 1.0 2 7]
 [1 2 1.0 3 8]
 [1 3 1.0 4 9]
 [1 4 1.0 5 10])
```

Listing 6.27: Example use of `gen-notes`

Listing 6.27 shows an example usage of `gen-notes` and its results. In the call to `gen-notes`, the first and third fields are constants, 1 and 1.0. These values are repeated for each generated note. For the sequences used in the 2nd, 4th, and 5th fields, the first value from each sequence is used for the first generated note, then the next values used for the second note, and so on. As the 4th field is a finite list with the shortest number of elements, only five notes will be generated.

gen-notes2 and score.mask

Score offers an alternate model for generating notes, `gen-notes2`, based on time-based generator functions:

```
(defn- const
  "Returns a function that generates a constant value."
  [val]
  (fn [t]
    val))

(defn seq->gen
  "Converts a sequence into a generator function with time
   argument."
  [vs]
  (let [curval (atom vs)]
    (fn [t]
      (let [[a & b] @curval]
        (swap! curval rest)
        a
        ))))

(defn wrap-generator
  "Utility function to convert argument into a generator
   function
  if not so already."
  [f]
  (cond
    (seq? f) (seq->gen f)
    (fn? f) f
    :else (const f)))

(defn gen-notes2
```

```
"Generate notes with time-based generator functions. This
 score
generation method is based on CMask. Given fields should be
single-arg functions that generate a value based on time
 argument."
[start dur & fields]
(let [gens (map wrap-generator fields)
      [instrfn startfn & r] gens
      dur (double dur)
      start (double start)]
  (loop [cur-start 0.0
         retval []]
    (if (< cur-start dur)
      (let [i (instrfn cur-start)
            ^double xt (startfn cur-start)
            note (into [i (+ start cur-start)]
                       (map (fn [a] (a cur-start)) r))]
        (recur (+ cur-start xt) (conj retval note)))
      retval))))
```

Listing 6.28: Implementation of `gen-notes2`

Given an initial start time, duration, and set of fields – which may be sequences, functions, or values – `gen-notes2` will generate a list of notes, where the values of each note is generated using the values from each field. Unlike `gen-notes`, fields in `gen-notes2` are single-argument generator functions that take in a time value. If the field is a sequence, `seq->gen` will be called to convert the sequence into a generator function. If the field is a function, it is assumed to already be a generator function and used as-is. Finally, if a single value is given, an infinite generator function is produced using the `const` function.

`gen-notes2` is modeled on CMask's processing model, where `gen-notes2` maps to CMask's *fields*, and field arguments to `gen-notes2` map to CMask's *parameters*. For each note, all fields will be called given the current start time value (`cur-start`). The value generated by the second field's value will be especially used to increment `cur-start` for the next note generated. Generation of notes will continue until the `cur-start` value is greater than or equal to the `dur` argument.

All of CMask's parameters – oscillators, items, probabilities, and breakpoint functions – have been implemented as generator functions packaged in sub-namespaces of the primary `score.mask` namespace. Additionally, CMask's masks, quantisers, and accumulators have also been implemented as generator functions that decorate other generator functions. The generator functions provided by `score.mask`, together with `gen-notes2`, provide a complete implementation of CMask's capabilities within Score.

As with `gen-notes`, the user should take special care of using infinite generator functions. This is especially important as `gen-notes2` eagerly generates the resulting note list. If a finite generator function is not provided, calling `gen-notes2` will result in an infinite loop.

```
user=> (gen-notes2 0.0 4.0
                   4 0.5 3
                   (rand-range 0.1 20)
                   (item-cycle [1 2 3])
                   (swing [8 9 10])
                   (heap [10 100 400])
                   (rand-item [50 500 5000]))
[[4 0.0 3 0.5357457756267113 1 8 400 5000]
```

```
[4 0.5 3 16.091049682038065 2 9 10 500]
[4 1.0 3 5.7949011228034 3 10 100 50]
[4 1.5 3 14.749602188427321 1 9 10 50]
[4 2.0 3 14.01972320806139 2 8 100 500]
[4 2.5 3 10.87719434050349 3 9 400 500]
[4 3.0 3 17.53310173768086 1 10 400 5000]
[4 3.5 3 4.218225062429189 2 9 10 500]]
```

Listing 6.29: Example use of `gen-notes2`

Listing 6.29 shows an example usage of `gen-notes2` and its results. The first two arguments are the initial start time (0.0) and duration (4.0). The next 3 values are constants. Note that while the second field always returns 0.5, the generated value in the note list is the `cur-start` value calculated in the loop. The 4th through 8th fields are all time-based generator functions created by calling `score.mask` functions. For each of these fields, calling the `score.mask` function returns another function that takes in a time argument. This is shown in the implementation of `rand-item` in Listing 6.30.

```
(defn rand-item
  "Generates values as random permutations of a sequence"
  [vs]
  (fn [t]
    (rand-nth vs)))
```

Listing 6.30: Implementation of `rand-item`

The anonymous function returned by `rand-item` has a single argument `t`. In this case, the `t` argument is not used. Instead, the `vs` argument, which is closed over by the anonymous function, is used. While generator functions for use with gen-notes2 must take in a single time argument, they are not required to use it.

6.5.5 Score Transformation

Since notes are generic lists of data, the Score library has no explicit knowledge about what is in a note. It does not know if a field in a note is a PCH, a frequency, an amplitude, or other value. This limits the library from providing fixed operations such as transposing or stretching notes.

However, a different approach is used in Score. The `process-notes` macro allows a given note list to be transformed in a generic fashion. It takes in a single note list and then pairs of indexes and transformation functions. It processes the note list such that for each note, the values at the given indexes will be given to the corresponding transformation functions. The transformed field value is then used in the resulting transformed note list.

```
(def notes
  [['trumpet 0 1 -12 :G5]
   ['trumpet 1 1 -12 :B5]
   ['trumpet 3 1 -12 :D6]])

(process-notes notes
  3 db->amp
  4 keyword->freq)
```

Listing 6.31: Example use of `process-notes`

Listing 6.31 shows an example use of `process-notes`. It reads as "given the `notes` note list, process each note, converting the 4th field from decibels to amplitude multipliers and the 5th field from keywords to frequencies". Note that the indexes are 0-based, so 0 refers to the first field, 1 to the second field, and so on. The results of processing are shown in Listing 6.32.

```
([trumpet 0 1 0.251188643150958 783.9908719634985]
 [trumpet 1 1 0.251188643150958 987.7666025122485]
 [trumpet 3 1 0.251188643150958 1174.6590716696305])
```

Listing 6.32: Results of `process-notes`

This example shows one way of approaching score transformation, which is to allow writing note values in a form that is convenient to the user but transforming the values into one more suitable for signal processing routines. As the transformation functions provided are generic, `process-notes` can also be used to implement musical operations such as transpositions, decrescendos, time stretching, and so on.

`process-notes` provides a generic way to transform scores. As users are in control of specifying the meaning of field values for notes, users must also have a way to specify transformations by field. By providing transformation functions, the user is acknowledging they know what a field means as well as how they would like it be transformed.

For more complex transformations of scores, the processing model of `process-notes` may not be enough. However, as note lists are generic list data structures, users can avail themselves of Clojure's standard list processing functions to implement their own custom transformations.

6.5.6 Score Organisation

Score offers two primary functions for higher level organisation of music: `convert-timed-score` and `convert-measured-score`. The two functions take in list data structures written in timed- or measured-score formats. They will process the score formats and yield a single, flattened note list. The two

functions operate similarly with the exception of how they work with time specifications. They are described with examples below.

convert-timed-score

`convert-timed-score` allows the user to organise smaller blocks of score into a larger score. The user specifies a list of values that can either be numbers or note lists. If a number is encountered, it sets the current time for note list start time translation. If a note list is encountered, it will be translated in time by the current time. For note lists, `convert-timed-score` requires that the second field of each note be a value for a start time.

```
(def pattern
  [['bass-drum 0.0 0.5]
   ['bass-drum 1.0 0.5]
   ['bass-drum 2.0 0.5]
   ['bass-drum 3.0 0.5]])

(def score
  [0.0 pattern
   4.0 pattern])

(println (convert-timed-score score))
```

Listing 6.33: Example use of `convert-timed-score`

Listing 6.33 show an example usage of `convert-timed-score`. Firstly, a score fragment is explicitly written out by hand and assigned to the pattern variable. Secondly, the `score` variable is defined in the timed-score format. It reads as "at time 0.0, play pattern, and at time 4.0, play the pattern again".

Listing 6.34 shows the note list generated by calling `convert-timed-score` with the `score` variable.

```
([bass-drum 0.0 0.5]
 [bass-drum 1.0 0.5]
 [bass-drum 2.0 0.5]
 [bass-drum 3.0 0.5]
 [bass-drum 4.0 0.5]
 [bass-drum 5.0 0.5]
 [bass-drum 6.0 0.5]
 [bass-drum 7.0 0.5])
```

Listing 6.34: Results of `convert-timed-score`

`convert-timed-score` also allows for multiple note lists to be used for a given time. Listing 6.35 shows an example where two note lists, `bd-pattern` and `snare-pattern`, are used together in the timed-score. The results are shown in Listing 6.36.

```
(def bd-pattern
  [['bass-drum 0.0 0.5]
   ['bass-drum 1.0 0.5]
   ['bass-drum 2.0 0.5]
   ['bass-drum 3.0 0.5]])

(def snare-pattern
  [['snare-drum 1.0 0.5]
   ['snare-drum 3.0 0.5]])

(def score
  [0.0 bd-pattern
   4.0 bd-pattern snare-pattern])
```

```
(println (convert-timed-score score))
```

Listing 6.35: `convert-timed-score` with multiple note lists

```
([bass-drum 0.0 0.5]
 [bass-drum 1.0 0.5]
 [bass-drum 2.0 0.5]
 [bass-drum 3.0 0.5]
 [bass-drum 4.0 0.5]
 [bass-drum 5.0 0.5]
 [bass-drum 6.0 0.5]
 [bass-drum 7.0 0.5]
 [snare-drum 5.0 0.5]
 [snare-drum 7.0 0.5])
```

Listing 6.36: Results of `convert-timed-score` with multiple note lists

Since note lists are just lists, users can hand-write blocks of notes, use note-processing functions, and use note-generating functions within a timed score. Listing 6.37 shows an example of using inline hand-written note lists and function calls within a time-score. In the example, the second use of `bd-pattern` has been processed with the `process-notes` function, such that the 3rd field of each note has its value multiplied by 0.5. Also, a single-shot-sample note has been introduced to the score, written in by hand. Results are shown in Listing 6.38.

```
(def score
  [0.0 bd-pattern
   4.0 (process-notes bd-pattern 2 #(* % 0.5))
       snare-pattern
       [['single-shot-sample 2.0 2.0]]])
```

```
(println (convert-timed-score score))
```

Listing 6.37: Inline hand-written note lists and function calls

```
([bass-drum 0.0 0.5]
 [bass-drum 1.0 0.5]
 [bass-drum 2.0 0.5]
 [bass-drum 3.0 0.5]
 [bass-drum 4.0 0.25]
 [bass-drum 5.0 0.25]
 [bass-drum 6.0 0.25]
 [bass-drum 7.0 0.25]
 [snare-drum 5.0 0.5]
 [snare-drum 7.0 0.5]
 [single-shot-sample 6.0 2.0])
```

Listing 6.38: Inline hand-written note lists and function calls results

`convert-timed-score` provides users a way to organise score fragments in time. The results from calling this function is a flattened note list. This note list may in turn be assigned to a variable and used within other calls to `convert-timed-score`.

convert-measured-score

`convert-measured-score` operates similarly to `convert-timed-score`, but uses the *measure* as a unit of time rather than a *time* value. The measured-score is also a list, but begins with a :meter definition. Following the meter, values may be either numbers or lists, just as in timed-scores, but the numbers are interpreted as measure numbers.

```
(def score
```

```
  [:meter 4 4
  0 bd-pattern
  1 bd-pattern snare-pattern])

(println (convert-measured-score score))
```

Listing 6.39: Example use of `convert-measured-score`

Listing 6.39 shows an example usage of `convert-measured-score`. The score reads as "with a 4/4 meter, at measure 0, play bd-pattern, and at measure 1, play bd-pattern and snare-pattern". Start time values for notes are interpreted as beats, and beats map to quarter note values of the meter. The results are shown in Listing 6.40.

```
([bass-drum 0.0 0.5]
 [bass-drum 1.0 0.5]
 [bass-drum 2.0 0.5]
 [bass-drum 3.0 0.5]
 [bass-drum 4.0 0.5]
 [bass-drum 5.0 0.5]
 [bass-drum 6.0 0.5]
 [bass-drum 7.0 0.5]
 [snare-drum 5.0 0.5]
 [snare-drum 7.0 0.5])
```

Listing 6.40: Results of `convert-measured-score`

`convert-measured-score` allows for multiple note lists to be used per measure. Also, users may use in-lined, hand-written note lists and function calls embedded within measure-scores just as they would with timed-scores. For musical genres that use a regular, measured framework of time, using `convert-measured-score` may be more convenient to use and think with

than using `convert-timed-score`. Choosing between one or the other system of time will be dependent upon the user's own musical goals.

`convert-measured-score` and `convert-timed-score` simply process score lists and generate a note list. The results of these functions may themselves be further processed. This allows the user to mix usage of each time system. For example, if one was working on a film score, one could use measured-score to write the main music track and use a timed-score to add sound effects according to clock time. The user could then use `concat` to merge the two scores together.

6.5.7 Mapping Note Lists

Like Common Music and SmOKe, Score's internal design is backend agnostic. This means that the representation of data is not tied to a single target music system. Users can use mapping functions to convert note lists generated by Score into a format that works with another system. This may be for use with other computer music systems but may also be used for visualisation or other purposes.

Listing 6.41 shows an example use of Score and Csound. It uses the `gen-notes2` function, generating a note list from time 0 to 5.0 using 5 fields. The first field is a constant field that will always generate 1. The rest of the p-fields of the Csound score is generated using the values provided by within the `score.mask` package.

```
(def notes
  (gen-notes2 0 5.0
              1
              (gauss 0.5 0.1)
```

```
                (heap [0.1 0.2 0.4])
                (rand-range 0.1 0.25)
                (rand-item
                  ["8.00" "8.03" "8.02"])))
(def csound-sco
  (format-sco notes))

(println notes)
(println csound-sco)
```

Listing 6.41: Score and Csound Example: Code

Listing 6.42 shows the printed output from running Listing 6.41. The first printout shows the results of running `gen-notes2`, which produces a Clojure list of lists. The second printout shows the result of using the `format-sco` function, provided by Score for formatting note lists into Csound SCO text format. The `csound-sco` text may then be further sent to a running Csound instance for live score performance or written to disk and later read by Csound as a SCO file.

```
;; output from (println notes)
[[1 0.0 0.1 0.1455446063675899 8.02]
 [1 0.07388877495229043 0.2 0.11487888605849467 8.00]
 [1 0.2684591839186033 0.4 0.12170487899979296 8.00]
 [1 1.0558572506209922 0.4 0.13304255988624555 8.03]
 [1 1.554791683668857 0.2 0.16436113185377213 8.00]
 [1 1.9392915161730429 0.1 0.11907587313489418 8.02]
 [1 2.3410899943560195 0.2 0.21996317376289015 8.03]
 [1 2.787924993057282 0.4 0.2119026696996974 8.00]
 [1 3.7580 770774079575 0.1 0.12327608647786711 8.00]
 [1 4.199933807980773 0.2 0.23620482696864334 8.00]]
```

```
;; output from (println csound-sco)
i1 0.0 0.1 0.1455446063675899 8.02
i1 0.07388877495229043 0.2 0.11487888605849467 8.00
i1 0.2684591839186033 0.4 0.12170487899979296 8.00
i1 1.0558572506209922 0.4 0.13304255988624555 8.03
i1 1.554791683668857 0.2 0.16436113185377213 8.00
i1 1.9392915161730429 0.1 0.11907587313489418 8.02
i1 2.3410899943560195 0.2 0.21996317376289015 8.03
i1 2.787924993057282 0.4 0.2119026696996974 8.00
i1 3.7580770774079575 0.1 0.12327608647786711 8.00
i1 4.199933807980773 0.2 0.23620482696864334 8.00
```

Listing 6.42: Score and Csound Example: Output

At this time, Score only provides output mapping for Csound. However, Score's generated note lists are usable as-is with Pink, as both systems are written in Clojure. Listing 6.43 shows an example note list fragment, taken from the track1.clj example in the music-examples project [198]. In this example, `growing-line` defines a note list using both features from Score and Pink. The code first uses two note lists generated using the `gen-notes` function that are concatenated together. This is then mapped over and the `growl` audio function is prepended as the first field of each note in the note list. The `e` argument given to `gen-notes` is itself a Pink audio function – the `env` function – that is wrapped using the `!*!` operator. The result is that for each note, the 6th field will be an instance of `env` used as the amplitude argument to the `growl` instrument.

```
;; from music-examples.track1 example file
(def growing-line
  (let [e (!*! env [0.0 400 0.11 5000])
        starts (range 0 1.8 (/ 1.0 3.0))
```

```
        amps (range 0.05 5 0.05)
        space (range 0.75 -1.0 -0.25)]
    (map #(into [growl] %)
         (concat
           (gen-notes starts 0.1 :G5 amps e 0.75 space)
           (gen-notes starts 0.1 :G3 amps e 0.75 space)
           ))))
```

Listing 6.43: Score and Pink: Generating higher-order events

From here, the `growing-line` note list is then reused as a part of a larger measured-score. `convert-measured-score` is used to prodcue to the total score, which is then mapped into Pink events using the `sco->events` function provided in the `pink.simple` namespace.

```
(defn apply-afunc-with-dur
  "Applies an afunc to given args within the context of a
  given duration. with-duration will bind the value of dur
  to the *duration* Pink context variable."
  [afunc dur & args]
  (with-duration (double dur)
    (apply!*! afunc args)))

(defn i
  "Csound style note events: audio-func, start, dur, & args."
  [afunc start dur & args]
  (apply event apply-afunc-with-dur start afunc dur args))

(defn sco->events
  "Converts Csound-style note list into a list of
  Pink Events."
  [notes]
```

```
(map #(apply i %) notes))
```

Listing 6.44: sco->events function from pink.simple

Listing 6.44 shows the code for `sco->events`. Given a list of notes, `sco->event` maps an anonymous function that applies the `i` function to the values found in each note. The `i` function in turn applies the `event` function to each note, using `apply-afunc-with-dur` as the event's function – the one that will fired by Pink's event processor – with the given arguments. Finally, when `apply-afunc-with-dur` is called, it fires by processing the values found in the original note, applying the first field – the audio function – to the rest of the fields.

In the full track1.clj example, these Pink events are further passed to the `add-audio-events` function from `pink.simple`. This is a convenience function that wraps events with another event that uses the `add-afunc` function to attach audio functions to the root node of the engine. At runtime, when an event is fired, the nested event will generate an audio function and the top-level event will add it to the engine for processing.

The mapping of note lists is the technique by which the generated data from Score is connected to other systems. Score currently provides a mapping function for Csound and works out of the box with Pink, as shown in the example code. In the future, more mappings could be provided with Score, such as MIDI, OSC, and MusicXML. As the data generated from Score is plain Clojure list data, users can create their own mappings relatively simply.

6.5.8 Summary

Score provides users tools for generating, organising, and processing musical scores. It is based on the concept of note as lists of values and a score as a list of notes. Functions are provided for the generation of values for use as part of notes, including values for frequencies, amplitudes, scales and more. These value functions are used in conjunction with Clojure sequences or Score's generator functions to generate note lists using `gen-notes` or `gen-notes2`. These functions provide similar score generation facilities found in SC3's Patterns library and CMask respectively. Score also provides functions for higher-level organisation of musical material using `convert-measured-score` and `convert-timed-score`. These provide a simple way to merge hand-written note lists and note list fragments together into a full score. This provides a system for hierarchically organising music as well as expressing scores as programs.

Like Pink, Score is an open-source, cross-platform system packaged as versioned libraries. The use of standard list data structures, rather than class hierarchies or custom abstractions, allows easier interoperability with other libraries and user code that also works with standard lists. The design of Score provides the desired features of extensibility, reusability, and music system interoperability sought out in the goals for this project.

6.6 Using Pink and Score

The following will present an example project that uses both Pink and Score. The program is a single file [199] and is available from the music-projects

project online. Some comments have been removed from the examples so as not to repeat the discussion below.

The example project is designed for real-time performance. The user would first evaluate the main part of the file to define instrument functions, score fragments, control functions, and other source material. These will then be used at performance time by the user. The user would evaluate other lines of code to trigger instruments, play score fragments, and operate control functions by modifying values. The following will begin by discussing the definitions aspect of the project, then follow by describing the performance code.

6.6.1 Definitions

```
(ns music-examples.features
  (:require [score.core :refer :all]
            [score.freq :refer :all]
            [score.sieves :refer :all])
  (:require [pink.simple :refer :all]
            [pink.engine :refer :all]
            [pink.config :refer :all]
            [pink.control :refer :all]
            [pink.filters :refer :all]
            [pink.envelopes :refer :all]
            [pink.util :refer :all]
            [pink.node :refer :all]
            [pink.oscillators :refer :all]
            [pink.space :refer :all]
            [pink.event :refer :all]
            [pink.effects.ringmod :refer :all]
```

```
            [pink.effects.reverb :refer :all]
            ))
```

Listing 6.45: Pink/Score Example: Imports

Listing 6.45 shows the beginning of the project. Here in the namespace declaration, all relevant symbols and namespaces are imported using the `:require` clauses in the `ns` form.

```
(defn fm
  "Simple frequency-modulation sound with default 1.77:1 cm
   ratio"
  ([freq amp]
   (fm freq amp 0.4 1.77))
  ([freq amp fm-index mod-mult]
  (let [freq (shared (arg freq))
        mod-freq (mul freq mod-mult)]
    (let-s [e (if (fn? amp)
                amp
                (mul amp (adsr 0.02 2.0 0.0 0.01)))]
      (->
        (sine2 (sum freq (mul freq fm-index e
                              (sine2 mod-freq))))
        (mul e)
        )))))

(defn ringm
  "Simple instrument with ring-modulation"
  ([freq amp]
   (let [e (if (fn? amp)
             amp
             (mul amp (adsr 0.04 2.0 0.0 0.01)))]
     (->
```

```
(ringmod
  (blit-saw freq)
  (sine2 (mul freq 2.0)))
(mul e)
))))
```

Listing 6.46: Pink/Score Example: Instruments

Listing 6.46 shows the definition of two different instrument functions, one for FM synthesis, and the other using ring modulation. These functions take in arguments and call Pink unit generator functions to assemble the final signal producing audio function.

Both functions use the `->` threading macro [3] to simplify the writing of the code. They also both check if the given `amp` argument is a function, and, if so, use it as-is, otherwise multiply it with an `adsr` envelope function. This allows users to provide either an amplitude value to control a default envelope or a unit generator that can evolve over time and produce any amplitude curve the user desires.

The `fm` function shows both the use of `shared` and `let-s` to create shared versions of audio functions. The `freq` and `e` are consequently used in multiple parts of the audio function graph that is built up within the threading macro.

```
;; Create stable Nodes
(def dry-node (create-node :channels 2))
(def reverb-node (create-node :channels 2))

;; Add nodes to root Node
(add-afunc (node-processor dry-node))
(add-afunc (freeverb (node-processor reverb-node)
                     0.9 0.5))
```

```
(defn clear-afns
  "Utility function for clearing dynamically attached
  audio functions, but leaving stable audio graph in
  place."
  []
  (node-clear dry-node)
  (node-clear reverb-node))
```

Listing 6.47: Pink/Score Example: Stable audio graph

Listing 6.47 shows the creation of stable part of the project's audio graph. Two stereo Nodes are created, `dry-node` and `reverb-node`. These Nodes will be used for attaching audio functions during performance. Node-processing audio functions are generated with calls to `node-processor`.

The `dry-node`'s processor is attached directly to the root of the audio graph using `add-afunc` from `pink.simple`. The functions from `pink.simple` work with a single, global engine, which simplifies coding for most user projects. The processor for `reverb-node` is used as the input signal to the `freeverb` reverb processor, which is itself added to the root of the audio graph.

The `clear-afns` function is defined for convenience while performing. It will remove all audio functions attached from outside the stable parts of the audio graph. The stable parts will remain. This is useful as a "kill all" function in case something goes awry during performance.

```
(defn mix-afn
  "Applies panning (loc) to a mono audio function,
  then attaches to stereo values to dry and reverb nodes."
  [afn loc]
  (let-s [sig (pan afn loc)]
    (node-add-func
```

```
      dry-node
      (apply-stereo mul sig 0.7))
    (node-add-func
      reverb-node
      (apply-stereo mul sig 0.3)))
  nil)

(defn perf-fm
  "Performance function for FM instrument."
  [dur & args]
  (binding [*duration* dur]
    (mix-afn (apply!*! fm args) -0.1)))

(defn perf-ringm
  "Performance function for ringm instrument."
  [dur & args]
  (binding [*duration* dur]
    (mix-afn (apply!*! ringm args) 0.1)))
```

Listing 6.48: Pink/Score Example: Instrument performance functions

Listing 6.48 shows functions used for "performing" the instruments. `perf-fm` and `perf-ringm` are given a duration and set of arguments. Each function then applies `fm` and `ringm` to the arguments to create the mono-signal instrument. From there, they pass the insrument function to `mix-afn` with a location argument.

`mix-afn` first pans the instrument and implicitly wraps the panning audio function with `shared` by using `let-s`. Next, the `apply-stereo` function is used to apply the `mul` operator to each of the channels from the `sig` function with the given multiplier argument (0.7 for the dry signal, and 0.3 for the wet signal). `apply-stereo` uses special audio functions that handle splitting

multi-channel audio and merging the results back into a multi-channel signal. The final functions are then attached to both the `dry-node` and `reverb-node`.

While the multiplier values for the `apply-stereo` function calls are fixed, they could modified to use arguments passed into the `mix-afn` function. This would allow each instrument instance to have their own wet and dry mulitpliers.

```
(defn sieve-chord
  "Given instrument function, base pitch, and sieve,
  generate chord where sieve values are offsets from
   base-pch."
  ([base-pch sieve dur amp]
   (sieve-chord perf-ringm base-pch sieve dur amp))
  ([instrfn base-pch sieve dur amp]
  (gen-notes
    (repeat instrfn)
    0.0 dur (map #(pch->freq (pch-add base-pch %)) sieve)
    amp)))

;; glissandi score fragment with higher-order event arguments
(def gliss-fragment
  (map #(into [perf-fm] %)
       (gen-notes
         0.0 6.0
         (->>
           [:A4 :C5 :C#5 :E5]
           (map keyword->freq)
           (map #(!*! env [0.0 % 6.0 (* 1.2 %)])))
         (repeat (!*! env [0.0 0.0 3.0 0.2 3.0 0.0])))))

;; Score in measured-score format
(def score
```

```
[:meter 4 4
 0.0 (sieve-chord perf-fm [8 0]
                  (gen-sieve 7 [2 0]) 1.0 0.25)
 0.25 (sieve-chord perf-fm [8 3]
                  (gen-sieve 7 [2 0]) 3.0 0.25)
 1.0 (sieve-chord perf-fm [9 0]
                  (gen-sieve 7 (U [4 0] [3 1])) 1.0 0.25)
 1.25 (sieve-chord perf-fm [7 3]
                  (gen-sieve 7 (U [4 0] [3 1])) 3.0 0.25)
 2.0 (sieve-chord perf-fm [8 3]
                  (gen-sieve 7 [2 0]) 8.0 0.05)
 3.0 gliss-fragment
 ])
```

Listing 6.49: Pink/Score Example: Notelists

Listing 6.49 shows code that generates a score (i.e., note list) using functions from Score and Pink. `sieve-chord` uses the `gen-notes` function from Score to generate a note list. The `instrfn`, `dur`, and `amp` arguments are used as constant values that each generated note will share in common. The sieve argument is the generated list of values from a Xenakis-style sieve. Each value in the sieve will be used as a transposition value from the base PCH value that will further be converted into a frequency. The result is a note list that represents a chord.

`gliss-fragment` is a named note list generated with `gen-notes`. The third and fourth fields are produced using the `!*!` operator to wrap Pink `env` audio functions as arguments. The third field uses keyword notation from Score to define pitch values that are converted into frequencies; these frequencies are then used as arguments to create instance of `env` that will transition from the original frequency to 1.2 times the frequency over 6 seconds.

The fourth field is used to control amplitude and it will linearly grow and fade out over 6 seconds. The notes generated here will eventually be used to generate higher-order Pink events.

`score` is a named list in the `measured-score` format. It specifies a 4/4 meter and organises various sieve chords to be played at measures 0.0, 0.25, 1.0, 1.25, and 2.0. The `score` also defines that the `gliss-fragment` note list is used starting at measure 3.0. The named note list may be reused many times in the measured-score, though here it is used only once.

```
(defn s
  "Convenience function for creating a Pink event from
  a Score note."
  [afn start dur & args]
  (event #(apply afn dur args) start ))

(defn play-from
  "Plays score starting from a given measure."
  [^double measure]
  (->>
    (convert-measured-score score)
    (starting-at (* measure 4))
    (map #(apply s %))
    (add-events)
    ))
```

Listing 6.50: Pink/Score Example: Notelist performing functions

Listing 6.50 shows code for performing the note lists in real-time. `play-from` is used to perform the `score` starting from the given measure. The function will first convert the measured score into a simple notelist. Next, the note list is translated in time, and the `s` function is applied to each note of the

note list. This converts each note into a Pink event. Finally, all of the Pink events will be added to the global Pink engine for performance.

```
(defn cause [func start & args]
  "Implementation of Canon-style cause function."
  (add-events (apply event func start args)))

(defn echoes
  "Temporally-recursive function for performing echoes"
  [perf-fn counter dur delta-time freq amp]
  (let [new-count (dec counter)
        new-amp (* amp 0.5)]
    ;; perform fm instrument
    (perf-fn dur freq amp)
    (when (>= new-count 0)
      (cause echoes delta-time perf-fn new-count
             dur delta-time freq new-amp))))

;; partial function applications to make custom
;; echoes functions
(def fm-echoes (partial echoes perf-fm))
(def ringm-echoes (partial echoes perf-ringm))
```

Listing 6.51: Pink/Score Example: Temporal recursion

Listing 6.51 shows code for performing "echoes" of instrument notes. Firstly, the `cause` function is defined that mimics Canon's `cause` function and allows a simple way to schedule events. It reads as "play this function at this time with these arguments."

Next, `echoes` is defined as a temporally-recursive event function. When `echoes` is fired, it will play a given instrument with a given `amp` and `freq` values. Next, it will decrement the given counter and check if it is greater

than or equal to 0. If so, the function will use `cause` to create another event to execute `echoes` at `delta-time` in the future. The new event will use the same instrument function and frequency, a new amplitude with half the value of the previous `amp`, and the new value for counter. The result is that when a user calls `echoes`, it will play the note `counter` number of times and produce an echoing effect.

```
(defn pulsing
  "Triggers ringm instrument and given frequency and
   delta-buffer time as atoms. User can adjust values for
   args externally."
  [done-atm freq delta-buffers]
  (let [counter (atom 0)]
    (fn []
      (when (not @done-atm)
        (swap! counter inc)
        (when (>= @counter @delta-buffers)
          (cause perf-ringm 0.0 5.0 @freq
                 (env [0 0.0 2.5 0.5 2.5 0.0]))
          (reset! counter 0))
        true))))
```

Listing 6.52: Pink/Score Example: Control function

Listing 6.52 shows the definition of a control function called `pulsing`. It uses atoms as arguments for signaling that processing should stop (`done-atm`), the current frequency (`freq`), and the number of buffers to wait before firing off a `ringm` note (`delta-buffers`). When the control function is added to the engine, it will increment its running counter and check if it is greater than the value held in the `delta-buffers` argument. Once the condition is

met, `pulsing` performs the note using the current value of `freq` and resets the counter for the next call.

Once the control function is running in the engine, the user has the opportunity to modify how it will perform by modifying the values within the atom arguments. By resetting the values in the atom, the user can change the pitch and frequency of the pulsing effect.

6.6.2 Performance Functions

The definitions above are the material which are then used for performance. The following performance functions are provided in the full example file within a `comment` form so that they are not run or performed when first loading the file. Instead, the user will start the engine manually using the `start-engine` function, then evaluate the code within the comments for live coding performance. User may also modify the code while performing. The following will cover the three main performance gestures.

```
(cause fm-echoes 0.0 5 0.25 1.5 400.0 0.5)
(cause fm-echoes 0.0 5 0.25 3.5 900.0 0.5)
(cause fm-echoes 0.0 5 0.25 4.5 800.0 0.5)
(cause fm-echoes 0.0 5 0.25 3.0 1500.0 0.5)
(cause ringm-echoes 0.0 5 0.25 2.5 220.0 0.5)
(cause ringm-echoes 0.0 5 0.25 4.25 60.0 0.5)
(cause ringm-echoes 0.0 5 0.25 4.25 51.0 0.5)
```

Listing 6.53: Pink/Score Example: Perform echoes

Listing 6.53 shows the use of the cause function to create different echoes. The different instances are generally differentiated by their instrument performance function, frequency, and time between echoes (i.e., `delta-time`).

The user can evaluate one line at a time to execute a single echo, or evaluate multiple lines to create chords where the echoes go out of phase due to their `delta-time` differences.

```
;; play score
(play-from 0)
(play-from 2)

;; play just glissando part
(play-from 3.0)
```

Listing 6.54: Pink/Score Example: Perform score

Listing 6.54 uses the `play-from` function to play back the pre-written `score` from various start times. The third version is timed to perform just the glissando part of the score.

```
;; Values held in atoms to be used both by control
;; functions and realtime manipulation by user
(def done-atm (atom false))
(def freq (atom 31.0))
(def delta-buffers (atom 3200))

(def done-atm2 (atom false))
(def freq2 (atom 33.0))
(def delta-buffers2 (atom 3500))

;; Predefined code to select and evaluate during
;; performance
(reset! done-atm true)
(reset! done-atm false)

(reset! done-atm2 true)
(reset! done-atm2 false)
```

```
(reset! freq 31.0)
(reset! freq2 33.0)

(reset! freq 41.0)
(reset! freq2 44.0)

(reset! delta-buffers 5300)
(reset! delta-buffers2 4700)

;; Evaluate to add control functions to perform
;; "pulsing" musical material
(add-post-cfunc (pulsing done-atm freq delta-buffers))
(add-post-cfunc (pulsing done-atm2 freq2 delta-buffers2))
```

Listing 6.55: Pink/Score Example: Perform pulsing

Listing 6.55 shows the use of two `pulsing` control functions. Two sets of atoms are defined, then used as arguments to each `pulsing` function. Once the functions are running in the engine, the user can execute the `reset!` code lines to modify the behavior of the pulsing effect.

6.7 Conclusions

In this chapter, I have presented two new music systems as libraries: Pink and Score. They are both open-source and work within the context of the general-purpose programming language Clojure. They were both developed for extensibility from the start.

With Pink, I developed a music engine capable of handling audio signal, control function, and event processing. It provides a fully-formed system for immediate use by the user. It also supports writing pre-composed and real-time works. The system has provided extensibility at all levels of abstraction. Users can customise the system for their own work, from creating new signal processing functions all the way to modifying or replacing the engine. This empowers the user to take advantage of whatever they desire from the library for their musical work.

With Score, I developed a library for generating and processing of higher-level symbolic representations of music. It employs the standard Clojure list to represent a musical note and a list of notes to represent scores. It comes with a number of functions for generating values, generating and transforming notes, and processing hierarchical organization of note lists. As Score generates standard Clojure lists, it interoperates well with Pink and mappings can be developed easily to work with other music systems.

The result of these systems is that they provide many features, can be extended, and work well with other Clojure code. By releasing Pink and Score as versioned libraries, users can specify and depend on an exact version of these systems without concern for any changes that may be introduced to either library. Pink and Score have all of the properties of growth over time and protection from change that were sought out at the beginning of this thesis.

Chapter 7

Conclusions

Extensibility in computer music systems is the way that developers and users can extend the programs they develop and work with. It is rooted in the consideration of software over time. As users ask more of their software, extensibility dictates who can extend the system and how it can be done. As the environment of computing changes, extensibility factors into how well-suited a program is to adapting to new or updated platforms. These qualities of extensibility address not only the growth and sustainability of software but also the durability and long-term value of a user's work and practice.

The original contributions of this thesis approached extensibility in numerous ways. In Chapter 3, the new type system, Parser3, revision of opcode polymorphism, and implementation of Runtime Type Identification all played a part to refine the infrastructure of Csound's language. These internal changes both simplified as well as enabled new ways to extend the language by developers.

The implementation of arrays, function-call syntax, explicit types, user-defined types, and new user-defined opcode syntax were built upon the new infrastructure. These language design changes have provided users with new ways to express their ideas and new opportunities to extend the Csound system themselves.

The developments for Csound 6 and Csound 7 have contributed to developer- and user-extensibility of the Csound language. It has done so in a backwards-compatible way, preserving the history of Csound works while opening up new ways to explore musical ideas in Csound.

In Chapter 4, the exploration of platform extensibility has brought Csound to new kinds of platforms. By porting Csound to mobile (iOS and Android) and Web (Emscripten and PNaCl) platforms, the overall ecosystem of Csound has grown. Users gained new places to run their existing works as well new ways to use their existing Csound knowledge and experiences.

In Chapter 5, the development of the Modular Score timeline in Blue was used to explore the benefits of run-time module-based systems. By making layers and layer groups a plugin, third-party developers can now extend the score timeline to offer unique new ways of working with music while coordinating with existing time-based interfaces. The new Pattern and Audio layers were implemented as plugins and demonstrated the flexibility of what could be implemented in the new system.

Finally, in Chapter 6, two new library-based music systems were presented: Pink and Score. These systems were developed to maximise user-extensibility and to explore the benefits of working within a general-purpose programming language. With Pink, the system was designed to offer users a complete audio music engine and signal processing library, while also providing the means to

reuse or replace parts to customise all aspects of the system for their work. With Score, the library was designed to work with generic lists of data as notes, which lets users easily integrate Score with other libraries or develop new functions to work with Score. Both systems offer releases as versioned libraries, providing a way for users to preserve their works by preserving the exact system used.

These explorations into extensibility have provided numerous facets to consider when developing computer music systems. They have also extended existing systems to provide new features as well as provided new systems to explore. The work to make extensible computer music systems will continue as long as computing changes and users require more for their work.

7.1 Original Contributions

The following lists original contributions completed for this thesis. It is organized by software and area of research.

Csound Language

- New type system.
- Introduction of arrays.
- Modification of opcode polymorphism.
- Extension of function-call syntax.
- Implementation of Runtime Type Identification.
- New Parser design (Parser3).

- Ability to explicitly specify types for variables.
- Introduced user-defined types.
- New user-defined opcode syntax.

Csound Platform

- Designed CsoundObj API and CsoundBindings system.
- Contributed to porting of Csound to iOS, development of examples and build system for the Csound for iOS SDK.
- Contributed to porting of Csound to Android, development of examples and build system for the Csound for Android SDK.
- Modified Csound to build with Emscripten.
- Developed Csound Notebook and Processing.js examples.

Blue

- Redesigned data model and UI architecture to support plugins for the Modular Score.
- Implemented new Audio Layers.
- Implemented new Pattern Layers.

Pink and Score

- Developed Pink, a new audio engine and music system library.
- Developed Score, a new library for generating and processing note lists.

7.2 Future Work

For Csound, the discussion of rates and data types in Chapter 3 looked at possibilities for reifying update rates as a first-class property of data types. The array universal data type has already been implemented (discussed in Section 3.2.2) and lays the groundwork for future universal types to be introduced into Csound. Listing 7.1 shows speculative language changes that would employ new universal types together with keywords or qualifier syntax to declare variables that operate at specific rates. A C-like typedef system is also shown that would allow defining a simpler `float` numeric data type and redefining Csound's `i`-, `k`-, and `a`-types as rate-attributed forms of `floats`. Further research in this area is required to evaluate whether the benefits of first-class rates would offset the implementation and pedagogical costs of remodeling Csound data types.

```
Sval = ``mutable string''
Init Sval = ``immmutable string''  ;; Keyword modifier
val:Init:S = ``immmutable string'' ;; additional qualifier
    in variable name

;; Typedefs for Csound's original types using keywords
typedef Init float i
typedef Control float k
typedef Audio float a

;; Typedefs for Csound's original types using qualifiers
typedef float:Init i
typedef float:Control k
typedef float:Audio a
```

Listing 7.1: Speculative Csound syntax for declaring rates

Analysing Csound's opcode system revealed a disparity between how arguments are handled when calling native opcodes and user-defined opcodes. For the former, all arguments are always passed-by-reference, and for the latter, all arguments are always passed-by-value. Pass-by-value introduces a performance cost when using UDOs that could be addressed if pass-by-reference was permitted.

However, switching to pass-by-reference outright for UDOs internally would introduce a backwards incompatibility for users who may have written code that mutates input arguments within their UDOs. Introducing new syntax to specify that UDO arguments should be handled as references may be a possible backwards-compatible solution. Listing 7.2 shows speculative syntax using new-style UDOs with keyword modifiers or custom syntax applied to argument type specifiers. The compiler would require modification to track reference arguments and the runtime would require an additional address-setting pass for propogating references. Further research is required to investigate both the appropriate syntax to use and the overall cost of introducing pass-by-reference and pass-by-value concepts to users.

```
;; keyword modifier syntax
opcode my_opcode(ref fftdata:f):(ref f)
  ...
  xout out_fsignal
endop

;; C-like reference syntax
opcode my_opcode(fftdata:f*):(f*)
  ...
  xout out_fsignal
endop
```

```
;; C++-like reference syntax
opcode my_opcode(fftdata:f&):(f&)
  ...
  xout out_fsignal
endop
```

Listing 7.2: Speculative Csound syntax for pass-by-reference UDO arguments

The new language features developed for Csound 7 within this thesis may be seen as but a step along a road well explored by other programming languages. Explicit types (Section 3.3.2) were necessary to allow naming variables without restrictions to the first letter used. However, the use of explicit types may become onerous over time, especially with very large bodies of code. Extending Csound to perform *type inference* [63] to determine the type of a variable would build upon the work of explicit types and allow users to freely write variable names without types, yet still statically type-check code.

Listing 7.3 shows a possible future Csound language using type inference. The first example shows an explicitly-typed version of code that would be possible using Csound 7 syntax. Next, the same code is shown where type inference is employed to resolve the type of the variables. The information from both the opcodes argument types and previous uses of variables would be used to to determine the type of the variable. Finally, the last example shows a case where historical Csound code shows an ambiguity with the `oscil` opcodes. In this example, using regular type inference alone would make it valid for `ksig` and `asig0` to be either `k`- or `a`-type variables. In this situation, the compiler would have to report an ambiguity in the code. If the type inference system additionally considered the previous single-letter rule

as part of its type resolution algorithm, prior Csound code would resolve to the same types as before. This shows a possible path for type inference that would provide users the freedom to write their code using names as they wish, yet still retain backwards compatibility and type safety.

```
;; explicitly typed variables
amp:i = 0.5
freq:i = 440
cutoff:i = freq * 4
sig0:a = vco2(amp, freq)
sig1:a = moogladder(sig0, cutoff)
out(sig0)

;; type-inferred code
amp = 0.5
freq = 440
cutoff = freq * 4
sig0 = vco2(amp, freq)
sig1 = moogladder(sig0, cutoff)
out(sig0)

;; single-letter rule resolves to k-rate var
ksig  oscil 0.25, 440
asig0 oscil 0.5, 440 * (1 + ksig)
```

Listing 7.3: Example Csound code using type inferences

Another Csound language change for the future would be to introduce opcodes as a type. This could open the door for more functional programming techniques to enter in to Csound use, such as higher-order functions (i.e., opcodes). Additionally, modifying Csound's event system to allow using opcode instances as arguments to events would reproduce the benefits found

in Pink's higher-order events (Section 6.4.3). Other language features to explore in Csound include the introduction of classes and objects as well as new generic data types, such as sets and dictionaries. These features have not yet been considered in detail and no speculation on syntax or implementation is given at this time.

For Blue, new kinds of layers are planned. Notation layers would permit the use of Western music notation on the timeline. Arc layers would be based on UPIC [118] and allow drawing lines to create events with time-varying pitch. These new layer types would add interesting ways to work with music over time in conjunction with the existing layer types.

For Pink, while the system design is extremely flexible, the included library of signal processing functions is currently limited. Adding implementations of audio processing routines found in more mature systems – such as Csound and SuperCollider 3 – would make Pink a more viable option when users consider what music system to use for new works. Also, the ability to write real-time event generation code similar to Common Music's *processes* [176] would bring a well known music programming model to Pink. Macros could be developed to transform process-like code into Pink control functions suitable for use with a Pink engine.

Finally, for Score, current plans are to maintain the current design and continue to expand the library of composition functions. This would include both existing functionality found in other systems as well as new research as it develops in the field. This would benefit users by providing them a large set of features that they can choose from, one that can easily integrate with their own personal musical programming work.

Bibliography

[1] Audiokit. http://www.audiokit.io. Accessed: 2016-03-28.

[2] Clojars. http://www.clojars.org. Accessed: 2016-03-28.

[3] ClojureDocs: ->. http://clojuredocs.org/clojure.core/->. Accessed: 2016-03-28.

[4] Creative Commons BY 3.0. http://http://creativecommons.org/licenses/by/3.0/. Accessed: 2016-03-28.

[5] CsoundQT. http://csoundqt.github.io. Accessed: 2016-03-28.

[6] CTK - The Common Toolkit. http://www.commontk.org. Accessed: 2016-03-28.

[7] Flaticon. http://www.flaticon.com. Accessed: 2016-03-28.

[8] Freepik. http://www.freepik.com. Accessed: 2016-03-28.

[9] Ruby on Rails. http://www.rubyonrails.org. Accessed: 2016-03-28.

[10] threads.c. https://github.com/csound/csound/blob/develop/Top/threads.c. Accessed: 2016-03-28.

[11] Sam Aaron et al. Overtone. http://overtone.github.io. Accessed: 2016-03-28.

[12] Samuel Aaron and Alan F. Blackwell. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, pages 35–46, New York, NY, USA, 2013. ACM.

[13] Ableton. Live. `https://www.ableton.com/en/live/`. Accessed: 2016-03-28.

[14] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison-Wesley, 1986.

[15] Xavier Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing*. PhD thesis, Universitat Pompeu Fabra, 2004.

[16] Xavier Amatriain. A Domain-Specific Metamodel for Multimedia Processing Systems. *IEEE Transactions on Multimedia*, 9(6):1284–1298, October 2007.

[17] Xavier Amatriain and Pau Arumi. Developing Cross-platform Audio and Music Applications with the CLAM Framework. In *Proceedings of the International Computer Music Conference*, pages 403–410, 2005.

[18] Xavier Amatriain, Pau Arumi, and David Garcia. CLAM: A framework for efficient and rapid development of cross-platform audio applications. In *Proceedings of the 14th annual ACM international conference on Multimedia*, pages 951–954. ACM, 2006.

[19] Android Open Source Project. ART and Dalvik. `https://source.android.com/devices/tech/dalvik/`. Accessed: 2016-03-28.

[20] Android Open Source Project. Audio Latency Measurements. https://source.android.com/devices/audio/latency_measurements.html. Accessed: 2016-03-28.

[21] Android Open Source Project. CPUs and Architectures. http://developer.android.com/ndk/guides/arch.html. Accessed: 2016-03-28.

[22] The Apache Software Foundation. Apache Celix. http://celix.apache.org. Accessed: 2016-03-28.

[23] The Apache Software Foundation. Maven Central Repository. http://maven.org. Accessed: 2016-03-28.

[24] Apple Inc. Audio Session Programming Guide. https://developer.apple.com/library/ios/documentation/Audio/Conceptual/AudioSessionProgrammingGuide/ConfiguringanAudioSession/ConfiguringanAudioSession.html. Accessed: 2016-03-28.

[25] Apple Inc. iOS 9 - What is iOS - Apple Inc. http://www.apple.com/ios/what-is/. Accessed: 2016-03-28.

[26] Christopher Ariza. The Xenakis Sieve as Object: A New Model and a Complete Implementation. *Computer Music Journal*, 29(2):40–60, 2005.

[27] Audiobus. Audiobus. https://audiob.us/. Accessed: 2016-03-28.

[28] Andre Bartetzki. CMask: A stochastic event generator for Csound. https://www2.ak.tu-berlin.de/~abartetzki/CMaskMan/CMask-Manual.htm, 1997. Accessed: 2016-03-28.

[29] David M Beazley et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk workshop*, pages 129–139, 1996.

[30] Olivier Bélanger. Pyo. `http://ajaxsoundstudio.com/software/pyo/`. Accessed: 2016-03-28.

[31] Ross Bencina and Phil Burk. PortAudio – an Open Source Cross Platform Audio API. In *Proceedings of the International Computer Music Conference*, 2001.

[32] Paul Berg. Using the AC Toolbox: A tutorial. `http://www.actoolbox.net/data/documents/AC_Toolbox_Tutorial.pdf`. Accessed: 2016-03-28.

[33] Heiko Böck. *The Definitive Guide to NetBeans Platform 7*. Apress, 2011.

[34] Richard Boulanger and Victor Lazzarini, editors. *The Audio Programming Book*. MIT Press, 2011.

[35] Richard J. Boulanger, editor. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February 2000.

[36] Boulanger Labs. csGrain. `http://www.boulangerlabs.com/products/csgrain/`. Accessed: 2016-03-28.

[37] Boulanger Labs. csGrain. `http://www.boulangerlabs.com/products/csgrain/cs_grain_manual.pdf`. Accessed: 2016-03-28.

[38] Ollie Bown, Ben Porter, Benito, et al. The Beads Project - Real-time Audio for Java and Processing. `http://www.beadsproject.net/`. Accessed: 2016-03-28.

[39] Peter Brinkmann. *Making Musical Apps*. O'Reilly Media, Inc., 2012.

[40] Phil Burk. JSyn–A Real-time Synthesis API for Java. In *Proceedings of the 1998 International Computer Music Conference*, pages 252–255. International Computer Music Association San Francisco, 1998.

[41] John Calcote. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press Series. No Starch Press, 2010.

[42] Rui Nuno Capela. Qtractor. `http://qtractor.sourceforge.net/qtractor-index.html`. Accessed: 2016-03-28.

[43] Cockos Inc. ReaScript. `http://cockos.com/reaper/sdk/reascript/reascript.php`. Accessed: 2016-03-28.

[44] Douglas J. Collinge. MOXIE: A Language for Computer Music Performance. In *Proceedings of the 1984 International Computer Music Conference (ICMC)*, New York City, 1984.

[45] Tcl Community. Tcl Developer Site. `http://tcl.tk`. Accessed: 2016-03-28.

[46] Perry R Cook and Gary Scavone. The Synthesis Toolkit (STK). In *Proceedings of the International Computer Music Conference*, pages 164–166, 1999.

[47] Oracle Corporation. Netbeans Rich-Client Platform Development (RCP). `https://netbeans.org/features/platform/`. Accessed: 2016-03-28.

[48] Roger B. Dannenberg. The CMU MIDI Toolkit. In *Proceedings of the International Computer Music Conference (ICMC)*, pages 53–56, Netherlands, 1986.

[49] Roger B Dannenberg. Real-Time Scheduling and Computer Accompaniment. In Max V Mathews and John R Pierce, editors, *Current Directions in Computer Music Research*. MIT Press, 1989.

[50] Roger B. Dannenberg. The Canon Score Language. *Computer Music Journal*, 13(1):47–56, 1989.

[51] Roger B. Dannenberg. The implementation of Nyquist, a sound synthesis language. *Computer Music Journal*, 21(3):71–82, 1997.

[52] Roger B. Dannenberg. Machine tongues XIX: Nyquist, a language for composition and sound synthesis. *Computer Music Journal*, 21(3):50–60, 1997.

[53] Roger B. Dannenberg. A Language for Interactive Audio Applications. In *Proceedings of the International Computer Music Conference (ICMC)*, San Francisco, 2002.

[54] Roger B. Dannenberg and Eli Brandt. A Flexible Real-Time Software Synthesis System. In *Proceedings of the International Computer Music Conference (ICMC)*, Hong Kong, 1996.

[55] Paul Davis et al. Ardour. `http://www.ardour.org`. Accessed: 2016-03-28.

[56] Paul Davis et al. JACK Audio Connection Kit. `http://www.jackaudio.org`. Accessed: 2016-03-28.

[57] Erik de Castro Lopo. libsndfile. `http://mega-nerd.com/libsndfile/`. Accessed: 2016-03-28.

[58] Peter Desain. LISP as a Second Language: Functional Aspects. *Perspectives of New Music*, 28(1):pp. 192–222, 1990.

[59] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-First Garbage Collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48. ACM, 2004.

[60] Nick Didkovsky and Phil Burk. Java Music Specification Language, an introduction and overview. In *Proceedings of the International Computer Music Conference*, pages 123–126, 2001.

[61] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[62] Charles Donnelly and Richard Stallman. *Bison: The YACC-compatible Parser Generator, Bison Version 3.0.4*. Free Software Foundation, 2015.

[63] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[64] Chas Emerick, Brian Carper, and Christophe Grand. *Clojure Programming*. O'Reilly Media, Inc., 2012.

[65] Emscripten Contributors. Emscripten. `http://www.emscripten.org`. Accessed: 2016-03-28.

[66] John ffitch. Parallel Execution of Csound. In *Proceedings of ICMC 2009*, Montreal, 2009. ICMA.

[67] John ffitch, Victor Lazzarini, and Steven Yi. Csound6: old code renewed. In *Linux Audio Conference 2013*, Graz, Austria, April 2013.

[68] John ffitch, Victor Lazzarini, Steven Yi, Michael Gogins, and Andres Cabrera. The New Developments in Csound 6. In *Proceedings of ICMC 2013*, Perth, 2013. ICMA.

[69] Michael Fogus and Chris Houser. *The Joy of Clojure: Thinking the Clojure Way*. Manning Publications Co., 2011.

[70] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[71] James George and YCAMInterlab. Commits - YCAMInterlab/-Duration. `https://github.com/YCAMInterlab/Duration/commits/master`. Accessed: 2016-03-28.

[72] James George and YCAMInterlab. Duration README.md. `https://github.com/YCAMInterlab/Duration/blob/master/README.md`. Accessed: 2016-03-28.

[73] James George and YCAMInterlab. Duration: Timeline for Creative Coding. `http://www.duration.cc/`. Accessed: 2016-03-28.

[74] Steinberg Media Technologies GmbH. Cubase. `http://www.steinberg.net/en/products/cubase/start.html`. Accessed: 2016-03-28.

[75] Michael Gold, John Stautner, and Steven Haflich. *An Introduction to Scot*. MIT Studio for Experimental Music, 1980.

[76] Google. Android. `http://www.android.com/`. Accessed: 2016-03-28.

[77] Google. Angularjs. `http://www.angularjs.org`. Accessed: 2016-03-28.

[78] Google. Native Client. `https://developer.chrome.com/native-client`. Accessed: 2016-03-28.

[79] Google. Patchfield. `https://github.com/google/patchfield`. Accessed: 2016-03-28.

[80] Google. Pepper C API Reference (Stable). `https://developer.chrome.com/native-client/c-api`. Accessed: 2016-03-28.

[81] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. Pearson Education, 2014.

[82] Robert Gross, Alexander Brinkman, and Matt Barber. SCORE-11 Reference Manual. `http://ecmc.rochester.edu/ecmc/docs/score11/index.html`. Accessed: 2016-03-28.

[83] Object Management Group. Corba. `http://www.corba.org/`. Accessed: 2016-03-28.

[84] Phil Hagelberg et al. Leiningen. `http://leiningen.org/`. Accessed: 2016-03-28.

[85] Richard Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in Action: Creating Modular Applications in Java*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2011.

[86] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[87] Henry James Harkins. A Practical Guide to Patterns. In *SuperCollider 3.3 Documentation*. 2009.

[88] David Herman, Luke Wagner, and Alan Zakai. asm.js: Working Draft 18 August 2014. `http://www.asmjs.org/spec/latest`. Accessed: 2016-03-28.

[89] Rich Hickey. Clojure. `http://www.clojure.org`. Accessed: 2016-03-28.

[90] Rich Hickey. Sequences. `http://clojure.org/reference/sequences`. Accessed: 2016-03-28.

[91] Rich Hickey. Transient Data Structures. `http://clojure.org/reference/transients`. Accessed: 2016-03-28.

[92] Rich Hickey. Vars and the Global Environment. `http://clojure.org/reference/vars`. Accessed: 2016-03-28.

[93] Doug Hoyte. *Let Over Lambda: 50 years of Lisp*. Lulu.com, 2008.

[94] Apple Inc. Logic Pro X. `http://www.apple.com/logic-pro/`. Accessed: 2016-03-28.

[95] Cakewalk Inc. Sonar. `http://www.cakewalk.com/Products/SONAR`. Accessed: 2016-03-28.

[96] Cockos Inc. Reaper. `http://www.cockos.com/reaper`. Accessed: 2016-03-28.

[97] Alan Curtis Kay. *The Reactive Engine*. PhD thesis, The University of Utah, 1969.

[98] Damián Keller, Victor Lazzarini, and Marcelo S Pimenta. *Ubiquitous Music*. Springer, 2014.

[99] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Prentice-Hall Englewood Cliffs, 2nd edition, 1988.

[100] Michael Kircher and Prashant Jain. Pooling Pattern. In *EuroPLoP 2002 Conference*, Kloster Irsee, Germany, 2002.

[101] Donald E. Knuth. Structured Programming with go to Statements. *Computing Surveys*, 6:261–301, 1974.

[102] Glenn E. Krasner and Stephen Travis Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[103] Mikel Kuehn. The nGen Manual. `http://www.mikelkuehn.com/ngen/man212/ngenman.htm`. Accessed: 2016-03-28.

[104] Paul Lansky. The Architecture and Musical Logic of Cmix. In *Proceedings of the International Computer Music Conference (ICMC)*, Glasgow, 1990.

[105] Tito Latini. Incudine. `http://incudine.sourceforge.net/`. Accessed: 2016-03-28.

[106] Victor Lazzarini. Scoreless Csound: Running Csound from the Orchestra. *Csound Journal*, (20), 2014. Accessed: 2016-03-28.

[107] Victor Lazzarini, Edward Costello, Steven Yi, and John ffitch. Csound on the Web. In *Linux Audio Conference 2014*, 2014.

[108] Victor Lazzarini, Edward Costello, Steven Yi, and John ffitch. Extending Csound to the Web. In *Proceedings of the Web Audio Conference 2015*, 2015.

[109] Victor Lazzarini, Steven Yi, and Joseph Timoney. Digital Audio Effects on Mobile Platforms. In *Proceedings of DAFx 2012*, 2012.

[110] Victor Lazzarini, Steven Yi, and Joseph Timoney. Web Audio: Some Critical Considerations. In *Proceedings of the VI Ubiquitous Music Workshop*, Växjö, 2015.

[111] Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. The Mobile Csound Platform. In *Proceedings of ICMC 2012*, 2012.

[112] Jean-Pierre Lemoine. AVSynthesis. `http://avsynthesis.blogspot.com/`. Accessed: 2016-03-28.

[113] John R. Levine. *Linkers and Loaders*. Operating Systems. Morgan Kaufmann, 2000.

[114] John R Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly Media, Inc., 1992.

[115] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Java series. Addison-Wesley, 1999.

[116] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification: Java SE 8 Edition*. Addison-Wesley, 2014.

[117] Barbara Liskov and Stephen Zilles. Programming with Abstract Data Types. In *ACM Sigplan Notices*, volume 9, pages 50–59. ACM, 1974.

[118] Henning Lohner. The UPIC System: A User's Report. *Computer Music Journal*, 10(4):pp. 42–49, 1986.

[119] Gareth Loy. The CARL System: Premises, History, and Fate. *Computer Music Journal*, 26(4):52–60, 2002.

[120] Ken Martin and Bill Hoffman. *Mastering CMake*. Kitware, 2010.

[121] Max V Mathews, Joan E Miller, F Richard Moore, John R Pierce, and Jean-Claude Risset. *The Technology of Computer Music*. MIT press Cambridge, 1969.

[122] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[123] James McCartney. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[124] Microsoft. Component Object Model (COM). `https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573(v=vs.85).aspx`. Accessed: 2016-03-28.

[125] Mozilla Developer Network and individual contributors. ScriptProcessorNode. `https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode`. Accessed: 2016-03-28.

[126] Han-Wen Nienhuys and Jan Nieuwenhuizen. LilyPond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics*, pages 167–172, 2003.

[127] Tim O'Brien, Manfred Moser, John Casey, Brian Fox, Jason Van Zyl, Eric Redmond, and Larry Shatzer. Maven: The Complete Reference. `http://books.sonatype.com/mvnref-book/reference/index.html`. Accessed: 2016-03-28.

[128] Manuel Op de Coul. Scala Downloads - Scale Archive. `http://www.huygens-fokker.org/scala/downloads.html#scales`. Accessed: 2016-03-28.

[129] Manuel Op de Coul. Scala scale file format. `http://www.huygens-fokker.org/scala/scl_format.html`. Accessed: 2016-03-28.

[130] openFrameworks Community. openFrameworks. `http://openframeworks.cc`. Accessed: 2016-03-28.

[131] Opusmodus Ltd. OpusModus. `http://www.opusmodus.com`. Accessed: 2016-03-28.

[132] Oracle. Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide. `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html#default_heap_size`. Accessed: 2016-03-28.

[133] Yann Orlarey, Dominique Fober, and Stéphane Letz. Faust: an Efficient Functional Approach to DSP Programming. *New Computational Paradigms for Computer Music*, 290, 2009.

[134] John K Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[135] Vern Paxson, Will Estes, and John Millaway. Lexical Analysis with Flex, for Flex version 2.5.37, 2012.

[136] Dave Philips. A brief survey of Linux audio session managers. `http://lwn.net/Articles/533594/`. Accessed: 2016-03-28.

[137] Dave Phillips. Composing With Csound In AVSynthesis. *Csound Journal*, (10), 2009. Accessed: 2016-03-28.

[138] Benjamin C Pierce. *Types and Programming Languages*. MIT press, 2002.

[139] Stephen Travis Pope. The SmOKe music representation, description language, and interchange format. In *Proceedings of the International Computer Music Conference*, pages 106–106, 1992.

[140] Stephen Travis Pope. Machine Tongues XV: Three Packages for Software Sound Synthesis. *Computer Music Journal*, 17(2):23, 1993.

[141] Stephen Travis Pope. The Siren 7.5 Music and Sound Package in Smalltalk. In *Proceedings of the International Computer Music Conference*, 2007.

[142] Stephen Travis Pope, Xavier Amatriain, Lance Putnam, Jorge Castellanos, and Ryan Avery. Metamodels and design patterns in CSL4. In *Proceedings of the 2006 International Computer Music Conference*, 2006.

[143] Stephen Travis Pope and Chandrasekhar Ramakrishnan. The Create Signal Library ("Sizzle"): Design, Issues and Applications. In *Proceedings of the 2003 International Computer Music Conference (ICMC'03)*, 2003.

[144] Steven Travis Pope. The Musical Object Development Environment: MODE (Ten Years of Music Software in Smalltalk). In *Proceedings of the International Computer Music Conference*, pages 241–241. International Computer Music Association, 1994.

[145] ProcessingJS Team. processing.js. `http://processingjs.org/`. Accessed: 2016-03-28.

[146] Miller Puckette. Pd documentation. `http://msp.ucsd.edu/Pd_documentation/index.htm`. Accessed: 2016-03-28.

[147] Miller Puckette. The Patcher. In *Proceedings of the International Computer Music Conference*, pages 420–425, San Francisco, 1988.

[148] Miller Puckette. Pure Data: another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[149] The QT Company. Qt. `http://www.qt.io`. Accessed: 2016-03-28.

[150] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley professional computing series. Pearson Education, 2003.

[151] Brian Redfern. Introducing the Android CSD Player: Jam Live with Android and Csound. *Csound Journal*, (17), 2012. Accessed: 2016-03-28.

[152] Charles Roberts, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. Gibber: Abstractions for Creative Multimedia Programming. pages 67–76. ACM Press, 2014.

[153] ROLI Ltd. JUCE. `https://www.juce.com`. Accessed: 2016-03-28.

[154] Hanns Holger Rutz. ScalaCollider. `http://www.sciss.de/scalaCollider`. Accessed: 2016-03-28.

[155] Carla Scaletti. Kyma: An Object-oriented Language for Music Composition. In *Proceedings of the International Computer Music Conference*, pages 49–56, 1987.

[156] Carla Scaletti. The Kyma/Platypus computer music workstation. *Computer Music Journal*, 13(2):23–38, 1989.

[157] Gary P. Scavone. RtAudio: A cross-platform C++ class for realtime audio input/output. In *Proceedings of the International Computer Music Conference*, pages 196–199, 2002.

[158] Gary P. Scavone and Perry R. Cook. RtMidi, RtAudio, and a synthesis toolkit (STK) update. In *Proceedings of the 2005 International Computer Music Conference*, 2005.

[159] Bill Schottstaedt. CLM. `https://ccrma.stanford.edu/software/snd/snd/clm.html`. Accessed: 2016-03-28.

[160] Bill Schottstaedt. Machine Tongues XVII: CLM: Music V Meets Common Lisp. *Computer Music Journal*, 18(2):30, 1994.

[161] Alexander Shaw, Dan Stowell, et al. SuperCollider-Android. https://github.com/glastonbridge/SuperCollider-Android. Accessed: 2016-03-28.

[162] Charles Simonyi. Hungarian Notation. https://msdn.microsoft.com/en-us/library/aa260976. Accessed: 2016-03-28.

[163] Leland Smith. SCORE - A Musician's Approach to Computer Music. *Journal of the Audio Engineering Society*, 20(1):7–14, 1972.

[164] Andrew Sorensen. Impromptu: An interactive programming environment for composition and performance. In *Proceedings of the Australasian Computer Music Conference 2009*, 2005.

[165] Andrew Sorensen and Henry Gardner. Programming with Time: Cyber-physical programming with Impromptu. *ACM Sigplan Notices*, 45(10):822–834, 2010.

[166] Andrew Sorenson. Extempore. http://extempore.moso.com.au/. Accessed: 2016-03-28.

[167] Andrew Sorenson. The Many Faces of a Temporal Recursion. http://extempore.moso.com.au/temporal_recursion.html. Accessed: 2016-03-28.

[168] Symbolic Sound. Kyma X. http://www.symbolicsound.com/cgi-bin/bin/view/Products/WebHome. Accessed: 2016-03-28.

[169] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher-order and symbolic computation*, 13(1-2):11–49, 2000.

[170] The SWIG Developers. SWIG. `http://swig.org`. Accessed: 2016-03-28.

[171] The SWIG Developers. SWIG Executive Summary. `http://swig.org/exec.html`. Accessed: 2016-03-28.

[172] Richard Taruskin. *Music from the Earliest Notations to the Sixteenth Century: The Oxford History of Western Music*, chapter 1, pages Kindle Locations 528–532. Oxford University Press, Kindle edition, 2010.

[173] Richard Taruskin. *Music in the Late Twentieth Century: The Oxford History of Western Music*, chapter 10, pages Kindle Locations 10466–10472. Oxford University Press, Kindle edition, 2010.

[174] Heinrich Taube. Common Music: A music composition language in Common Lisp and CLOS. *Computer Music Journal*, pages 21–32, 1991.

[175] Heinrich Taube. An Introduction to Common Music. *Computer Music Journal*, 21(1):29, 1997.

[176] Heinrich Taube. Common Music 3. In *Proceedings of the International Computer Music Conference (ICMC 2009)*, Montreal, Canada, 2009.

[177] Barry Vercoe. *Reference manual for the MUSIC 360 language for digital sound synthesis.* Studio for Experimental Music, MIT, 1973.

[178] Barry Vercoe. *MUSIC 11 Reference Manual.* Studio for Experimental Music, MIT, 1981.

[179] Barry Vercoe et al. cpspch. In *The Canonical Csound Reference Manual, Version 6.05.* `http://csound.github.io/docs/manual/cpspch.html`. Accessed: 2016-03-28.

[180] Barry Vercoe et al. Csound. https://github.com/csound/csound/. Accessed: 2016-03-28.

[181] Barry Vercoe et al. lenarray. In *The Canonical Csound Reference Manual, Version 6.05*. http://csound.github.io/docs/manual/subinstr.html. Accessed: 2016-03-28.

[182] Barry Vercoe et al. loop_lt. In *The Canonical Csound Reference Manual, Version 6.05*. http://csound.github.io/docs/manual/loop_lt.html. Accessed: 2016-03-28.

[183] Barry Vercoe et al. opcode. In *The Canonical Csound Reference Manual, Version 6.05*. http://csound.github.io/docs/manual/opcode.html. Accessed: 2016-03-28.

[184] Barry Vercoe et al. subinstr. In *The Canonical Csound Reference Manual, Version 6.05*. http://csound.github.io/docs/manual/subinstr.html. Accessed: 2016-03-28.

[185] Barry Vercoe et al. until. In *The Canonical Csound Reference Manual, Version 6.05*. http://csound.github.io/docs/manual/until.html. Accessed: 2016-03-28.

[186] Chris Walshaw. ABC. http://abcnotation.com. Accessed: 2016-03-28.

[187] Ge Wang. ChucK: Language Specification > Concurrency & Shreds. http://chuck.cs.princeton.edu/doc/language/spork.html. Accessed: 2016-03-28.

[188] Ge Wang, Perry R Cook, et al. ChucK: A concurrent, on-the-fly audio programming language. In *Proceedings of International Computer Music Conference*, pages 219–226, 2003.

[189] Watanabe-DENKI Inc. et al. supercollider_iOS. `https://github.com/wdkk/supercollider_iOS`. Accessed: 2016-03-28.

[190] Scott Wilson, David Cottle, and Nick Collins. *The SuperCollider Book*. The MIT Press, 2011.

[191] World Wide Web Consortium. Web Audio API. `http://webaudio.github.io/web-audio-api/`. Accessed: 2016-03-28.

[192] Iannis Xenakis. *Formalized Music: Thought and Mathematics in Composition*. Number 6 in Harmonologia Series. Pendragon Press, 1992.

[193] Iannis Xenakis and John Rahn. Sieves. *Perspectives of New Music*, pages 58–78, 1990.

[194] Steven Yi. Blue. `http://blue.kunstmusik.com`. Accessed: 2016-03-28.

[195] Steven Yi. Blue: a music composition environment for Csound. `http://blue.kunstmusik.com/manual/html/index.html`. Accessed: 2016-03-28.

[196] Steven Yi. Csound Notebook. `http://csound-notebook.kunstmusik.com`. Accessed: 2016-03-28.

[197] Steven Yi. Csound Notebook. `http://github.com/kunstmusik/csound-notebook`. Accessed: 2016-03-28.

[198] Steven Yi. music-examples. `https://github.com/kunstmusik/music-examples`. Accessed: 2016-03-28.

[199] Steven Yi. music-examples: features.clj. https://github.com/kunstmusik/music-examples/blob/master/src/music_examples/features.clj. Accessed: 2016-03-28.

[200] Steven Yi. ProcessingJS and Csound PNaCl Example. https://github.com/kunstmusik/processingjs_example. Accessed: 2016-03-28.

[201] Steven Yi. sndfile.c.patch. https://github.com/csound/csound/blob/develop/emscripten/patches/sndfile.c.patch. Accessed: 2016-03-28.

[202] Steven Yi, Roger Dannenberg, Victor Lazzarini, and John ffitch. Extending Aura with Csound Opcodes. In *Proceedings of ICMC 2014*, Athens, Greece, 2014. ICMA.

[203] Steven Yi and Victor Lazzarini. Csound for Android. In *Linux Audio Conference 2012*, 2012.

[204] Steven Yi, Victor Lazzarini, Roger Dannenberg, et al. Extending aura with csound opcodes. In *40th International Computer Music Conference, ICMC 2014, Joint with the 11th Sound and Music Computing Conference, SMC 2014-Music Technology Meets Philosophy: From Digital Echos to Virtual Ethos*, pages 1542–1549. University of Bath, 2014.

[205] Alon Zakai. Emscripten: An LLVM-to-Javascript Compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, pages 301–312. ACM, 2011.

[206] Johannes M. Zmölnig. How to Write an External for Pure Data. *Institute for Electronic Music and Acoustics*, March 2014. http://

pdstatic.iem.at/externals-HOWTO/pd-externals-HOWTO.pdf. Accessed: 2016-03-28.

Appendix A

The Mobile Csound Platform

Original Publication:

Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. The Mobile Csound Platform. *In Proceedings of ICMC 2012*, 2012.

THE MOBILE CSOUND PLATFORM

Victor Lazzarini, Steven Yi, Joseph Timoney

Sound and Digital Music Technology Group
National University of Ireland, Maynooth, Ireland
victor.lazzarini@nuim.ie
stevenyi@gmail.com
jtimoney@cs.nuim.ie

Damian Keller

Nucleo Amazonico de Pesquisa Musical (NAP)
Universidade Federal do Acre, Brazil
dkeller@ccrma.stanford.edu

Marcelo Pimenta

LCM, Instituto de Informatica
Universidade Federal do Rio Grande do Sul, Brazil
mpimenta@info.ufrgs.br

ABSTRACT

This article discusses the development of the Mobile Csound Platform (MCP), a group of related projects that aim to provide support for sound synthesis and processing under various new environments. Csound is itself an established computer music system, derived from the MUSIC N paradigm, which allows various uses and applications through its Application Programming Interface (API). In the article, we discuss these uses and introduce the three environments under which the MCP is being run. The projects designed for mobile operating systems, iOS and Android, are discussed from a technical point of view, exploring the development of the CsoundObj toolkit, which is built on top of the Csound host API. In addition to these, we also discuss a web deployment solution, which allows for Csound applications on desktop operating systems without prior installation. The article concludes with some notes on future developments.

1. INTRODUCTION

Csound is a well-established computer music system in the MUSIC N tradition [1] , developed originally at MIT and then adopted as a large community project, with its development base at the University of Bath. A major new version, Csound 5, was released in 2006, offering a completely re-engineered software, as a programming library with its own application programming interface (API).
This allowed the system to be embedded and integrated into many applications. Csound can interface with a variety of programming languages and environments (C/C++, Objective C, Python, Java, Lua, Pure Data, Lisp, etc.). Full control of Csound compilation and performance is provided by the API, as well software bus access to its control and audio signals, and hooks into various aspects of its internal data representation. Composition systems, signal processing applications and various frontends have been developed to take advantage of these features. The Csound API has been described in a number of articles [4], [6] [7].

New platforms for Computer Music have been brought to the fore by the increasing availability of mobile devices for computing (in the form of mobile phones, tablets and netbooks). With this, we have an ideal scenario for a variety of deployment possibilities for computer music systems. In fact, Csound has already been present as the sound engine for one of the pioneer portable systems, the XO-based computer used in the One Laptop per Child (OLPC) project [5]. The possibilities allowed by the re-engineered Csound were partially exploited in this system. Its development sparked the ideas for a Ubiquitous Csound, which is now steadily coming to fruition with a number of parallel projects, collectively named the Mobile Csound Platform (MCP). In this paper, we would like to introduce these and discuss the implications and possibilities provided by them.

2. THE CSOUND APPLICATION ECOSYSTEM

Csound originated as a command-line application that parsed text files, setup a signal processing graph, and processed score events to render sound. In this mode, users hand-edit text files to compose, or use a mix of hand-edited text and text generated by external programs. Many applications–whether custom programs for individual use or publicly shared programs–were created that could generate text files for Csound usage. However, the usage scenarios were limited as applications could not communicate with Csound except by what they could put into the text files, prior to starting rendering.

Csound later developed realtime rendering and event input, with the latter primarily coming from MIDI or standard input, as Csound score statements were also able to be sent to realtime rendering Csound via pipes. These features allowed development of Csound-based music systems that could accept events in realtime at the note-level,

such as Cecilia [8]. These developments extended the use cases for Csound to realtime application development.

However, it was not until Csound 5 that a full API was developed and supported that could allow finer grain interaction with Csound [3]. Applications using the API could now directly access memory within Csound, control rendering frame by frame, as well as many other low-level features. It was at this time that desktop development of applications grew within the Csound community. It is also this level of development that Csound has been ported to mobile platforms.

Throughout these developments, the usage of the Csound language as well as exposure to users has changed as well. In the beginning, users were required to understand Csound syntax and coding to operate Csound. Today, applications are developed that expose varying degrees of Csound coding, from full knowledge of Csound required to none at all. Applications such as those created for the XO platform highlight where Csound was leveraged for its audio capabilities, while a task-focused interface was presented to the user. Other applications such as Cecilia show where users are primarily presented with a task-focused interface, but the capability to extend the system is available to those who know Csound coding. The Csound language then has grown as a means to express a musical work, to becoming a domain-specific language for audio engine programming.

Today, these developments have allowed many classes of applications to be created. With the move from desktop platforms to mobile platforms, the range of use cases that Csound can satisfy has achieved a new dimension.

3. CSOUND FOR IOS

At the outset of this project, it was clear that some modifications to the core system would be required for a full support of applications on mobile OSs. One of the first issues arising in the development of Csound for iOS was the question of plugin modules. Since the first release of Csound 5, the bulk of its unit generators (opcodes) were provided as dynamically-loaded libraries, which resided in a special location (the OPCODEDIR or OPCODEDIR64 directories) and were loaded by Csound at the orchestra compilation stage. However, due to the uncertain situation regarding dynamic libraries (not only in iOS but also in other mobile platforms), it was decided that all modules without any dependencies or licensing issues could be moved to the main Csound library code. This was a major change (in Csound 5.15), which made the majority of opcodes part of the base system, about 1,500 of them, with the remaining 400 or so being left in plugin modules. The present release of Csound for iOS includes only the internal unit generators.

With a Csound library binary for iOS (in the required arm and x86 architectures, for devices and simulators), a new API was created in Objective-C, called CsoundObj. This is a toolkit that provides a wrapper around the standard Csound C API and manages all hardware connectivity. A CsoundObj object controls Csound performance and provides the audio input and output functionality, via the CoreAudio AuHAL mechanism. MIDI input is also handled either by the object, by allowing direct pass-through to Csound for standard Csound MIDI-handling, or by routing MIDI through a separate MIDIManager class to UI widgets, which in turn send values to Csound. Additionally, a number of sensors that are found on iOS devices come pre-wrapped and ready to use with Csound through CsoundObj.

To communicate with Csound, an object-oriented callback system was implemented in the CsoundObj API. Objects that are interested in communicating values, whether control data or audio signals, to and from Csound must implement the CsoundValueCacheable protocol. These CsoundValueCacheables are then added to CsoundObj and values will then be read from and written to on each control cycle of performance (fig.1). The CsoundObj API comes with a number of CsoundValueCacheables that wrap hardware sensors as well as UI widgets, and examples of creating custom CsoundValueCacheables accompany the Csound for iOS Examples project.

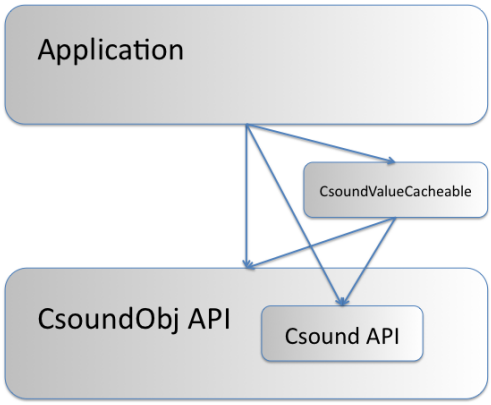

Figure 1. CsoundObj and the Application

While the CsoundObj API covers most of the general use cases for Csound, it does not wrap the Csound C API in its entirety. Instead, the decision was made to handle the most common use cases from Objective-C, and for less used functions, allow retrieval of the CSOUND object. This is the lower-level object that encapsulates all of the C API functionality. It is a member of CsoundObj and it is exposed so that developers can use methods not directly available in that class. It is expected that as more developers use CsoundObj, the CsoundObj API may continue to further wrap C API functions as they are identified as being popular.

Together with the API for iOS, a number of application examples complete the SDK. These can be used during development both as a practical guide for those interested in using Csound on iOS, as well as a test suite for the API. Examples include a number of realtime instruments

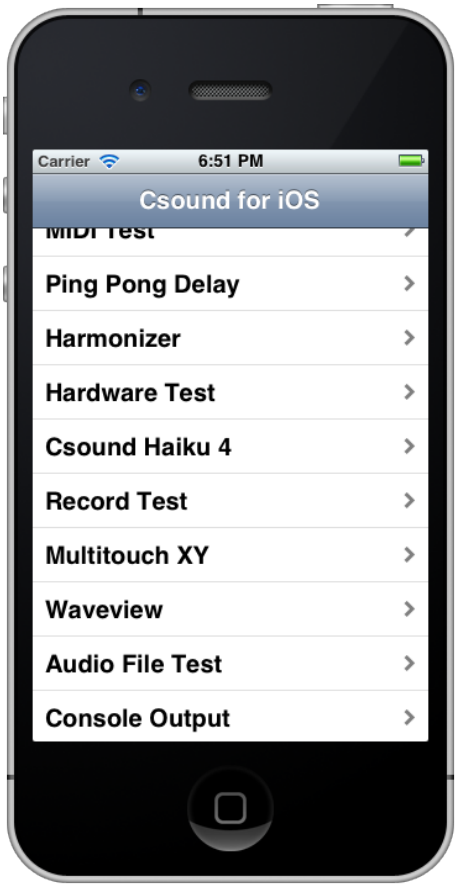

Figure 2. Csound for iOS SDK sample app

(performed by screen or MIDI input), signal processing applications (harmonizer, pitch shifter, ping-pong echo), a generative music example, and other audio-related utilities (fig.2). These examples, together with the manual created for the project, were assembled to assist in learning Csound for iOS.

4. CSOUND FOR ANDROID

Csound for Android is based on a native shared library (libcsoundandroid.so) built using the Android Native Development Kit (NDK)[1], as well as pure Java code for the Android Dalvik compiler. The native library is composed by the object files that are normally used to make up the main Csound library (libcsound), its interfaces extensions (libcsnd), and the external dependency, libsndfile[2]. The Java classes include those commonly found in the csnd.jar library used in standard Java-based Csound development (which wrap libcsound and libcsnd), as well as unique classes created for easing Csound development on Android.

As a consequence of this, those users who are familiar with Csound and Java can transfer their knowledge when working on Android. Developers who learn Csound on Android can take their experience and work on standard Java desktop applications. The two versions of Java do differ, however, in some areas such as classes for accessing hardware and different user interface libraries. Similarly to iOS, in order to help ease development, a CsoundObj class, here written in Java, of course, was developed to provide straightforward solutions for common tasks.

As with iOS, some issues with the Android platform have motivated some internal changes to Csound. One such problem was related to difficulties in handling temporary files by the system. As Csound was dependent on these in the compilation/parsing stage, a modification to use core (memory) files instead of temporary disk files was required.

Two options have been developed for audio IO. The first involves using pure Java code through the AudioTrack API provided by the Android SDK. This is, at present, the standard way of accessing the DAC/ADC, as it appears to provide a slightly better performance on some devices. It employs the blocking mechanism given by AudioTrack to push audio frames to the Csound input buffer (spin) and to retrieve audio frames from the output buffer (spout), sending them to the system sound device. Although low latency is not available in Android, this mechanism works satisfactorily.

As a future low-latency option, we have also developed a native code audio interface. It employs the OpenSL API offered by the Android NDK. It is built as a replacement for the usual Csound IO modules (portaudio, alsa, jack, etc.), using the provided API hooks. It works asynchronously, integrated into the Csound performance cycle. Currently, OpenSL does not offer lower latency than AudioTrack, but this situation might change in the future, so this option has been maintained alongside the pure Java implementation. It is presented as an add-on to the native shared library. Such mechanism will also be used for the future addition of MIDI IO (replacing the portmidi, alsamidi, etc. modules available in the standard platforms), in a similar manner to the present iOS implementation.

At the outset of the development of Csound for Android, a choice was made to port the CsoundObj API from Objective-C to Java. The implementation of audio handling was done so in a manner following the general design as implemented on iOS (although, internally, the current implementations differ in that iOS employs an asynchronous mechanism, whereas in Android blocking IO is used). Also, the APIs match each other as much as possible, including class and method names. There were inevitable differences, resulting primarily from what hardware sensors were available and lack of a standard MIDI library on Android. However, the overall similarities in the APIs greatly simplified the porting of example applications from iOS to Android. For application developers using MCP, the parity in APIs means an easy migration path when moving projects from one platform to the other.

[1] http://developer.android.com/sdk/ndk/index.html
[2] http://www.mega-nerd.com/libsndfile/

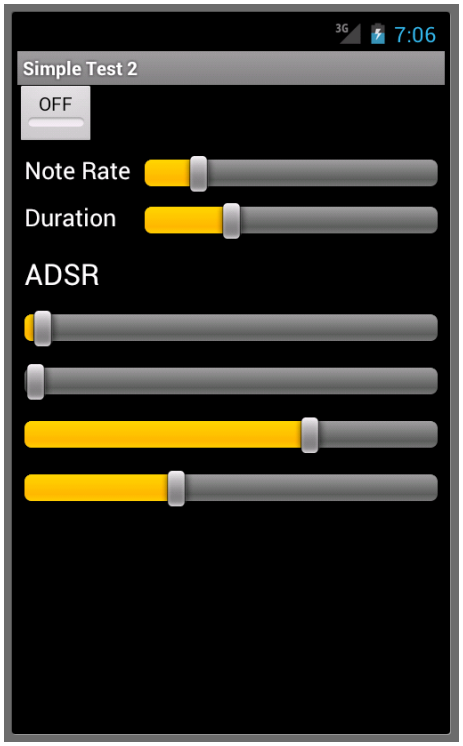

Figure 3. Csound for Android SDK example

5. CSOUND FOR JAVA WEB START

Csound 5 has long included a Java wrapper API that is used by desktop applications such as *AVSynthesis* and *blue*. During research for a music-related project that required being deployable over the web, work was done to explore using Java as the technology to handle the requirements of the project, particularly Java Web Start (JAWS). The key difference between ordinary Java desktop and Java Web Start-based applications is that with the former, the Csound library must be installed by the user for the program to function. With the latter, instead, the application will be deployed, downloading the necessary libraries to run Csound without the user having anything installed (besides the Java runtime and plugin).

Regarding security, JAWS allows for certificate-signed Java applications to package and use use native libraries. Typically, JAWS will run an application within a *sandbox* that limits what the application is allowed to do, including things like where files can be written and what data can be read from the user's computer. However, to run with native libraries, JAWS requires use of all permissions, which allows full access to the computer. Applications must still be signed, verifying the authenticity of what is downloaded, and users must still allow permission to run. This level of security was deemed practical and effective enough for the purposes of this research.

In order to keep the native library components to a minimum, JAWS Csound only requires the Csound core code (and soundfile access through libsndfile, which is packaged with it). Audio IO is provided by the Java-Sound library, which is a standard part of modern Java runtime environments. JAWS Csound has been chosen as the sound engine for the DSP eartraining online course being developed at the Norwegian University of Science and Technology [2].

Figure 4. Csound for JAWS example

6. CSOUND 6

In February 2012, the final feature release of Csound 5 was launched (5.16) with the introduction of a new bison/flex-based orchestra parser as default. The development team has now embarked on the development of the next major upgrade of the system, Csound 6. The existence of projects such as the MCP will play an important part in informing these new developments. One of the goals for the new version is to provide more flexibility in the use of Csound as a synthesis engine by various applications. This is certainly going to be influenced by the experience with MCP. Major planned changes for the system will include:

- Separation of parsing and performance
- Loading/unloading of instrument definitions
- Further support for parallelisation

As Csound 6 is developed, it is likely that new versions of the MCP projects will be released, in tandem with changes in the system.

7. CONCLUSIONS

The Mobile Csound Platform has been developed to bring Csound to popular mobile device operating systems. Work was done to build an idiomatic, object-oriented API for both iOS and Android, implemented using their native languages (Objective-C and Java respectively). Work was also done to enable Csound-based applications to be deployed over the internet via Java Web Start. By porting Csound to these platforms, Csound as a whole has moved from embracing usage on the desktop to become pervasively available. The MCP, including all the source code for the SDK, and technical documentation, is available for download from

http://sourceforge.net/projects/csound/files/csound5

For the future, it is expected that current work on Csound 6 will help to open up more possibilities for music application development. Developments such as real-time orchestra modification within Csound should allow for more

flexibility in kinds of applications that are possible to develop. As mobile hardware continues to increase in number of cores and multimedia capabilities, Csound will continue to grow and support these developments as first-class platforms.

8. ACKNOWLEDGEMENTS

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.

9. REFERENCES

[1] R. Boulanger, Ed., *The Csound Book*. Cambridge, Mass: MIT Press, 2000.

[2] O. Brandtsegg, J. Inderberg, H. Kvidal, V. Lazzarini, J. Rudi, S. Saue, A.Tidemann, N. Thelle, and J. Tro, "Developing an online course in dsp eartraining," *submitted to DAFx 2012*, 2012.

[3] J. ffitch, "On the design of csound 5," in *Proceedings of 4th Linux Audio Developers Conference*, Karlsruhe, Germany, 2006, pp. 79–85.

[4] V. Lazzarini, "Scripting csound 5," in *Proceedings of 4th Linux Audio Developers Conference*, Karlsruhe, Germany, 2006, pp. 73–78.

[5] ——, "A toolkit for audio and music applications in the xo computer," in *Proc. of the International Computer Music Conference 2008*, Belfast, Northern Ireland, 2008, pp. 62–65.

[6] V. Lazzarini and J. Piche, "Cecilia and tclcsound," in *Proc. of the 9th Int. Conf. on Digital Audio Effects (DAFX)*, Montreal, Canada, 2006, pp. 315–318.

[7] V. Lazzarini and R. Walsh, "Developing ladspa plugins with csound," in *Proceedings of 5th Linux Audio Developers Conference*, Berlin, Germany, 2007, pp. 30–36.

[8] J. Piche and A. Burton, "Cecilia: a production interface for csound," *Computer Music Journal*, vol. 22, no. 2, pp. 52–55, 1998.

Appendix B

Csound for Android

Original Publication:

Steven Yi and Victor Lazzarini. Csound for Android. In *Linux Audio Conference*, volume 6, 2012.

Csound for Android

Steven YI and **Victor LAZZARINI**
National University of Ireland, Maynooth
{steven.yi.2012, victor.lazzarini}@nuim.ie

Abstract

The Csound computer music synthesis system has grown from its roots in 1986 on desktop Unix systems to today's many different desktop and embedded operating systems. With the growing popularity of the Linux-based Android operating system, Csound has been ported to this vibrant mobile platform. This paper will discuss using the Csound for Android platform, use cases, and possible future explorations.

Keywords

Csound, Android, Cross-Platform, Linux

1 Introduction

Csound is a computer music language of the MUSIC N type, developed originally at MIT for UNIX-like operating systems[Boulanger, 2000]. It is Free Software, released under the LGPL. In 2006, a major new version, Csound 5, was released, offering a completely re-engineered software, which is now used as a programming library with its own application programming interface (API). It can now be embedded and integrated into several systems, and it can be used from a variety of programming languages and environments (C/C++, Objective-C, Python, Java, Lua, Pure Data, Lisp, etc.). The API provides full control of Csound compilation and performance, software bus access to its control and audio signals, as well as hooks into various aspects of its internal data representation. Several frontends and composition systems have been developed to take advantage of these features. The Csound API has been described in a number of articles [Lazzarini, 2006], [Lazzarini and Piche, 2006] [Lazzarini and Walsh, 2007].

The increasing popularity of mobile devices for computing (in the form of mobile phones, tablets and netbooks), has brought to the fore new platforms for Computer Music. Csound has already been featured as the sound engine for one of the pioneer systems, the XO-based computer used in the One Laptop per Child (OLPC) project [Lazzarini, 2008]. This system, based on a Linux kernel with the Sugar user interface, was an excellent example of the possibilities allowed by the re-engineered Csound. It sparked the ideas for a Ubiquitous Csound, which is steadily coming to fruition with a number of parallel projects, collectively called the *Mobile Csound Platform* (MCP). One such project is the development of a software development kit (SDK) for Android platforms, which is embodied by the CsoundObj API, an extension to the underlying Csound 5 API.

Android [1] is a Linux-kernel-based, open-source operating system, which has been deployed on a number of mobile devices (phones and tablets). Although not providing a full GNU/Linux environment, Android nevertheless allows the development of Free software for various uses, one of which is audio and music. It is a platform with some good potential for musical applications, although at the moment, it has a severe problem for realtime use that is brought by a lack of support for low-latency audio.

In this article we will discuss Csound usage on Android. We will explore the CsoundObj API that has been created to ease developing Android applications with Csound, as well as demonstrate some use cases. Finally, we will look at what Csound uniquely brings to Android, with a look at the global Csound ecosystem and how mobile apps can be integrated into it.

[1] `http://www.android.com`

2 Csound for Android

The Csound for Android platform is made up of a native shared library (libCsoundandroid.so) built using the Android Native Development Kit (NDK)[2], as well as Java classes that are compilable with the more commonly used Android Dalvik compiler. The native library is linked using the the object files that are normally used to make up the libcsound, libcsnd, and libsndfile[3] libraries that are found part of the desktop version of Csound. The Java classes include those commonly found in the csnd.jar library used for desktop Java-based Csound development, as well as unique classes created for easing Csound development on Android.

The SWIG[4] wrapping used for Android contains all of the same classes as those used in the Java wrapping that is used for desktop Java development with Csound. Consequently, those users who are familiar with Csound and Java can transfer their knowledge when working on Android, and users who learn Csound development on Android can take their experience and work on desktop Java applications. However, the two platforms do differ in some areas such as classes for accessing hardware and different user interface libraries. To help ease development, a CsoundObj class was developed to provide out-of-the-box solutions for common tasks (such as routing audio from Csound to hardware output). Also, applications using CsoundObj can be more easily ported to other platforms where CsoundObj is implemented (i.e. iOS).[5]

One of the first issues arising in the development of Csound for Android was the question of plugin modules. Since the first release of Csound 5, the bulk of its unit generators (opcodes) were provided as dynamically-loaded libraries, which resided in a special location (the OPCODEDIR or OPCODEDIR64 directories) and were loaded by Csound at the orchestra compilation stage. However, due to the uncertain situation regarding dynamic libraries (not only in Android but also in other mobile platforms), it was decided that all modules without any dependencies or licensing issues could be moved to the main Csound library code. This was a major change (in Csound 5.15), which made the majority of opcodes part of the base system, about 1,500 of them, with the remaining 400 or so being left in plugin modules. The present release of Csound for Android includes only the internal unit generators. Another major internal change to Csound, which was needed to facilitate development for Android, was the move to use core (memory) files instead of temporary disk files in orchestra and score parsing.

Audio IO has been developed in two fronts: using pure Java code through the AudioTrack API provided by the Android SDK and, using C code, as a Csound IO module that uses the OpenSL API that is offered by the Android NDK. The latter was developed as a possible window into a future lower-latency mode, which is not available at the moment. It is built as a replacement for the usual Csound IO modules (PortAudio, ALSA, JACK, etc.), using the provided API hooks. The Csound input and output functions, called synchronously in its performance loop, pass a buffer of audio samples to the DAC/ADC using the OpenSL enqueue mechanism. This includes a callback that is used to notify when a new buffer needs to be enqueued. A double buffer is used, so that while one half is being written or read by Csound, the other is enqueued to be consumed or filled by the device. The code fragment below in listing 1 shows the output function and its associated callback. The OpenSL module is the default mode of IO in Csound for Android. Although it does not currently offer low-latency, it is a more efficient means of passing data to the audio device and it operates outside the influence of the Dalvik virtual machine garbage collector (which executes the Java application code).

The AudioTrack code offers an alternative means accessing the device. It pushes/retrieves input/output frames into/from the main processing buffers (spin/spout) of Csound synchronously at control cycle intervals. It is offered as an option to developers, which can be used for instance, in older versions of Android without OpenSL support.

[2] http://developer.android.com/sdk/ndk/index.html

[3] http://www.mega-nerd.com/libsndfile/

[4] http://www.swig.org

[5] There are plans to create CsoundObj implementations for other object-oriented desktop development languages/-platforms such as C++, Objective-C, Java, and Python, but at the time of this writing, CsoundObj is only available in Objective-C for iOS.

3 Application Development using CsoundObj

Developers using the CsoundObj API will essentially partition their codebase into three parts: application code, audio code, and glue code. The application code contains the standard Android code for creating applications, including such things as view controllers, views, database handling, and application logic. The audio code is a standard Csound CSD project that contains code written in Csound and will be run using a CsoundObj object. Finally, the glue code is what will bridge the user interface with Csound.

```
/* this callback handler is called every time a buffer finishes playing */
void bqPlayerCallback(SLAndroidSimpleBufferQueueItf bq, void *context)
{
   open_sl_params *params = (open_sl_params *) context;
   params->csound->NotifyThreadLock(params->clientLockOut);
}

/* put samples to DAC */
void androidrtplay_(CSOUND *csound, const MYFLT *buffer, int nbytes)
{
    open_sl_params *params;
    int     i = 0, samples = nbytes / (int) sizeof(MYFLT);
    short* openslBuffer;

    params = (open_sl_params *) *(csound->GetRtPlayUserData(csound));
    openslBuffer = params->outputBuffer[params->currentOutputBuffer];
    if (params == NULL)
      return;
    do {
      /* fill one of the double buffer halves */
      openslBuffer[params->currentOutputIndex++] = (short) (buffer[i]*CONV16BIT);
      if (params->currentOutputIndex >= params->outBufSamples) {
         /* wait for notification */
          csound->WaitThreadLock(params->clientLockOut, (size_t) 1000);
         /* enqueue audio data */
         (*params->bqPlayerBufferQueue)->Enqueue(params->bqPlayerBufferQueue,
                          openslBuffer,params->outBufSamples*sizeof(short));
         /* switch double buffer half */
         params->currentOutputBuffer = (params->currentOutputBuffer ?  0 : 1);
         params->currentOutputIndex = 0;
         openslBuffer = params->outputBuffer[params->currentOutputBuffer];
      }
    } while (++i < samples);
}
```

Listing 1: OpenSL module output C function and associated callback

```
public interface CsoundValueCacheable {
  public void setup(CsoundObj csoundObj);
  public void updateValuesToCsound();
  public void updateValuesFromCsound();
  public void cleanup();
}
```

Listing 2: CsoundValueCacheable Interface

```
String csd = getResourceFileAsString(R.raw.test);
File f = createTempFile(csd);
csoundObj.addSlider(fSlider, "slider", 0.0, 1.0);
csoundObj.startCsound(f);
```

Listing 3: Example CsoundObj usage

CsoundObj uses objects that implement the CsoundValueCacheable interface for reading value from and writing values to Csound (listing 2). Any number of cacheables can be used with CsoundObj. The design is flexible enough such that you can design your application to use one cacheable per user interface or hardware sensor element, or one can make a cacheable that reads and writes along many channels.

CsoundObj contains utility methods for binding Android Buttons and SeekBars to a Csound channel, as well as for a method for binding the hardware Accelerometer to preset Csound channels. These methods wrap the View or sensor objects with pre-made CsoundValueCacheables that come with the CsoundObj API. Since these are commonly used items that would be bound, the utility methods were added to CsoundObj as a built-in convenience to those using the API. Note that CsoundValueCacheables are run within the context of the audio processing thread; this was done intentionally so that the cacheable could copy any values it needed to from Csound, then continue to do processing in another thread and eventually post back to the main UI thread via a Handler.

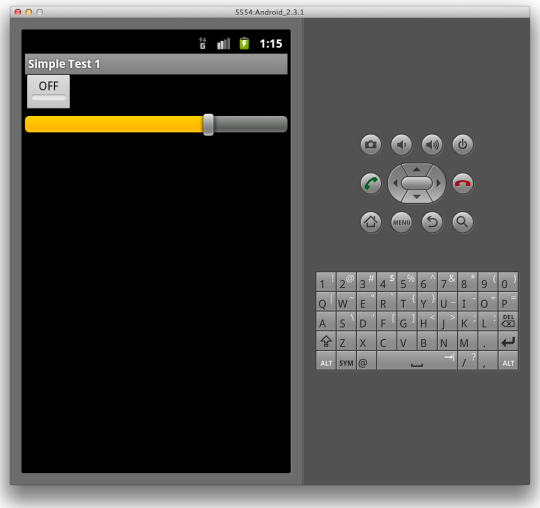

Figure 1: Android Emulator showing Simple Test 1 Activity

Listing 3 shows example code of using CsoundObj with a single slider, from the Simple Test 1 Activity, shown in Figure 1.The code above shows how a CSD file is read from the projects resources using the getResourceFileAsString utility method, saved as a temporary file, then used as an argument to CsoundObj's startCsound method. The 2nd to last line shows the addSlider method being used to bind fslider, an instance of a SeekBar, to Csound with a channel name of "slider" and a range from 0.0 to 1.0. When Csound is started, the values from that SeekBar will be read by the Csound project using the chnget opcode, which will be reading from the "slider" channel.

Figure 2 shows the relationships between different parts of the platform and different usage scenarios. An application may work with CsoundObj alone if they are only going to be starting and stopping a CSD. The application may also use CsoundValueCacheables for reading and writing values from either CsoundObj or the CsoundObject. Finally, an application may do additional interaction with the Csound object that the CsoundObj has as its member, taking advantage of the standard Csound API.

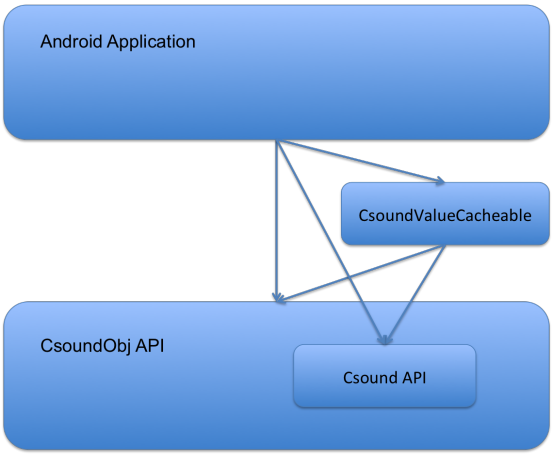

Figure 2: CsoundObj Usage Diagram

A Csound for Android examples project has been created the contains a number of different Csound example applications. These examples demonstrate different ways of using the CsoundObj API as well as different approaches to applications, such as realtime synthesis instruments and generative music. The examples were ported over from the Csound for iOS examples project and users can study the code to better understand both the CsoundObj API on Android as well as what is required to do cross-platform

development with Csound as an audio platform.

4 Benefits of using Csound on Android

Using Csound on Android provides many benefits. First, Csound contains one of the largest libraries of synthesis and signal processing routines. By leveraging what is available in Csound, the developer can spend more time working on the user interface and application code and rely on the Csound library for audio-related programming. The Csound code library is also tested and supported by a open-source community, meaning less testing work required for your project.

In addition to the productivity gain of using a library for audio, Csound projects–developed in text files with .csd extensions–can be developed on the desktop, and later moved to the Android application. Developing and testing on the desktop allows for a faster development process than testing in the Android emulator or on a device, as it removes the application compilation and deployment stage, which can be slow at times.

Having the audio-related code in a CSD file for a project also brings with it two benefits. First, development of an application can be split amongst multiple people; one can work on the audio code while the other focuses on developing other areas of the application. Second, developing an application based around Csound allows for moving that CSD to other platforms, such as iOS or desktop operating systems. The developer would then only have to develop the user-interface and glue code to work with that CSD on each platform.

Additionally, cleanly separating out the audio system of an application and enforcing a strict API (Application Programmer Interface) to that system is a good practice for application development. This helps to prevent tangled, hard to maintain code. This is of benefit to the beginning and advanced programmer alike.

5 Conclusions

From its roots in the Music N family of programs, Csound has grown over the years, continually expanding it features as a synthesis library as well as its usefulness as a music platform. With its availability on multiple operating systems, Csound offers a multi-platform option for developing musical applications. Current Csound 6 developments to enable realtime modification of the processing graph as well as other features will expand the types of applications that can be built with Csound. As Android is now supported within the core Csound repository, it will continue to be developed as a primary platform for deployment as part of the MCP distribution.

6 Availability

The Csound for Android platform and examples project are included in the main Csound GIT repository. Build files are included for those interested in building Csound with the Android Native Development Kit. Archives including a pre-compiled Csound as well as examples are available at `http://sourceforge.net/projects/csound/files/csound5/Android/`.

7 Acknowledgements

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.

References

R. Boulanger, editor. 2000. *The Csound Book*. MIT Press, Cambridge, Mass.

V Lazzarini and J. Piche. 2006. Cecilia and tclcsound. In *Proc. of the 9th Int. Conf. on Digital Audio Effects (DAFX)*, pages 315–318, Montreal, Canada.

V Lazzarini and R. Walsh. 2007. Developing ladspa plugins with csound. In *Proceedings of 5th Linux Audio Developers Conference*, pages 30–36, Berlin, Germany.

V Lazzarini. 2006. Scripting csound 5. In *Proceedings of 4th Linux Audio Developers Conference*, pages 73–78, Karlsruhe, Germany.

V Lazzarini. 2008. A toolkit for audio and music applications in the xo computer. In *Proc. of the International Computer Music Conference 2008*, pages 62–65, Belfast, Northern Ireland.

Appendix C

Csound 6: old code renewed

Original Publication:

John ffitch, Victor Lazzarini, and Steven Yi. Csound6: old code renewed. In *Linux Audio Conference 2013*, Graz, Austria, April 2013.

Csound6: old code renewed

John FITCH and **Victor LAZZARINI** and **Steven YI**
Department of Music
National University of Ireland
Maynooth,
Ireland,
{jpff@codemist.co.uk, victor.lazzarini@nuim.ie, stevenyi@gmail.com}

Abstract

This paper describes the current status of the development of a new major version of Csound. We begin by introducing the software and its historical significance. We then detail the important aspects of Csound 5 and the motivation for version 6. Following, this we discuss the changes to the software that have already been implemented. The final section explores the expected developments prior to the first release and the planned additions that will be coming on stream in later updates of the system.

Keywords

Music Programming Languages, Sound Synthesis, Audio Signal Processing

1 Introduction

In March 2012, a decision was taken to move the development of Csound from version 5 to a new major version, 6. This meant that most of the major changes and improvements to the software would cease to be made in Csound 5, and while new versions would be released, these will consist mainly of bug fixes and minor changes (possibly including new opcodes). Moving to a new version allowed developers to rethink key aspects of the system, without the requirement of keeping ABI or API compatibility with earlier iterations. The only restriction, which is a fundamental one for Csound, is to provide backwards language compatibility, ensuring that music composed with the software will continue to be preserved.

This paper describes the motivation for the changes, current state of development and prospective plans for the system.

1.1 Short History of Csound

1.1.1 Early History

Csound has had a long history of development, which can be traced back to Barry Vercoe's MUSIC 360[Vercoe, 1973] package for computer music, which was itself a variant of Max Mathews' and Joan Miller's MUSIC IV[Mathews and Miller, 1964]. Following the introduction of the PDP-11 minicomputer, a modified version of the software appeared as MUSIC 11[Vercoe, 1981]. Later, with the availability of C (and UNIX), this program was re-written in that language as Csound[Boulanger, 2000], allowing a simpler cycle of development and portability, in comparison to its predecessor.

The system, in its first released version, embodied a largely successful attempt at providing a cross-platform program for sound synthesis and signal processing. Csound was then adopted by a large development community in the mid 90s, after being translated into the ANSI C standard by John ffitch in the early half of the decade. In the early 2000s, the final releases of version 4 attempted to retrofit an application programming interface (API), so that the system could be used as a library.

1.1.2 Csound 5

The need for the further development of the Csound API, as well as other innovations, prompted a code freeze and a complete overhaul of the system into version 5[ffitch, 2005]. Much of this development included updating 1970s programming practices by applying more modern standards. One of the major aims was to make the code reentrant, so that its use as a library could be made more robust. In 2006, version 5.00 was released. The developments embodied by this and subsequent releases allowed a varied use of the software, with a number of third-party projects benefitting from them.

1.2 Csound operation in a nutshell

As a MUSIC-N language, Csound incorporates a compiler for instruments. During performance, these can be activated (instantiated) by various means, the traditional one being the standard numeric score. In Csound 5, compilation can only be done once per performance run, so new instruments cannot be added to an already running engine (for this performance

needs to be interrupted so the compilation can take place).

The steps involved in the compiler can be divided into two: parsing, and compilation proper. The first creates an abstract syntax tree (AST) representing the instruments. The compilation then creates data structures in memory that correspond to the AST. When an instrument is instantiated, an init-pass loop is performed, executing all the once-off operations for that instance. This is then inserted in a list of active instruments, and its performance code is executed sequentially, processing vectors (audio signals), scalars (control signals) or frames of spectral data. The list orders instruments by ascending number, so higher-order ones will always be executed last. All of the key aspects of Csound operation are exposed by the API.

2 Motivation

In the six years since its release, Csound 5 continued to develop in many ways, mostly in response to user needs, as well as providing further processing capabilities in the form of new opcodes. After a long gestation, early in 2012, the new flex-bison parser was completed and added as a standard option. This was the final major step of development for Csound, where the last big chunk of 1970s code, the old ad-hoc parser, was replaced by a modern, maintainable, and extendable parser. Following the 2011 Csound Conference in Hannover, it was clear that there were a number of user requests that would be more easily achievable with a rethink of the system. Such suggestions included:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine
- additions to the orchestra language, for instance, generic arrays
- rationalisation of the API to allow further features in frontends
- loadable binary formats, API construction of instruments
- further development of parallelism
- facilities for live coding

The time was ripe for major changes to be made. User suggestions prompted developers to begin an internal cleanup of code, the removal of older components (such as the old parser), and a reorganisation of the API. It was also an opportunity to code-walk, and with that find inconsistencies and bugs that would normally be hidden. In particular, changes related to repeated loading and compilation of new instruments would require (and indeed force) a welcome separation of language and synthesis engine, which is well underway at present.

3 Developments to date

3.1 Build System and Tests

In Csound 5, the official build system is SCons[1]. Over time, a CMake-based[2] build was introduced and used for local developer use, as well as later for Debian packaging and iOS builds. In Csound 6, the official build system is now the CMake-based build. Moving to CMake introduced some hurdles and changes in workflow, but it also brought with it generation of build system files, such as Makefiles, XCode projects, and Eclipse projects. This solved a problem of IDE-based projects for building Csound becoming out of sync with changes in the SConstruct file for SCons, as well as brought more ways for developers to approach building and working with Csound code, particularly through IDE's.

Using the CTest feature in CMake, unit and functional tests have been added to Csound 6's codebase. CTest is the test running utility used to execute the individual C-code tests. In addition, CUnit[3] is employed to create the individual tests and test-suites within the test code files. In addition to C-code testing, the suite of CSD's used for application/integration testing continues to grow, and a new set of Python tests has also been added for testing API usage from a host language.

3.2 Code reorganisation

The Csound code base is passing through a significant reorganisation. Firstly, parts of it that are now obsolete, such as the old parser, have been removed. Some opcodes with special licensing conditions that have been deemed not to be conducive to further development have been completely rewritten (also with some efficiency and generality improvements). The CSOUND struct has been rationalised and reorganised, with many modifications due to the various changes outlined in the next sections.

[1] http://www.scons.org
[2] http://www.cmake.org
[3] http://cunit.sourceforge.net

Finally, the public API is going through a redesign process (details of which are discussed below).

3.3 Type system

The Csound Orchestra language uses strongly typed variables and enforces these at compile-time. This type information is used to determine the size of memory to allocate for a variable as well as for specifying the in- and out-arg types for opcodes. The system of types used prior to Csound 6 was hard-coded into the parser and compiler. Adding new types would require adding code in many places.

In Csound 6, a generic type system was implemented as well as tracking of variable names to types. The new system provides a mechanism to create and handle types, such that new types can be easily added to the language. The system also helps clarify how types are used during compilation. Another feature is that variable definitions and types were previously discarded after compile-time; in Csound 6, this information is kept after compilation. This allows the possibility of inspecting variables found in instruments or in the global memory space.

3.4 Generic Arrays

In Csound 5, a 't' type was added that provided a user-definable length, single-dimension array of floating-point numbers. In Csound 6, with the introduction of the generic type system, the code for t-types was extended to allow creation of homogenous, multi-dimensional arrays of any type. Additionally, the argument list specification for opcodes was extended to allow denoting arrays as arguments.

3.5 On-the-fly Compilation

The steps necessary for the replacement or addition of new instruments or UDOs to a running Csound engine, or, more concisely, on-the-fly compilation, started to be taken in the latter versions of Csound 5. It was, of course, sine-qua-non to have a properly structured parser, which we did in 5.17. Also, as a side-effect from the Csound for Android project, compilation from text files was replaced by a new core (memory) file subsystem, so now strings containing Csound code could be presented directly to the parser.

The first step in Csound 6 was made by breaking down the monolithic API call to compile Csound (`csoundCompile()`) into `csoundParseOrc()` and `csoundCompileTree()`, as well as by the addition of a general `csoundStart()` function to get the engine going. The parsing function creates an abstract syntax tree (AST) from a string containing Csound code. The compilation function then creates the internal data structures that the AST represents, ready for engine instantiation(see figure 1).

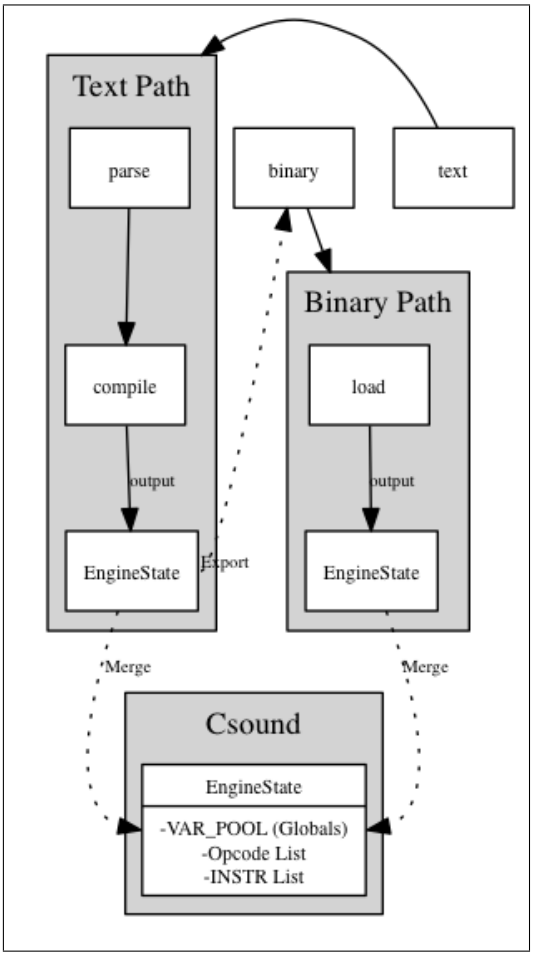

Figure 1: Csound compilation and engineState.

These modifications provided the infrastructure for changes in the code to allow repeated compilation. For this, we have abstracted the data objects relating to instrument definition into an engineState structure. On first compilation, Csound creates its global instrument 0, which is made up of the header statements, global variables declared outside instruments and their initialisation. It then proceeds to compile any other instruments defined in the orchestra (including UDOs, which are a special kind of instrument). On any subsequent compilations, instruments other than 0 are added to a newly-created engineState. After compilation, the new engineState is merged into the current one be-

longing to the running Csound object.

Instrument definitions with the same name or number will replace previously existing ones, but any instances of the old definitions that are active are not touched. New instances will use the new definition, and replaced instruments get added to a deadpool for future memory recovery (which will happen once all old instances are deallocated). A similar process applies to UDOs.

Currently, no built-in thread-safety mechanisms have been placed in the API, so hosts are left to make sure compilation calls are not made concurrently to audio processing calls. However, it is envisaged that the final API will provide functions with built-in thread safe as well as ordinary calls.

3.6 Sample-level accuracy

Csound has always allowed sample-level accuracy, a feature present since its MUSIC 11 incarnation. However, a performance penalty was incurred, since the requirement for this was to set the size of the processing block (`ksmps`) to 1 sample. Code can become very inefficient, since there is a single call of an opcode performance function for each sample of output and this is in conflict with caching.

In Csound 6, an alternative sample accuracy method has been introduced. This involves setting an offset into the processing block, which will round the start time of an event to a single sample. Similarly, event durations are also made to be sample accurate, as the last iteration of each processing loop is limited to the correct number of samples (see figure 2). This option is provided with the non-default `--sample-accurate` flag, to preserve backward compatibility.

Tied events[4] are not subject to sample accurate processing as they involve state reuse and are, in its current form, incompatible with the mechanism. Real-time events are also not affect by the process, as event sensing works on a ksmps-to-ksmps basis. Events scheduled to at least one control-cycle ahead can be made to be sample accurate through this mechanism.

The changes needed for this mechanism to work were significant. Each opcode had to be modified to take account of the offset and end position. The scheduler had to be altered so the start of all events was truncated, instead of rounded, to ksmps boundaries, and the calculation of event duration had to be modified. The offset and end position had to be properly defined for each event, as well as set and reset at specific times for each instrument instance.

[4]In Csound, it is possible to have instrument instances that take up a previously-used memory space, which allows the 'tieing' of events, in analogy to slurs in instrumental music

3.7 Realtime priority mode

Csound has been a realtime audio synthesis engine since 1990. However, it was never provided with strict realtime-safe behaviour, even though in practice, it has been used successfully in many realtime applications. Given the multiple applications of Csound, it makes sense to provide separate operation modes for its engine. In Csound 6, we introduce the realtime priority mode, set by the `--realtime` option, which aims to provide better support for realtime safety, with complete asynchronous file access and a separate thread for unit generator initialisation.

3.7.1 Asynchronous file access

For Csound 6, a new lock-free mechanism has been introduced and some key opcodes have been modified to use it when operating in realtime. It uses a circular buffer, employing an interface which had been already present in Csound (used previously only for lock-free realtime audio). It shares the common file IO structure adopted throughout Csound, with a similar, but dedicated interface. For specific file reading/writing requirements, though, as required for instance by diskin, diskin2 or pvsfwrite, the general interface is not suitable. For this case, special opcode-level asynchronous code has been designed.

3.7.2 Unit generator initialisation

Another important modification of the engine in realtime priority mode is the spawning of a separate thread that is responsible for running all of the unit generator initialisation code. This is more commonly known as the 'init-pass', which is separate from synthesis performance ('perf-pass'). In this mode, when an instrument is instantiated, the init-pass code is immediately run in a separate thread. Once this is done, an instrument is allowed to perform. What this does is to prevent any interruption in the synthesis performance due to non-realtime-safe operations in the initialisation code (memory allocation, file opening, etc.). A side-effect of this is that in some situations, an instrument may be prevented to start performing straight away, as

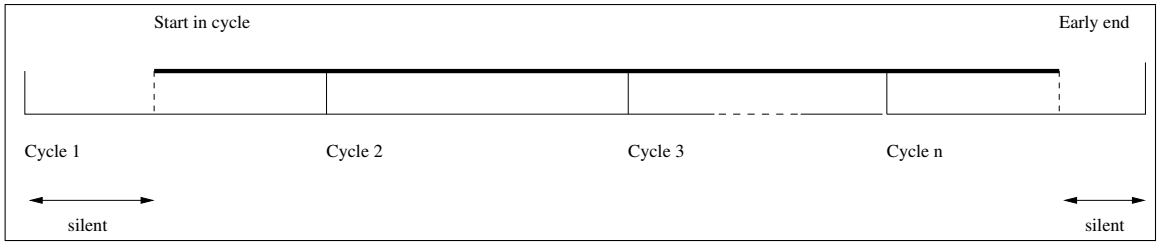

Figure 2: Sample accurate scheme.

the initialisation has not been done. However, this is balanced with the gains in uninterrupted performance.

3.8 Multicore operation

In 2009 an experimental system for using multiple cores for parallel rendering of instruments was written [Wilson, 2009], and this was later incorporated in the standard Csound [ffitch, 2009]. While the design was generally semantically correct it only delivered a performance gains in the case of low control rate and computationally heavy unit generators. Profiling the code showed that the overheads in creating and consuming the directed acyclic graph (DAG) of dependencies, and especially in memory allocation activity.

For Csound 6 we are developing a different approach, that while maintaining the semantic analysis only needs to rebuild the DAG when a score event starts or stops, and in use does not call for changes in the structure. The clue is in the use of watch-lists as found in SAT-solvers [Brown and Purdom Jr, 1982; Eén and Sörensson, 2003]. For each task we only need to watch for the completion of one of the dependencies; when a task finishes it can release any task that is waiting for it, and for which all other precursors have already finished. This strategy is also possible with no locking of critical sections, and can use atomic swap primitives instead.

At the same time some simplification of the semantics-gathering has been achieved. This scheme preserves the order-semantics that Csound has always had, but offers efficient utilisation of multiple cores with threads without user intervention beyond saying how many threads to use for the performance stage. Initial measurements (see table 3.8) are very encouraging, in most cases providing significant speed-up. We are continuing to work on possible optimisations.

4 Further work

4.1 Pre-release prospective development (i.e. the "todo list")

The final feature set of Csound 6 is still not finalised. There are a number of possible enhancements that we are considering; some grow from the changes we have described above, and some are long-standing desires.

The introduction of separate compilation and replaceable instruments naturally suggests that we could add a fast loadable format for instruments, building on for example LISP FASL formats, and API and opcode access to loading. It remains to be seen if the source version is sufficiently fast, and whether we can solve the semantic issues that arise, such as f-table independence. What is needed is to document the abstract syntax tree that the parser produces, and thus allow advocates of alternative orchestra languages to provide them.

A restriction in Csound than has long been an irritation is the limit of one string in a score statement. Previous work in this area has attempted to allow up to four strings, but this is both limiting and still buggy. The radical solution would be to introduce a flex/bison parser for the score language and take the opportunity for rethinking the score area. A small start has been made, but the need to support users and the amount of effort needed here has relegated this work to a later release. Until then a simpler scheme will have to be tried for the interim.

The Csound suite of software include a number of analysis programs, most dating from an early time, and written without regard of floating point formats or byte order. From time to time this has caused problems. The task here is to redefine these formats to indicate at least their formats, or even to make the readers capable of format transformations. This needs to be done at some stage and this break seems like a good moment.

With the introduction of on-the-fly compilation one can consider that a user might main-

-j	CloudStrata	Xanadu		Trapped...		
	ksmps=500 (sr=96000)	ksmps=10	ksmps=100	ksmps=10	ksmps=100	ksmps=1000
1	1	1	1	1	1	1
2	0.54	0.57	0.55	0.75	0.79	0.78
3	0.39	0.40	0.40	0.66	0.76	0.73
4	0.32	0.39	0.33	0.61	0.72	0.70

Table 1: *Relative performance with multiple threads in three existing Csound code examples, -j indicates the number of threads used.*

tain a long-running Csound binary and use it for different tasks at different times. This suggests that the current command-line options or API equivalents may need to change at some time after the initialisation. Some changes may be easy, but some may require re-engineering of parts of the engine. We have not yet realised to use-changes that the compilation change will engender.

The new API still needs to be refined. In response to what has been discussed above, we plan, for instance, to expose the configuration parameters in some form (currently held in the OPARMS data structure). At the moment, there is a simple provision for setting separately specific configuration items in the API (as flags). This is to be substituted by a more flexible form, via the exposing of the OPARMS or an OPARMS-like struct to API users.

A number of other changes are planned, some of which are already present in an early form. For instance, the various stages of parsing, compilation, and engine start are now exposed in the provisional API (as detailed for instance in 3.4). There is a plan to provide built-in thread-safety, so some functions can be used directly in a multi-threading environment without further synchronisation or resource protection. The software bus, which now exists in three forms, will be unified to a single mechanism.

4.2 Future developments

A number of ideas have also been put forward, which will be tackled in due course. These include for instance:

- support for alternative orchestra languages (through access to the parse tree format or some sort of intermediary representation)
- further language features (e.g. namespaces, functions with more than one argument, tuples)
- a system for streaming linear predictive coding processing (in similar fashion to PVOC)
- decoupling of widget opcodes from FLTK dependency (and exposure through API)
- input / output buffer reorganisation (output buffers added to instruments)

5 Conclusions

In this paper, we have sought to examine the current development status of Csound 6, as well as the motivations for the fundamental re-engineering of the code that has been underway. We hope to have demonstrated how the technology embodied in this software package has been renovated continuously in response to developments in Computer Science and Music. Our aim is to continue to support a variety of styles of computer music composition and performance, as well as the various ways in which Csound can be used for application development. It is also important to note, for readers, that the re-engineering of Csound is taking place quite publicly in the Csound 6 git repository on Sourceforge (`git://git.code.sf.net/p/csound/csound6-git`). Anyone is welcome to check out and examine our struggles with computer technology and the solutions we are putting forward in this paper.

6 Acknowledgements

Our thanks go to the Csound community for their indulgence, suggestions and support. In addition Martin Brain introduced the idea of watch-lists and co-developed the detailed performance algorithm. We also acknowledge the implicit support from Sourceforge hosting

References

Richard J. Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design.* MIT Press, February.

Cynthia A. Brown and Paul Walton Purdom Jr. 1982. An Empirical Comparison of Backtracking Algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(3).

Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, number 2919 in Lecture Notes in Computer Science, pages 333–336. Springer-Verlag, May.

John ffitch. 2005. The Design of Csound5. In *LAC2005*, pages 37–41, Karlsruhe, Germany, April. Zentrum für Kunst und Medientechnologie.

John ffitch. 2009. Parallel Execution of Csound. In *Proceedings of ICMC 2009*, Montreal. ICMA.

M. Mathews and J. E. Miller. 1964. *MUSIC IV Programmer's Manual*. Bell Telephone Labs.

B. Vercoe. 1973. *Reference manual for the MUSIC 360 language for digital sound synthesis*. Studio for Experimental Music, MIT.

B. Vercoe. 1981. *MUSIC 11 Reference Manual*. Studio for Experimental Music, MIT.

Christopher Wilson. 2009. Csound Parallelism. Technical Report CSBU-2009-07, Department of Computer Science, University of Bath.

Appendix D

The New Developments in Csound 6

Original Publication:

John ffitch, Victor Lazzarini, Steven Yi, Michael Gogins, and Andres Cabrera. The New Developments in Csound 6. In *Proceedings of ICMC 2013*, Perth, 2013. ICMA.

The New Developments in Csound 6

John ffitch
University of Bath
Department of Computer Science

Victor Lazzarini **Steven Yi**
National University of Ireland, Maynooth
Department of Music

Michael Gogins
Irreducible Productions
New York

Andrés Cabrera
University of California, Santa Barbara
Media Arts and Technology

ABSTRACT

In this paper we introduce a major new version of Csound, the audio processing system and library. We begin with an overview of the current status of Csound (version 5), as well as its three layers of use (high, middle, and low). We then outline the design motivations and features of version 6. We continue by exploring external changes and discussing some examples of use. We conclude by looking forward to the next steps in the development of Csound.

1. INTRODUCTION

In 2012, six years after the initial release of the first major re-engineering of the well-known and widely-used software sound synthesis system Csound [1], we (its developers) decided to embark on a further revision of many of its internal and external aspects. Developments since version 5.00 [2] until the current release, 5.19, have been mostly incremental. They have also been limited by our commitment to maintaining both binary and API (Application Programming Interface) compatibility with earlier versions (although the system has actually come through a binary upgrade, after version 5.09). To allow for a number of requested changes, we decided a new major version was necessary, which would mean a break in backwards compatibility (both API and binary). This does not, however, mean a break in backwards compatibility of Csound code and pieces. Older pieces and code will always continue to work with Csound 6. This paper discusses the motivation for Csound 6, its development process, and major features of the new system.

2. WHAT IS CSOUND?

For the ICMC audience, it might not seem necessary to describe such a well-known and established software package. After all, there have been a number of papers on the subject of Csound presented here, over the years [3] [4] [5] [6] [7] [8] [9]. However, it is well worth describing what Csound is in a bit more detail, because 1) Csound has a long history of development, and much of the information describing it is outdated; and 2) the motivation for the present directions will become clearer as we outline the present system.

The best way to describe Csound, in its version 5, is to present it as a series of layers, with various 'modes of entry' for users and for related applications.

At the lowest level, Csound is a self-contained audio programming language implemented in a cross-platform library, with a well-defined API, which allows software developers to create programs for audio synthesis and processing, and computer music composition. Csound supports a variety of synthesis techniques in its orchestra language, and allows various means/levels of internal and external control. Csound is extensible via plugin modules. Software that uses Csound can be written in C, C++, Objective-C, Java, Python, Lua, Tcl, Lisp, and others. Csound runs on Windows, Linux, OSX, Solaris, Haiku, Android and iOS.

The middle layer is characterized by writing programs in the Csound language for performance, composition, and other audio processing tasks such as sonification. At this level, the system allows composers to design computer music instruments, and to control them in real time or deferred time. Interaction with the system comes via various frontends, many of which are third party (i.e. not maintained as part of the Csound releases). The 'classic' command-line interface (CLI) is the basic frontend, where the system is controlled by a single terminal command. As this was the only original means of using the software, traditionally a number of frontends have been designed to provide a simpler wrapper around CLI Csound. More commonly, today, frontends access the Csound library directly (via its API). These frontends provide diverse modes of interaction. For example, Csound can be embedded in graphical environments such as Pure Data via the *csoundapi~* frontend, and in Max/MSP via *csound~*. Composition environments such as *blue* use it as a sound engine. For more general-purpose uses, there are integrated development environments (IDEs) for programming with Csound (such as *CsoundQt* and *WinXsound*), and plugin/application generators, such as *Cs-LADSPA* [6] and *Cabbage*[8].

At this level, Csound co-exists with a number of tools and languages which add support for activities such as algorithmic composition and graphical user interaction.

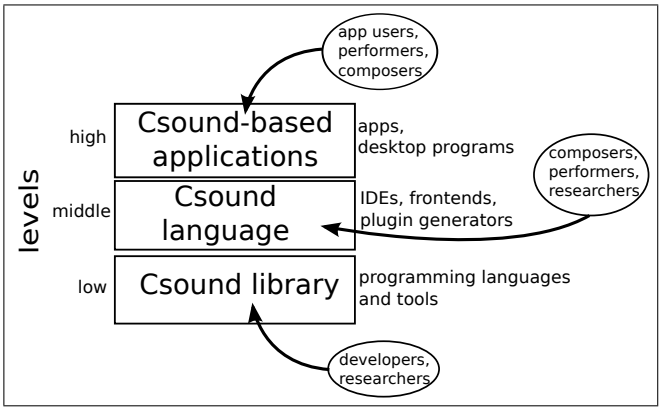

Figure 1. The various levels of Csound, related software and end users.

At the highest level, use of Csound occurs by developing applications based on the middle and lower levels. Here, the user might not even know that Csound is involved, as user programming is generally not involved. This is seen, for instance, in some frontends, such as Cecilia and blue, where the user might only need to deal with parameter setting in the graphical interface, in plugins or applications generated by Cabbage, in bundled packages such as Csound4Live (which uses the csound~ frontend), or in mobile applications for iOS and Android [9].

It is clear that Csound has attracted a diverse set of users, from the expert programmer to the mobile app customer. In addition, thanks to Csound's long history, especially by composers working at the middle level, there is a legacy of music written with Csound that is worth preserving (and that in fact stretches back to 70s compositions written for Csound's predecessors MUSIC 11 and MUSIC 360 [10]). This has focused our minds to provide a completely backwards-compatible system (as far as the language is concerned) as *sine qua non* condition for future versions.

Some may criticise this as recipe for an ever increasing language, with its associated complexity penalty, and, as often vocalised in the detractors' corner, 'bloat.' But although some 'bloat' is inevitable in a system nearly three decades old, the Csound language is still syntactically very simple (it consists of just a few simple syntactical constructs), and the processing engine is generally efficient in terms of DSP and algorithm implementations. What criticisms fail to consider is that Csound has fostered a vibrant community of users and developers.

We understand that *community* is the biggest asset a system like Csound can have. Without users, expert and non-expert, a system withers and dies. It would not be a huge task (in comparative terms) to ditch the old system and recreate one whose language adapts completely to the flavour-of-the-moment software design. Also, creating a whole new computer music language from scratch is also not too difficult now, especially with the availability of models that exist as open-source code. In fact, there is a multiplication of incipient systems that claim to be the intelligent solution to perceived problems in existing software. The majority of these do not cross the 80/20 divide of development. This occurs possibly for a variety of reasons, but especially for the lack of an enthusiastic user (and developer) community. There is great value in the accumulated knowledge of the community and the large body of existing code. We understand that moving Csound away from its origins as a system does not mean ditching users and music along the way. The requirements of the community are paramount to where we want the software to go. By supporting the various levels of entry into the system, we aim to foster interest in the software and in computer music in general. This translates as well into the different levels of difficulty that the Csound language contains. It allows educators to provide a smooth learning curve for students, going from the early (and simpler) set of language elements into the expanded one that the system supports today.

3. WHY CSOUND 6?

By 2012, we began to feel that Csound 5's incremental model of development was becoming a limitation. At the 2011 International Csound Conference in Hannover, users and developers met to agree on a number of desired features that the software should have in future versions. Some of these (like support for mobile platforms and some additional language features) were achievable in Csound 5 and indeed were soon made available. Others have required a major re-engineering of the system. Among them, we can cite:

- the capacity of new orchestra code, ie. instruments and user-defined opcodes (UDOs), to be added to a running instance of the engine (enhancing, for instance, live-coding support and interactive sound design);
- major additions to the orchestra language, for instance, generic arrays, debugging/introspection, and a type system;
- rationalisation of the API to simplify its usage and to allow further features in frontends;
- fast loadable (FASL-like) binary formats, API construction of instruments;
- further development of concurrency (enhancement of existing support [7]).

This list was our starting point for the development of Csound 6.

4. INTERNAL CHANGES IN VERSION 6.0

A number of important changes have been made to the code base, which not only introduce significant improvements and scalability in performance (ie. in parallel processing), but also provide a robust infrastructure for future developments

Threads	CloudStrata	Xanadu		Trapped In Convert		
	ksmps=500 (sr=96000)	ksmps=10	ksmps=100	ksmps=10	ksmps=100	ksmps=1000
1	1	1	1	1	1	1
2	0.54	0.57	0.55	0.75	0.79	0.78
3	0.39	0.40	0.40	0.66	0.76	0.73
4	0.32	0.39	0.33	0.61	0.72	0.70

Table 1. *Relative performance with multiple threads in three existing Csound code examples.*

4.1 Build system and tests

We have adopted CMake as Csound's primary build tool, replacing scons as used in Csound5. We have added test suites for the language and API, as a well as individual CUnit tests, to the code base and build system. These changes are well aligned with modern standards of software testing and project development.

4.2 Code reorganisation

We have removed obsolete code, such as the old parser. The CSOUND class has been rationalised and refactored. Some opcodes have been rewritten/substituted, especially in cases where they incorporated special licensing issues beyond LGPL. Syntax checking in the parser has been completely overhauled, and, by extension, the old annotation system used for opcode overloading has been substituted by a simpler and more robust mechanism. Data-structure utilities such as hash tables etc have been given a clean and easy to maintain interface and implementation.

4.3 Type system

To better support the strong typing of the Csound language and also to allow its expansion, we have implemented a new type system to replace the old hard-coded typing in the parser and compiler. This is more generic and implements tracking of variable names to types. The type system allows the creation of opcodes that accept and produce complex data structures, as well as new semantics for opcode inputs and outputs. It will also allow the development of debugging/inspection tools for Csound code. In addition, the code for the string type has been completely replaced, allowing for dynamic allocation and variable sizes. This was required to allow any size orchestra code to be manipulated as strings, and passed to the compilation stage inside a Csound instrument.

4.4 Asynchronous operations

We implemented mechanisms for the access of files in an asynchronous mode (non-blocking). These mechanisms are generic enough for the use in opcodes and plugins. In cases where the generic mechanism was not suitable (e.g. the `diskin` opcode), a dedicated solution was implemented. We also added support for asynchronous i-time operations to the engine, which will allow initialisation code to be performed in a separate thread to performance.

4.5 Thread-safety

In Csound 5, library users were expected to take care of thread-safety when splitting performance and control in separate threads (although some helper classes were available for this purpose in the Csound interfaces API). In Csound 6, thread-safety is built into the library, so API calls can be placed in separate threads (e.g. for control, table access and performance). The software bus channels, for instance use gcc atomic built-in functions (i.e. `__sync_lock_test_and_set()`)

4.6 Multicore operation

Following the introduction of multicore support in Csound 5 [11] [7], we have created an improved design with better use of resources. The new design uses more conservative re-drawing of directed acyclic graphs (DAG), which is now done only at the beginning and end of events rather than on every control cycle, and uses watch-lists, as found in SAT-solvers [12]. The effect of this change is significant; in almost all cases it gives major speed-up with two or more threads being used on recent processors, delivering about 60% of the time or better. Some preliminary figures are shown in table 1.

Finding a way of using multiple cores is a major challenge to software writers, and is particularly difficult in audio processing [13]. We think that this scheme will scale to a significant number of cores and open up the possibility for complex synthesis in real time.

It is important to note that parallelism in Csound is completely automatic and provided out-of-box by a single configuration option (requesting a given number of threads). No user modification of Csound code is required, and more importantly, no expertise in how to parallelise code is required. We understand this as a compiler problem, not a user one. Initial tests have indicated that the parallelism is generic enough to provide gains and scalability for arbitrary orchestras, only failing in pathological cases, such as when ksmps=1 and the computation small when the overhead is apparent or where there is no parallelism to find. It might even be possible to have an automatic mode where the code analyser can determine whether there is likely to be and advantage in using multiple threads.

5. EXTERNAL CHANGES

In addition to the developer-level changes listed above, significant external changes are also visible to end-users.

5.1 Generic Arrays

Csound 5 introduced a new type of variable that implemented simple one-dimensional arrays, and with it a suite of operations was also added. In Csound 6 arrays have been generalised, and all types can be constructed as one- or two-dimensional objects. This provides substantial flexibility for users of the language. For instance, we can have code constructs like this, where a bank of oscillators is spawned:

```
opcode OscBank,a,kki
 setksmps 1
 kamp,kfr,inum xin
 kph[] init inum
 kcnt = 0
 au = 0
 until kcnt == inum do
   au += sin(kph[kcnt])
   kph[kcnt] += kfr*kcnt*(2*$M_PI)/sr
   kcnt += 1
 od
     xout au*kamp
endop
```

Previously, such designs would have had to be implemented via recursive user-defined opcodes or instruments. But now, more straightforward loops can be used. The only care is that, as unit generators (opcodes) are effectively anonymous classes in the current syntax, those whose internal state advances on every call cannot be directly used in loops as in the example above. We are considering a number of possible syntactical solutions, including automatic parallel expansion, so that arrays can be used more freely with opcodes. Functions, and many unit generators that don't have an evolving internal state (e.g. a phase accumulator) can be used with no limitations. In addition to array data types, we have designed a full set of operations (such as list comprehensions, maps, copying, table access, etc.). We will implement these in subsequent updates.

5.2 New functional syntax

Another major external change to Csound is the possibility of a new functional syntax, where opcodes can be used in expressions of the general form

ans = opcode(arg-list)

This allows the inlining of opcodes in expressions, for instance, with the following code

```
out(moogladder
       (vco2
         (linen(p4,0.01,p3,0.1),p5),
        p5+linen(p5*4,0.01,p3,0.5),
        0.8))
```

being the equivalent of

```
k1 linen p4,0.01,p3,0.1
k2 linen p5*4,0.01,p3,0.5
a1 vco2  k1, p5
a2 moogladder  a1, k2+p5, 0.8
     out  a2
```

in the traditional Csound syntax.

Given the extensive use of polymorphism in Csound, the mechanism of type annotation can be used to resolve certain ambiguous expressions and to select the required opcode for a desired output type. The general form of annotations in functional syntax is

opcode:type(arg-list)

In version 6.00, only opcodes with a single output are allowed in this form, as multiple outputs will require the introdution of tuple types (current under plans). However, the functional syntax can be intermingled with the traditional out-op-in syntax in Csound code. Note that, as Csound is not a purely functional language, there are no guarantees that functions will not have side effects, so the change in syntax does not imply any internal operation modifications.

5.3 On-the-fly Compilation

With Csound 5, recompilation of code running in an instance of the engine required interruption of performance. This came to seem restrictive, specially for performances involving live coding, where either two instances would be used (so one could be alternatively recompiled while the other was active), or a complete set of instruments was required to be supplied.

In Csound 6, we have removed this restriction. Any new instruments can be added at any point, and will be available for new insertions. The mechanism allows for replacement of existing instruments, with any running instances of these being unaffected. User-defined opcodes can also be added at any point. From the use-case point of view, we expect that software using Csound will allow on-the-fly scripting of instruments, loading and instantiation.

From inside the orchestra language, however, it is also possible to add new instruments, via two special opcodes, `compileorc` and `compilestr`. The first opcode reads orchestra code from a file, parses and compiles it. The second performs the same operations on a string.

Hosts can also send instruments as strings via bus channels to be compiled, or save them in plain text files. Full access to parsing and compilation is provided via the API. The parse tree is also exposed via the API, so it is feasible that in the future alternative languages might be implemented, ready for Csound compilation. This is yet another step towards the full separation of engine and language, which started in Csound 5.

5.4 Sample-level accuracy

Traditionally, sample-level accuracy had been achieved in Csound by running it with a ksmps (block) size of 1. This has been always available as a global orchestra setting. Since Csound 5, user-defined opcodes can also have local ksmps values, enabling sample-level processing. In Csound 6, this is extended to instrument definitions, which can have a per-instance block size. However, global (whole-orchestra) sample-by-sample processing of this kind is relatively inefficient (even though in some other systems this is all that is available). For Csound 6, we have introduced a mechanism that allows sample-level accuracy that is completely independent of ksmps. This is enabled by an engine option ('sample-accurate'), but it is not on by default (for backward compatibility reasons, as it would possibly alter behaviour of older code). With this feature, we also have means of optimising multicore performance by processing in larger blocks, without loss of timing accuracy [1] [14].

5.5 Realtime priority mode

Another new feature of Csound 6 is a realtime priority mode that allows performance to be uninterrupted by blocking or time-consuming operations. This mode effectively forces opcodes that access disk to do so asynchronously, and also performs all init-time code in a separate thread. In this case, new instrument instances will invoke their init-pass code to happen in a worker thread, then immediately resume executing their performance-pass code. For example, the loading of large tables and similar operations will no longer directly affect performance. Similarly, opcodes reading or writing to disk will not cause dropouts (which was liable to happen in Csound 5, esp. on disk writing). This should enhance the performance of Csound code in interrupt-driven callbacks.

5.6 The new API

We have carefully revised the low-level Csound API. Functions exposing the new functionality have been added, and others have been removed in an effort to simplify API use. In particular, access to the software bus has been simplified. Also, as noted above, with on-the-fly compilation, new means of starting and running Csound instances has been added. Csound performances can be started with no orchestra or score, instruments and events can be added at any time to it. New ways of configuring the engine have also been provided, previously only possible via string flags and arguments. A simple Python example demonstrating some of the new API functions is shown below:

```
import csnd6
import time

cs = csnd6.csoundCreate(None)
csnd6.csoundSetOption(cs,``-odac'')
csnd6.csoundStart(cs)
perf = csnd6.CsoundPerformanceThread(cs)
perf.Play()

csnd6.csoundCompileOrc(cs, ```
event_i ``i'',1,0.1,1,1000,500
instr 1
k1 expon 1,p3,0.001
a2 oscili k1*p4,p5
event_i ``i'',1,0.1,1,p4,rnd(p5)+500
out a2
endin ''')

time.sleep(5)
perf.Stop()
```

This script runs for the synthesis engine only for 5 seconds, but in interactive contexts Csound would be open for performance indefinitely, accepting input in terms of orchestra code or realtime events. Examples such as these can be run in a read-eval-print loop (REPL) provided by emacs, vim, ipython or similar environments, for live-coding with Csound, as well as from other languages (Lua, Java, Clojure, etc.). Such possibilities are not limited to performance and composition, but also allow flexible use in research and teaching.

5.7 Miscellaneous improvements

Utilities have been updated to provide cross-platform support in terms of file formats, which is byte-order and precision independent. Support for string data in the score has also been made more flexible, so that an unlimited number of strings can be passed from events to instrument instances (previously this was limited to one). There is a proposal for a new parser for the score language, but details of this are still in the planning stage.

6. NEXT STEPS

At the time of writing, we are providing a Release Candidate version of Csound 6, which is available for all users to test. This will be followed by the first full release of Csound 6 for Linux, OSX , Windows, Android and iOS. Beyond that, we expect that the infrastructure changes will now allow significant room for further incremental development of new features and improvements, and publication of the internal abstract syntax tree format will allow new user-level languages to access the Csound engine and unit generators. The new releases will be developed in conjunction with third-party developments of frontends and applications, whose functionality, it is hoped, will be greatly enhanced by Csound 6.

[1] The effectiveness of parallelization of audio processes in general is tied to the granularity of processing, due to the overhead from spawning and joining the parallel processes. Larger granularity generally leads to greater event jitter and latency.

7. REFERENCES

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

[2] J. ffitch, "The Design of Csound5," in *LAC2005*. Karlsruhe, Germany: Zentrum für Kunst und Medientechnologie, April 2005, pp. 37–41.

[3] P. Manning, R. Berry, I. Bowler, N. Bailey, and A. Purvis, "Studio report, University of Durham, England," in *Proc. Int. Computer Music Conf. 1990*. Glasgow: ICMA, 1990.

[4] B. Vercoe, "Real-Time Csound, Software Synthesis with Sensing and Control," in *Proc. Int. Computer Music Conf. 1990*. Glasgow: ICMA, 1990, pp. 209–211.

[5] ——, "Extended Csound," in *On the Edge*, ICMA. ICMA and HKUST, 1996, pp. 141–142.

[6] V. Lazzarini, R. Walsh, and M. Brogan, "Two Cross Platform Csound-based Plugin Generators," in *Proc. Int. Computer Music Conf. 2008*. Belfast: ICMA, 2008.

[7] J. ffitch, "Parallel Execution of Csound," in *Proceedings of ICMC 2009*. Montreal: ICMA, 2009.

[8] R. Walsh, "Cabbage Audio Plugin Framework," in *Proc. Int. Computer Music Conf. 2011*. Huddersfield: ICMA, 2011.

[9] V. Lazzarini, S. Yi, J. Timoney, D. Keller, and M. Pimenta, "The Mobile Csound Platform," in *Proc. Int. Computer Music Conf. 2012*. Ljubljana: ICMA, 2012.

[10] V. Lazzarini, "The Development of Computer Music Programming Systems," *Journal of New Music Research*, vol. 42, no. 1, 2013.

[11] C. Wilson, "Csound Parallelism," Department of Computer Science, University of Bath, Tech. Rep. CSBU-2009-07, 2009.

[12] C. A. Brown and P. W. Purdom Jr, "An Empirical Comparison of Backtracking Algorithms," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 4, no. 3, 1982.

[13] D. Wessel, R. Dannenberg, Y. O. a d M. Puckette, P. V. Roy, and G. Wang, "Reinventing Audio and Music Computation for Many-Core Processors," in *Proc. Int. Computer Music Conf. 2008*. Belfast: ICMA, 2008.

[14] Y. Orlarey, S. Letz, and D. Fober, "Automatic Parallelization of FAUST code," in *LAC2009*. Parma: Italy: Casa della Musica, April 2009.

Appendix E

Csound on the Web

Original Publication:

Victor Lazzarini, Edward Costello, Steven Yi, and John ffitch. Csound on the Web. In *Linux Audio Conference 2014*, 2014.

Csound on the Web

Victor LAZZARINI and **Edward COSTELLO** and **Steven YI** and **John FITCH**
Department of Music
National University of Ireland
Maynooth,
Ireland,
{victor.lazzarini@nuim.ie, edwardcostello@gmail.com, stevenyi@gmail.com, jpff@codemist.co.uk }

Abstract

This paper reports on two approaches to provide a general-purpose audio programming support for web applications based on Csound. It reviews the current state of web audio development, and discusses some previous attempts at this. We then introduce a Javascript version of Csound that has been created using the Emscripten compiler, and discuss its features and limitations. In complement to this, we look at a Native Client implementation of Csound, which is a fully-functional version of Csound running in Chrome and Chromium browsers.

Keywords

Music Programming Languages; Web Applications;

1 Introduction

The web browser has become an increasingly viable platform for the creation and distribution of various types of media computing applications[Wyse and Subramanian, 2013]. It is no surprise that audio is an important part of these developments. For a good while now we have been interested in the possibilities of deployment of client-side Csound-based applications, in addition to the already existing server-side capabilities of the system. Such scenarios would be ideal for various uses of Csound. For instance, in Education, we could see the easy deployment of Computer Music training software for all levels, from secondary schools to third-level institutions. For the researcher, web applications can provide an easy means of creating prototypes and demonstrations. Composers and media artists can also benefit from the wide reach of the internet to create portable works of art. In summary, given the right conditions, Csound can provide a solid and robust general-purpose audio development environment for a variety of uses. In this paper, we report on the progress towards supporting these conditions.

2 Audio Technologies for the Web

The current state of audio systems for world-wide web applications is primarily based upon three technologies: Java[1], Adobe Flash[2], and HTML5 WebAudio[3]. Of the three, Java is the oldest. Applications using Java are deployed via the web either as Applets[4] or via Java Web Start[5]. Java as a platform for web applications has lost popularity since its introduction, primarily due to historically sluggish start-up times as well as concerns over security breaches. Also of concern is that major browser vendors have either completely disabled Applet loading or disabled them by default, and that NPAPI plugin support–which the Java plugin for browsers is implemented with–is planned to be dropped in future browser versions[6]. While Java sees strong support on the server-side and desktop, its future as a web-deployed application is tenuous at best and difficult to recommend for future audio system development.

Adobe Flash as a platform has seen large-scale support across platforms and across browsers. Numerous large-scale applications have been developed such as AudioTool[7], Patchwork[8], and Noteflight[9]. Flash developers can choose to deploy to the web using the Flash plugin, as well as use Adobe Air[10] to deploy to desktop and mobile devices. While these applications demonstrate what can be developed for the web

[1] http://java.oracle.com
[2] http://www.adobe.com/products/flashruntimes.html
[3] http://www.w3.org/TR/webaudio/
[4] http://docs.oracle.com/javase/tutorial/deployment/applet/index.html
[5] http://docs.oracle.com/javase/tutorial/deployment/webstart/index.html
[6] http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html
[7] http://www.audiotool.com/
[8] http://www.patchwork-synth.com
[9] http://www.noteflight.com
[10] http://www.adobe.com/products/air.html

using Flash, the Flash platform itself has a number of drawbacks. The primary tools for Flash development are closed-source, commercial applications that are unavailable on Linux, though open source Flash compilers and IDEs do exist[11]. There has been a backlash against Flash in browsers, most famously by Steve Jobs and Apple[12], and the technology stack as a whole has seen limited development with the growing popularity of HTML5. At this time, Flash may be a viable platform for building audio applications, but the uncertain future makes it difficult to recommend.

Finally, HTML5 Web Audio is the most recent of technologies for web audio applications. Examples include the "Recreating the sounds of the BBC Radiophonic Workshop using the Web Audio APIßite[13], Gibberish[14], and WebPd[15]. Unlike Java or Flash, which are implemented as browser plug-ins, the WebAudio API is a W3C proposed standard that is implemented by the browser itself.[16] Having built-in support for Audio removes the security issues and concerns over the future of plug-ins that affect Java and Flash. However, the Web Audio API has limitations that will be explored further below in the section on Emscripten.

3 Csound-based Web Application Design

Csound is a music synthesis system that has roots in the very earliest history of computer music. Csound use in Desktop and Mobile applications has been discussed previously in [Lazzarini et al., 2012b], [Yi and Lazzarini, 2012], and [Lazzarini et al., 2012a].

Prior to the technologies presented this paper, Csound-based web applications have employed Csound only on the server-side. For example, NetCsound [17] allows sending a CSD file to the server, where it would render the project to disk and email the user a link to the rendered file when complete. Another use of Csound on the server is Oeyvind Brandtsegg's VLBI Music [18], where Csound is running on the server and publishes its audio output to an audio stream that end users can listen to. A similar architecture is found in [Johannes and Toshihiro, 2013]. Since version 6.02, Csound also includes a built-in server, that can be activated through an option on start up. The server is able to receive code directly through UDP connections and compile them on the fly.

Using Csound server-side has both positives and negatives that should be evaluated for a project's requirements. It can be appropriate to use if the project's design calls for a single audio stream/Csound instance that is shared by all listeners. In this case, users might interact with the audio system over the web, at the expense of network latency. Using multiple realtime Csound instances—as would be the case if there was one per user—would certainly be taxing for a single server and would require careful resource limiting. For multiple non-realtime Csound instances, as in the case of NetCsound, multiple jobs may be scheduled and batch processed with less problems than with realtime systems, though resource management is still a concern.

A possibly more flexible way to deploy Csound over the internet is to support client-side applications that use the browser as a platform. Two attempts at this have been explored in the past. The first was the now-defunct ActiveX Csound (also known as AXCsound)[19], which allowed embedding Csound into a webpage as an ActiveX Object. This technology is no longer maintained and was only available for use on Windows with Internet Explorer. A second attempt was made in the Mobile Csound Project[Lazzarini et al., 2012b], where a proof-of-concept Csound-based application was developed with Java and deployed using Java Web Start, achieving client-side Csound use via the browser. However, the technology required special permissions to run on the client side and required Java to be installed. Due to those issues and the unsure future of Java over the web, the solution was not further explored.

The two systems described in this paper are browser-based solutions that run on the client-side. The both share the following benefits:

[11] `http://www.flashdevelop.org/`

[12] `http://www.apple.com/hotnews/thoughts-on-flash/`

[13] `http://webaudio.prototyping.bbc.co.uk/`

[14] Available at `https://github.com/charlieroberts/Gibberish`, discussed in [Roberts et al., 2013]

[15] `https://github.com/sebpiq/WebPd`

[16] `http://caniuse.com/audio-api` lists current browsers that support the Web Audio API

[17] Available at `http://dream.cs.bath.ac.uk/netcsound/`, discussed in [ffitch et al., 2007]

[18] `http://www.researchcatalogue.net/view/55360/55361`

[19] We were unable to find a copy of this online, but one is available from the CD-ROM included with [Boulanger, 2000]

- Csound has a large array of signal processing opcodes made immediately available to web-based projects.
- They are compiled using the same source code as is used for the desktop and mobile version of Csound. They only require recompiling to keep them in sync with the latest Csound features and bug fixes.
- Csound code that can be run with these browser solutions can be used on other platforms. Audio systems developed using Csound code is then cross-platform across the web, desktop, mobile, and embedded systems (i.e. Raspberry Pi, Beaglebone; discussed in [Batchelor and Wignall, 2013]). Developers can reuse their audio code from their web-based projects elsewhere, and vice versa.

4 Emscripten

Emscripten is a a project created by Alon Zakai at the Mozilla Foundation that compiles the assembly language used by the LLVM compiler into Javascript [Zakai, 2011]. When used in combination with LLVM's Clang frontend, Emscripten allows applications written in C/C++ or languages that use C/C++ runtimes to be run directly in web browsers. This eliminates the need for browser plugins and takes full advantage of web standards that are already in common use.

In order to generate Javascript from C/C++ sourcecode the codebase is first compiled into LLVM assembly language using LLVM's Clang frontend. Emscripten translates the resulting LLVM assembly language into Javascript, specifically an optimised subset of Javascript entitled asm.js. The asm.js subset of Javascript is intended as a low-level target language for compilers and allows a number of optimisations which are not possible with standard Javascript[20]. Code semantics which differ between Javascript and LLVM assembly are emulated when accurate code is required. Emscripten has built-in methods to check for arithmetic overflow, signing issues and rounding errors. If emulation is not required, code is translated without semantic emulation in order to achieve the best execution performance [Zakai, 2011].

Implementations of the C and C++ runtime libraries have been created for applications compiled with Emscripten. These allow programs written in C/C++ to transparently perform common tasks such as using the file system, allocating memory and printing to the console. Emscripten allows a virtual filesystem to be created using its FS library, which is used by Emscripten's libc and libcxx for file I/O[21]. Files can be added or removed from the virtual filesystem using a number of Javascript helper functions. It is also possible to directly call C functions from Javascript using Emscripten[22]. These functions must first be named at compile time so they are not optimised out of the resulting compiled Javascript code. The required functions are then wrapped using Emscripten's *cwrap* function, and assigned to a Javascript function name. The *cwrap* function allows many Javascript variables to be used transparently as arguments to C functions, such as passing Javascript strings to functions which require the C languages *const char* array type.

Although Emscripten can successfully compile a large section of C/C++ code there are still a number of limitations to this approach due to limitations within the Javascript language and runtime. As Javascript doesn't support threading, Emscripten is unable to compile codebases that make use of threads. Some concurrency is possible using web workers, but they do not share state. It is also not possible to directly implement 64-bit integers in Javascript as all numbers are represented using 64-bit doubles. This results in a risk of rounding errors being introduced to the compiled Javascript when performing arithmetic operations with 64-bit integers [Zakai, 2011].

4.1 CsoundEmscripten

CsoundEmscripten is an implementation of the Csound language in Javascript using the Emscripten compiler. A working example of CsoundEmscripten can be found at `http://eddyc.github.io/CsoundEmscripten/`. The compiled Csound library and *CsoundObj* Javascript class can be found at `https://github.com/eddyc/CsoundEmscripten/`. CsoundEmscripten consists of three main modules:

- The Csound library compiled to Javascript using Emscripten.
- A structure and associated functions written in C named *CsoundObj* implemented

[20] `http://asmjs.org/spec/latest/`

[21] `https://github.com/kripken/emscripten/wiki/Filesystem-API`

[22] `https://github.com/kripken/emscripten/wiki/Interacting-with-code`

on top of the Csound library that is compiled to Javascript using Emscripten.

- A handwritten Javascript class also named *CsoundObj* that contains the public interface to CsoundEmscripten. The Javascript class both wraps the compiled *CsoundObj* structure and associated functions, and connects the Csound library's audio output to the Web Audio API.

4.1.1 Wrapping the Csound C API for use with Javascript

In order to simplify the interface between the Csound C API and the Javascript class containing the CsoundEmscripten public interface, a structure named *CsoundObj* and a number of functions which use this structure were created. The structure contains a reference to the current instance of Csound, a reference to Csound's input and output buffer, and Csound's 0dBFS value. Some of the functions that use this structure are:

- `CsoundObj_new()` - This function allocates and returns an instance of the *CsoundObj* structure. It also initialises an instance of Csound and disables Csound's default handling of sound I/O, allowing Csound's input and output buffers to be used directly.
- `CsoundObj_compileCSD(self, filePath, samplerate, controlrate, buffersize)` - This function is used to compile CSD files, it takes as its arguments: a pointer to the *CsoundObj* structure *self*, the address of a CSD file given by *filePath*, a specified sample rate given by *samplerate*, a specified control rate given by *controlrate* and a buffer size given by *buffersize*. The CSD file at the given address is compiled using these arguments.
- `CsoundObj_process(self, inNumberFrames, inputBuffer, outputBuffer)` - This function copies audio samples to Csound's input buffer and copies samples from Csound's output buffer. It takes as its arguments: a pointer to the *CsoundObj* structure *self*, an integer *inNumberFrames* specifying the number of samples to be copied, a pointer to a buffer containing the input samples named *inputBuffer* and a pointer to a destination buffer to copy the output samples named *outputBuffer*.

Each of the other functions that use the *CsoundObj* structure simply wrap existing functions present in the Csound C API. The relevant functions are:

- `csoundGetKsmps(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio frames per control sample.
- `csoundGetNchnls(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio output channels.
- `csoundGetNchnlsInput(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio input channels.
- `csoundStop(csound)` - This function takes as its argument a pointer to an instance of Csound stops the current performance pass.
- `csoundReset(csound)` - This function takes as its argument a pointer to an instance of Csound and resets its internal memory and state in preparation for a new performance.
- `csoundSetControlChannel(csound, name, val)` - This function takes as its arguments: a pointer to an instance of Csound, a string given by *name*, and number given by *val*, it sets the numerical value of a Csound control channel specified by the string *name*.

The *CsoundObj* structure and associated functions are compiled to Javascript using Emscripten and added to the compiled Csound Javascript library. Although this is not necessary, keeping the compiled *CsoundObj* structure and functions in the same file as the Csound library makes it more convenient when including CsoundEmscripten within web pages.

4.1.2 The CsoundEmscripten Javascript interface

The last component of CsoundEmscripten is the *CsoundObj* Javascript class. This class provides the public interface for interacting with the compiled Csound library. As well as allocating

an instance of Csound this class provides methods for controlling performance and setting the values of Csound's control channels. Additionally, this class interfaces with the Web Audio API, providing Csound with samples from the audio input bus and copying samples from Csound to the audio output bus. Audio I/O and the Csound process are performed in Javascript using the Web Audio API's *ScriptProcessorNode*. This node allows direct access to input and output samples in Javascript allowing audio processing and synthesis using the Csound library.

Csound can be used in any webpage by creating an instance of *CsoundObj* and calling the available public methods in Javascript. The methods available in the *CsoundObj* class are:

- `compileCSD(fileName)` This method takes as its argument the address of a CSD file *fileName* and compiles it for performance. The CSD file must be present in Emscripten's virtual filesystem. This method calls the compiled C function *CsoundObj_compileCSD*. It also creates a *ScriptProcessorNode* instance for Audio I/O.
- `enableAudioInput()` This method enables audio input to the web browser. When called, it triggers a permissions dialogue in the host web browser requesting permission to allow audio input. If permission is granted, audio input is available for the running Csound instance.
- `startAudioCallback()` This method connects the *ScriptProcessorNode* to the audio output and, if required, the audio input. The *ScriptProcessorNodes* audio processing callback is also started. During each callback, if required, audio samples from the *ScriptProcessorNodes* input are copied into Csound's input buffer and any new values for Csound's software channels are set. Csound's *csoundPerformKsmps()* function is called and any output samples are copied into the *ScriptProcessorNodes* output buffer.
- `stopAudioCallback()` This method disconnects the current running *ScriptProcessorNode* and stops the audio process callback. If required this method also disconnects any audio inputs.
- `addControlChannel(name, initialValue)` This method adds an object to a Javascript array that is used to update Csound's named channel values. Each object contains a string value given by *name*, a float value given by *initialValue* and additionally a boolean value indicating whether the float value has been updated.
- `setControlChannelValue(name, value)` This method sets the value of a named control channel given by the string *name* to the specified input value.
- `getControlChannelValue(name)` This method returns the current value of a named control channel given by the string *name*.

4.1.3 Limitations

Using CsoundEmscripten, it is possible to add Csound's audio processing and synthesis capabilities to any web browser that supports the Web Audio API. Unfortunately this approach of bringing Csound to the web comes with a number of drawbacks.

Although Javascript engines are constantly improving in speed and efficiency, running Csound entirely in Javascript is a processor intensive task on modern systems. This is especially troublesome when trying to run even moderately complex CSD files on mobile computing devices.

Another limitation is due to the design of the *ScriptProcessorNode* part of the Web Audio API. Unfortunately, the *ScriptProcessorNode* runs on the main thread. This can result in audio glitching when another process on the main thread—such as the UI—causes a delay in audio processing. As part of the W3Cs Web Audio Spec review it has been suggested that the *ScriptProcessorNode* be moved off of the main thread[23]. There has also been a resolution by the Web Audio API developers that they will make it possible to use the *ScriptProcessorNode* with web workers[24]. Hopefully in a future version of the Web Audio API the *ScriptProcessorNode* will be more capable of running the kind complex audio processing and synthesis capabilities allowed by the Csound library.

This version of Csound also doesn't support plugins, making some opcodes unavailable. Additionally, MIDI I/O is not currently suppor-

[23] https://github.com/w3ctag/spec-reviews/blob/master/2013/07/WebAudio.md#issue-scriptprocessornode-is-unfit-for-purpose-section-15

[24] https://www.w3.org/Bugs/Public/show_bug.cgi?id=17415#c94

ted. This is not due to the technical limitations of Emscripten, rather it was not implemented due to the current lack of support for the Web MIDI standard in Mozillas Firefox[25] and in the Webkit library[26].

5 Beyond WebAudio: Audio Applications with PNaCl

As an alternative to the development of audio applications for web deployment in pure Javascript, it is possible to take advantage of the Native Clients (NaCl) platform[27]. This allows the use of C and C++ code to create components that are accessible to client-side Javascript, and run natively inside the browser. NaCl is described as a sandboxing technology, as it provides a safe environment for code to be executed, in an OS-independent manner [Yee et al., 2009] [Sehr et al., 2010]. This is not completely unlike the use of Java with the Java Webstart Technology (JAWS), which has been discussed elsewhere in relation to Csound [Lazzarini et al., 2012b].

There are two basic toolchains in NaCl: native/gcc and PNaCl [Donovan et al., 2010]. While the former produces architecture-dependent code (arm, x86, etc.), the latter is completely independent of any existing architecture. NaCl is currently only supported by the Chrome and Chromium browsers. Since version 31, Chrome enables PNaCl by default, allowing applications created with that technology to work completely out-of-the-box. While PNaCl modules can be served from anywhere in the open web, native-toolchain NaCl applications and extensions can only be installed from Google's Chrome Web Store.

5.1 The Pepper Plugin API

An integral part of NaCl is the Pepper Plugin API (PPAPI, or just Pepper). It offers various services, of which interfacing with Javascript and accessing the audio device is particularly relevant to our ends. All of the toolchains also include support for parts of the standard C library (eg. stdio), and very importantly for Csound, the pthread library. However, absent from the PNaCl toolchain are dlopen() and friends, which means no dynamic loading is available there.

Javascript client-side code is responsible for requesting the loading of a NaCl module. Once the module is loaded, execution is controlled through Javascript event listeners and message passing. A postMessage() method is used by Pepper to allow communication from Javascript to PNaCl module, triggering a message handler in the C/C++ side. In the opposite direction, a *message* event is issued when C/C++ code calls the equivalent PostMessage() function.

Audio output is well supported in Pepper with a mid-latency callback mechanism (ca. 10-11ms, 512 frames at 44.1 or 48 KHz sampling rate). Its performance appears to be very uniform across the various platforms. The Audio API design is very straightforward, although the library is a little rigid in terms of parameters. It supports only stereo at one of the two sampling rates mentioned above). Audio input is not yet available in the production release, but support can already be seen in the development repository.

The most complex part of NaCl is access to the local files. In short, there is no open access to the client disk, only to sandboxed filesystems. It is possible to mount a server filesystem (through httpfs), a memory filesystem (memfs), as well as local temporary or permanent filesystems (html5fs). For those to be useful, they can only be mounted and accessed through the NaCl module, which means that any copying of data from the user disk into these partitions has to be mediated by code written in the NaCl module. For instance, it is possible to take advantage of the file HTML5 tag and to get data from NaCl into a Javascript blob so that it can be saved into the user's disk. It is also possible to copy a file from disk into the sandbox using the URLReader service supplied by Pepper.

5.2 PNaCl

The PNaCl toolchain compiles code down to a portable bitcode executable (called a *pexe*). When this is delivered to the browser, an ahead-of-time compiler is used to translate the code into native form. A web application using PNaCl will contain three basic components: the pexe binary, a manifest file describing it, and a client-side script in JS, which loads and allows interaction with the module via the Pepper messaging system.

5.3 Csound for PNaCl

A fully functional implementation of Csound for Portable Native Clients is available from `http:`

[25] `https://bugzilla.mozilla.org/show_bug.cgi?id=836897`

[26] `https://bugs.webkit.org/show_bug.cgi?id=107250`

[27] `https://developers.google.com/native-client`

//vlazzarini.github.io. The package is composed of three elements: the Javascript module (csound.js), the manifest file (csound.nmf), and the pexe binary (csound.pexe). The source for the PNaCl component is also available from that site (csound.cpp). It depends on the Csound and Libsndfile libraries compiled for PNaCl and the NaCL sdk. A Makefile for PNaCl exists in the Csound 6 sources.

5.3.1 The Javascript interface

Users of Csound for PNaCl will only interact with the services offered by the Javascript module. Typically an application written in HTML5 will require the following elements to use it:

- the csound.js script
- a reference to the module using a div tag with id="engine"
- a script containing the code to control Csound.

The script will contain calls to methods in csound.js, such as:

- `csound.Play()` - starts performance
- `csound.PlayCsd(s)` - starts performance from a CSD file *s*, which can be in ./http/ (ORIGIN server) or ./local/ (local sandbox).
- `csound.RenderCsd(s)` - renders a CSD file *s*, which can be in ./http/ (ORIGIN server) or ./local/ (local sandbox), with no RT audio output. The "finished render"message is issued on completion.
- `csound.Pause()` - pauses performance
- `csound.CompileOrc(s)` - compiles the Csound code in the string *s*
- `csound.ReadScore(s)` - reads the score in the string *s* (with preprocessing support)
- `csound.Event(s)` - sends in the line events contained in the string *s* (no preprocessing)
- `csound.SetChannel(name, value)` - sends the control channel *name* the value *value*, both arguments being strings.

As it starts, the PNaCl module will call a `moduleDidLoad()` function, if it exists. This can be defined in the application script. Also the following callbacks are also definable:

- `function handleMessage(message)`: called when there are messages from Csound (pnacl module). The string *message.data* contains the message.
- `function attachListeners()`: this is called when listeners for different events are to be attached.

In addition to Csound-specific controls, the module also includes a number of filesystem facilities, to allow the manipulation of resources in the server and in the sandbox:

- `csound.CopyToLocal(src, dest)` - copies the file *src* in the ORIGIN directory to the local file *dest*, which can be accessed at ./local/*dest*. The "Complete"message is issued on completion.
- `csound.CopyUrlToLocal(url,dest)` - copies the url *url* to the local file *dest*, which can be accessed at ./local/*dest*. Currently only ORIGIN and CORS urls are allowed remotely, but local files can also be passed if encoded as urls with the webkitURL.createObjectURL() javascript method. The "Complete"message is issued on completion.
- `csound.RequestFileFromLocal(src)` - requests the data from the local file *src*. The "Complete"message is issued on completion.
- `csound.GetFileData()` - returns the most recently requested file data as an ArrayObject.

A series of examples demonstrating this API is provided in github. In particular, an introductory example is found on `http://vlazzarini.github.io/minimal.html`.

5.3.2 Limitations

The following limitations apply to the current release of Csound for PNaCl:

- no realtime audio input (not supported yet in Pepper/NaCl)
- no MIDI in the NaCl module. However, it might be possible to implement MIDI in JavaScript, and using the csound.js functions, send data to Csound, and respond to MIDI NOTE messages.
- no plugins, as pNaCl does not support dlopen() and friends. This means some

Csound opcodes are not available as they reside in plugin libraries. It might be possible to add some of these opcodes statically to the Csound pNaCl library in the future.

6 Conclusions

In this paper we reviewed the current state of support for the development of web-based audio and music applications. As part of this, we explored two approaches in deploying Csound as an engine for general-purpose media software. The first consisted of a Javascript version created with the help of the Emscripten compiler, and the second a native C/C++ port for the Native Client platform, using the Portable Native Client toolchain. The first has the advantage of enjoying widespread support by a variety of browsers, but is not yet fully deployable. On the other hand, the second approach, while at the moment only running on Chrome and Chromium browsers, is a robust and ready-for-production version of Csound.

7 Acknowledgements

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.

References

Paul Batchelor and Trev Wignall. 2013. BeaglePi: An Introductory Guide to Csound on the BeagleBone and the Raspberry Pi, as well other Linux-powered tinyware. *Csound Journal*, (18).

Richard J. Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February.

Alan Donovan, Robert Muth, Brad Chen, and David Sehr. 2010. PNaCl: Portable Native Client Executables. *Google White Paper*.

John ffitch, James Mitchell, and Julian Padget. 2007. Composition with sound web services and workflows. In Suvisoft Oy Ltd, editor, *Proceedings of the 2007 International Computer Music Conference*, volume I, pages 419–422. ICMA and Re:New, August. ISBN 0-9713192-5-1.

Tarmo Johannes and Kita Toshihiro. 2013. „Và, pensiero!“ - Fly, thought! Experiment for interactive internet based piece using Csound6 . `http://tarmo.uuu.ee/varia/failid/cs/pensiero-files/pensiero-presentation.pdf`. Accessed: February 2nd, 2014.

Victor Lazzarini, Steven Yi, and Joseph Timoney. 2012a. Digital audio effects on mobile platforms. In *Proceedings of DAFx 2012*.

Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. 2012b. The Mobile Csound Platform. In *Proceedings of ICMC 2012*.

Charles Roberts, Graham Wakefield, and Matthew Wright. 2013. The Web Browser As Synthesizer And Interface. *Proceedings of the International Conference on New Interfaces for Musical Expression*.

David Sehr, Robert Muth, Cliff Bife, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*.

Lonce Wyse and Srikumar Subramanian. 2013. The Viability of the Web Browser as a Computer Music Platform. *Computer Music Journal*, 37(4):10–23.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 IEEE Symposium on Security and Privacy*.

Steven Yi and Victor Lazzarini. 2012. Csound for Android. In *Linux Audio Conference*, volume 6.

Alon Zakai. 2011. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, pages 301–312. ACM.

Appendix F

Extending Aura with Csound Opcodes

Original Publication:

Steven Yi, Roger Dannenberg, Victor Lazzarini, and John ffitch. Extending Aura with Csound Opcodes. In *Proceedings of ICMC 2014*, Athens, Greece, 2014. ICMA.

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.

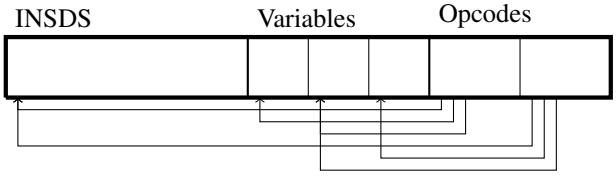

Figure 4. Memory block diagram for a Csound instrument instance.

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 resposibilty 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.

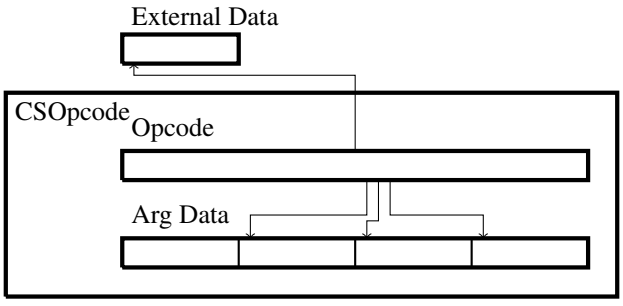

Figure 9. Memory diagram for `CSOpcode` and argument handling.

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 Instrs 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.

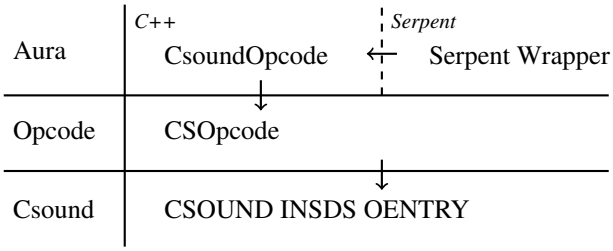

Figure 11. Architecture showing relationship between Csound, Opcode, and Aura layers.

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 Instrs, 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 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 CSOpcodes 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 Instrs, 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 OENTRYs 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 Instrs 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 Instrs 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.

Appendix G

Extending Csound to the Web

Original Publication:

Victor Lazzarini, Edward Costello, Steven Yi, and John ffitch. Extending Csound to the Web. In Proceedings of the Web Audio Conference 2015, 2015.

Extending Csound to the Web

Victor Lazzarini, Edward Costello, Steven Yi, John ffitch
Department of Music
Maynooth University
Ireland
victor.lazzarini@nuim.ie, edward.costello@nuim.ie, stevenyi@gmail.com,
jpff@codemist.co.uk

ABSTRACT

This paper discusses the presence of the sound and music computing system Csound in the modern world-wide web browser platform. It introduces the two versions of the system currently available, as pure Javascript code, and as portable Native Client binary module with a Javascript interface. Three example applications are presented, showing some of the potential uses of the system. The paper concludes with a discussion of the wider Csound application ecosystem, and the prospects for its future development.

Keywords

Music Programming Languages; Web Applications;

1. INTRODUCTION

In a recent paper[6], we have introduced two ports of the Csound sound and music computing system to the web-browser platform. In the first one, the Csound codebase was adapted and compiled to a subset of the Javascript language, asm.js, via the Emscripten compiler [11]. The second port employed the Portable Native Client (PNaCl)[3] technology to provide a platform for the implementation a Csound API-based Javascript frontend.

While the former is available for a wider range of internet browsers, as it is based on pure Javascript, the second project takes advantage of the near-native performance of PNaCl to provide a very efficient implementation of the system. Other significant differences between the two offerings are notable: the existence of pthread support in PNaCl versus the single-thread nature of pure Javascript; the dependence on Web Audio ScriptProcessorNode and audio IO in the Emscripten-based Csound versus the Pepper API-based audio and threading offered by PNaCl; and finally, the fact that the pure-Javascript implementation functions as a wrapper to the Csound API, whereas the PNaCl version provides a higher-level Javascript frontend to the system, with no direct access to the API.

Csound on the web browser is, therefore, an attractive option for audio programming targeting applications that run on clients (as opposed to server-side solutions). It offers an alternative to Adobe Flash (used, for instance in AudioTool[1], Patchwork[2], and Noteflight[3]), as well as standard HTML (used by the BBC Radiophonic workshop recreations[4], Gibberish[5], and WebPd[6]). It also fits with the development of an ecosystem of applications based on Csound, which allows users to easily move from one platform to another: desktop, mobile[8][10][7], small and custom computers[1], servers[4][5] (see also `http://www.researchcatalogue.net/view/55360/5536` for another application example), and now web clients. This paper is organised as follows: we will start with a brief overview of the two implementations of Csound for web browsers; this is followed by a discussion of some key example applications; we then explore the concept of the Csound application ecosystem and its significance; the final section shows the directions we intend to take the current ideas, and how they fit in the overall development of the system.

2. BROWSER-BASED CSOUND: OVERVIEW

The two implementations of Csound for web browsers use distinct technologies. The first is a Javascript-only port, created with the Emscripten compiler; the second is a C/C++-based application, which uses the PNaCl toolchain and the its ahead-of-time compiler module, which currently exists only on Chrome and Chromium browsers.

2.1 Javascript Csound

Csound can now be run natively within any major web browser as a Javascript library using Emscripten. Emscripten can translate from LLVM bitcode into Javascript enabling programs written in a language supported by the LLVM compiler, such as C, to be compiled into Javascript and executed on a web page. Emscripten translates the LLVM bitcode into a strict subset of Javascript called *asm.js*. By disallowing some features of the language, Javascript engines can perform optimisations not possible using standard Javascript,

[1] `http://www.audiotool.com/`
[2] `http://www.patchwork-synth.com`
[3] `http://www.noteflight.com`
[4] `http://webaudio.prototyping.bbc.co.uk/`
[5] Available at `https://github.com/charlieroberts/Gibberish`, discussed in [9]
[6] `https://github.com/sebpiq/WebPd`

which can result in significant performance gains.

As it is written entirely in C and has only one required external dependency, Csound makes an ideal codebase for adding Javascript as a build target using Emscripten. The only external library required to build Csound is libsndfile. This library is used by some of Csound's built-in opcodes and the core system for saving and opening various sound file formats. In order to build and run Csound successfully it is first necessary to compile libsndfile into a Javascript library. Emscripten comes with a number of python scripts which set the necessary environmental variables for the build configuration and compilation of software projects into Javascript. These scripts can be used to invoke the libsndfile *configure* script and *make* file which compile the libsndfile source code into an asm.js library. The resulting Javascript library can be linked to Csound during the build process.

Csound uses the CMake build system to manage the compilation of binaries for supported platforms. Fortunately, Emscripten provides support for using CMake and comes with a toolchain file which sets the required toolchain variables for project compilation using Emscriptens compiler.

In order for Csound to compile successfully, there are also some minor changes which have to be made to the source code. Csound has the option of using threads for a number of operations during runtime, but as Emscripten does not support trans-compiling code bases which make use of threads, this functionality is removed during the build configuration step. Additionally, many of the features available in the Desktop build of Csound are also disabled in the Javascript library which do not currently make sense within a web page context such as JACK support. The plugin opcodes such as Fluidsynth and STK are also unavailable at this time but may be included in future releases.

Communicating with the Csound process is done through the provided C API. This allows an external application to control the Csound process in a number of ways, including compiling instruments, sending control signals and accessing Csound's audio input and output sample buffers. Emscripten provides wrapper functions which allow Javascript variables to be used as arguments to Emscripten compiled C functions, for instance, when using a Javascript string type as input to a C function taking a character array as an argument. This makes it possible to use Csound's C API functions directly within Javascript, however, an interface to a number of API functions has been created which greatly simplifies using API calls in a web page context. The interface consists of a Javascript class *CsoundObj*, which contains the necessary methods for instantiating and controlling Csound.

The following html creates a new instance of Csound, sends an orchestra string for compilation and plays the compiled instrument for one second.

```
<!DOCTYPE html>
<head>
<title></title>
<script src="javascripts/libcsound.js"></script>
<script src="javascripts/CsoundObj.js"></script>
</head>
<body>
<script>
var csound = new CsoundObj();
csound.compileOrc("ksmps=256\n" +
                  "nchnls=2\n" +
                  "0dbfs=1\n" +
                  "instr 1\n" +
                  "a1 vco2 0.2, 440\n" +
                  "outs a1, a1\n" +
                  "endin\n");
csound.startAudioCallback();
var scoreString = "i1 0 1"
csound.readScore(scoreString);
</script>
</body>
</html>
```

The *CsoundObj* class also contains methods for sending control messages using html and audio input to the running Csound instance via the Web Audio API. As Emscripten also provides a virtual file system that compiled C code can access, it is possible for Csound to write and play back audio files. A number of examples demonstrating the functionality provided by the Csound Javascript API can be found at `http://eddyc.github.io/CsoundEmscripten/`.

2.2 PNaCl Csound

Native Client is a recent technology developed by the Chromium project, which provides a sandboxing environment for applications running on browsers. It exists in two basic forms: one that works with natively-compiled modules (hardware-dependent, for i386, x86_64, arm, mips, etc); and another that is hardware independent, PNaCl. The former is currently only enabled for Chrome-store supplied applications, while the latter can be offered on the open web. The Csound port for Native Client has targeted the PNaCl platform, as it provides a flexible environment for the development of audio-based web applications.

The PNaCl project provides a toolchain so that C/C++ applications to be easily ported to it. Code is compiled to a bytecode representation (called a pexe module). This is then further compiled ahead-of-time to the target hardware as the page containing it is loaded. Web pages containing a PNaCl module need to be served over http, so for testing and debugging, a minimal http server is required.

As part of the PNaCl platform, we have the Pepper API, which fulfills three main roles here: general-purpose communication between the browser and the PNaCl code; access to the sandbox for file IO; and audio IO. In addition to Pepper, a number of basic C libraries are present in PNaCl, such as pthreads, and the C stdio library. Ports of common Unix libraries are also available (libogg, libvorbis, libpng, libopenal, libjpeg, to cite but a few).

PNaCl Csound is composed of two main elements:

1. the pexe module (csound.pexe): based on the Csound library, provides means to run and control Csound, as well as access to files in the sandbox
2. a Javascript interface (csound.js): the PNaCl Csound

functionality is exposed via a simple Javascript module, which allows applications to interface with Csound programmatically, in similar a way to the other language frontends like csound6 for PD, and csound for MaxMSP.

Each pexe module (one per page) runs one single Csound engine instance. For multiple instances, we would require separate web pages for each. A simple PNaCl Csound application to play a sine beep for 5 seconds looks like this:

```
<!DOCTYPE html>
<html>
<head>
 <title>Beep!</title>
 <script type="text/javascript" src="csound.js">
 </script>
 <script type="text/javascript">
 // this function is called by csound.js
 // after the PNaCl module is loaded
 function moduleDidLoad() {
  csound.Play();
  csound.CompileOrc(
  "schedule 1,0,5\n" +
  "instr 1 \n" +
  "a1 oscili 0.1, 440\n" +
  "outs a1,a1 \n" +
  "endin");
 }
 </script>
</head>
<body
   <!--module messages-->
  <div id="console"></div>
  <!--pNaCl csound module-->
  <div id="engine"></div>
</body>
</html>
```

There is, of course, full scope for the development of interactive controls via HTML5 tags, and to integrate other Javascript packages. A set of introductory examples and the module programming reference is found at

`http://vlazzarini.github.io`

3. SOME EXAMPLE APPLICATIONS

The following discusses a few example client-side web applications using Csound built with Emscripten or PNaCl.

3.1 Csound Notebook

The Csound Notebook[7] is an online organizer for Csound projects. Users can create Notebooks filled with Csound notes, with each note being equivalent to a Csound ORC/SCO project. The interface for note editing is designed for live coding, such that the user incrementally edits and evaluates Csound ORC and SCO code using a running Csound engine. The project is written using Ruby on Rails for the server-side, and Angular.js and PNaCl Csound for the client-side.

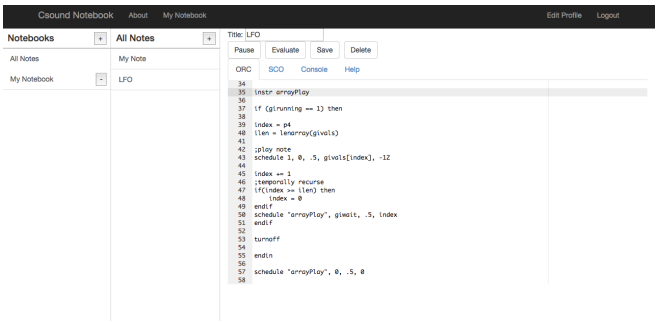

Figure 1: Csound Notebook

This project demonstrates a couple of use cases where Csound in the browser can be applied. The first use case is that a Csound user who works on the desktop or other platform wants to sketch and experiment with ideas while on the go. They can organize and experiment with their projects online and later retrieve their code to use on their desktop system. Another use case is where a Csound user wants to work with Csound but is on a computer where Csound is not installed. With the Csound Notebook web application, users do not require any plugins or applications to be installed to the user's system and can work entirely within a browser. While these use cases cater towards users who already know Csound and want to extend their use of the technology to the web, one can imagine that such a web application may also serve as a way for users who do not know Csound to try using it without having to pre-install any applications first.

3.2 Manual integration

As the number of opcodes within the Csound language is quite large, the Csound manual is a valuable resource for information about which opcodes are available to the language. Manual entries also provide examples of how to use opcodes within an orchestra file. Although it is available in other formats, the manual is distributed as a set of linked html documents. This allows the Csound Javascript library to be embedded within a manual page providing a mechanism to compile and run opcode examples directly from the manual.

In the prototype implementation shown in (fig. 2), the manual entry for the *vco2* opcode was used. Instead of static text providing an example of the opcode usage within a *csd* file, two editable text fields are provided which contain example instrument and score text. The text within each editable text field can be compiled and sent to a running instance of Csound using the provided *Send Score* and *Send Instrument* buttons. There is also an on-screen piano keyboard available, which can send score to the compiled Csound instrument along with frequency values represented by the text macro *<KEY>* within the score string.

3.3 Livecoder example

[7] `http://csound-notebook.kunstmusik.com`, source code available at `https://github.com/kunstmusik/csound-notebook`

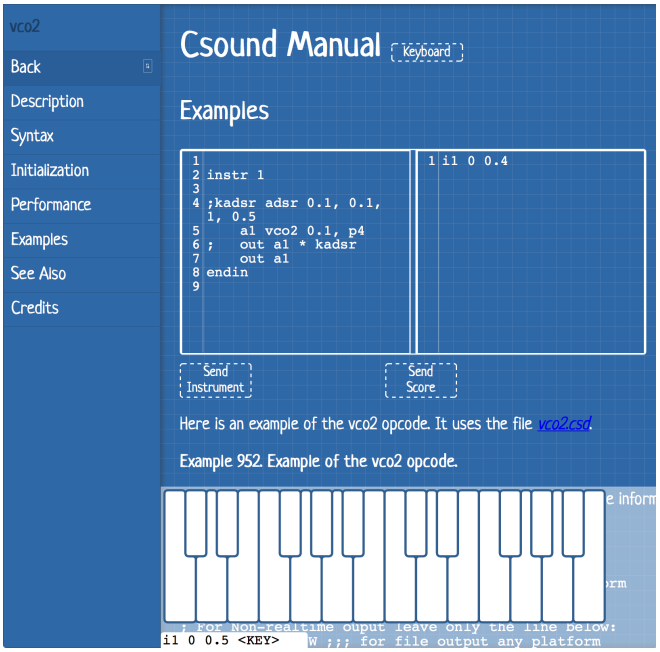

Figure 2: Csound manual integration

Figure 3: Csound's *Try it online!*

A final example of how this technology can be employed is shown in a livecoder interactive page, which is currently featured as a *Try it online!* item in the Csound community github page[8] (fig. 3). This page includes, as one of its main components, a html5 `<textarea>` element, which can be edited to hold Csound orchestra code. The code is passed to the `csound.CompileOrc()` function, which compiles it on-the-fly. In complement to this, the page also allows users to upload files to be used by the engine, and to enable audio capture for realtime processing.

This example also highlights the educational aspects of the technology, which allow the design of online, distance/blended learning initiatives for computer music and programming. This is being incorporated in new courses such as the DSP Eartraining programme[2], developed at NTNU Trondheim, in Norway.

4. THE CSOUND APPLICATION ECOSYSTEM

The presence of Csound on the web, be it as a client or as a server application, is a part of a wider application ecosystem, which is also integrated by software running on desktop, mobile, small and embedded systems, and servers. The development of this ecosystem has been founded on the presence of an API, which has been a key feature of the Csound system since version 5, launched in 2006 (although earlier releases had already shipped with an incipient API).

Users developing multimedia applications and musical works benefit in a number of ways by using Csound. Learning one music system that can be applied to multiple musical problem spaces increases the value of that knowledge. For example, because Csound renders ORC and SCO code the same on each platform, users need only modify their projects for the platform-specific parts, such as their graphical user interfaces. This allows the user to leverage their existing framework for musical computing and focus on the unique features of each platform.

From the perspective of the existing Csound user, the web offers numerous features, such as easy deployment of applications, as well as long-term preservations of works. For example, if a Csound user creates a web-based application, they are able to share it with non-Csound users without the end user having to install Csound or other dependencies. The only requirement is that they have a browser that supports Javascript and optionally PNaCl. Having easy to reproduce projects greatly simplifies the dissemination of a work. Also, for a Csound-only project, the project can be preserved indefinitely by creating a web version of the piece. Not only is the entire project preserved, but also the specific version of Csound.

[8] `http://csound.github.io`

Finally, for non-Csound users looking to develop music applications for the web, using Csound offers numerous benefits. By developing a web-based music project with Csound code, users have options to create desktop, mobile, and embedded applications reusing their Csound code. Csound also offers a rich library of unit generators, giving a large foundation on which to build upon. Lastly, having a long history, users learning Csound have a wealth of examples to draw upon for inspiration for their own work.

5. FUTURE PROSPECTS

Csound on the web is an important platform for the Csound community. The current Emscripten and PNaCl builds are done using the same source code as is used for the desktop and mobile releases. Csound development currently takes into account all platforms and plans are to continue to support each system equally. As a result, improvements made in the main codebase are automatically shared with all platform builds, and the entire ecosystem progresses together.

For platform-specific code, the CsoundObj API is re-written for each platform in the native language of the platform. This API is offered to help facilitate easier cross-platform development. Future plans are to create a full CsoundObj implementation for the web that will match closely in features to the Android and iOS versions. It is also planned to explore making CsoundObj delegate to either PNaCl or Emscripten builds of Csound, depending on what is available in the user's browser. Having a unified CsoundObj API would then allow users to depend on a single API to develop against that would work across browsers.

6. CONCLUSIONS

The Csound computer music platform has been available for composition, research, and musical application development on the desktop, mobile, and embedded platforms. In this paper, we have shown two implementations of Csound for the web, one using Emscripten and another using PNaCl, that extends the existing Csound ecosystem into the browser. This research explores not only the possibilities of web-based music applications, but also the benefits of extending existing systems to the web.

7. 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.

8. REFERENCES

[1] P. Batchelor and T. Wignall. BeaglePi: An Introductory Guide to Csound on the BeagleBone and the Raspberry Pi, as well other Linux-powered tinyware. *Csound Journal*, (18), 2013.

[2] O. Brandtsegg, S. Saue, J. P. Inderberg, A. Tidemann, V. Lazzarini, J. Tro, H. Kvidal, J. Rudi, and N. J. W. Thelle. The Development of an online course in DSP eartraining. In *Proceedings of DAFx 2012*, 2012.

[3] A. Donovan, R. Muth, B. Chen, and D. Sehr. PNaCl: Portable Native Client Executables. *Google White Paper*, 2010.

[4] J. ffitch, J. Mitchell, and J. Padget. Composition with sound web services and workflows. In S. O. Ltd, editor, *Proceedings of the 2007 International Computer Music Conference*, volume I, pages 419–422. ICMA and Re:New, August 2007. ISBN 0-9713192-5-1.

[5] T. Johannes and K. Toshihiro. „Và, pensiero!“ - Fly, thought! Experiment for interactive internet based piece using Csound6 . `http://tarmo.uuu.ee/varia/failid/cs/pensiero-files/pensiero-presentation.pdf`, 2013. Accessed: February 2nd, 2014.

[6] V. Lazzarini, E. Costello, S. Yi, and J. Fitch. Csound on the Web. In *Linux Audio Conference*, pages 77–84, Karlsruhe, Germany, May 2014.

[7] V. Lazzarini, S. Yi, and J. Timoney. Digital audio effects on mobile platforms. In *Proceedings of DAFx 2012*, 2012.

[8] V. Lazzarini, S. Yi, J. Timoney, D. Keller, and M. Pimenta. The Mobile Csound Platform. In *Proceedings of ICMC 2012*, 2012.

[9] C. Roberts, G. Wakefield, and M. Wright. The Web Browser As Synthesizer And Interface. *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2013.

[10] S. Yi and V. Lazzarini. Csound for Android. In *Linux Audio Conference*, volume 6, 2012.

[11] A. Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, pages 301–312. ACM, 2011.

Appendix H

Web Audio: Some Critical Considerations

Original Publication:

Victor Lazzarini, Steven Yi, and Joseph Timoney. Web Audio: Some Critical Considerations. In *Proceedings of the VI Ubiquitous Music Workshop*, Växjä, 2015.

Web Audio: Some Critical Considerations

Victor Lazzarini, Steven Yi and Joseph Timoney

[1]Sound and Music Research Group
Maynooth University
Maynooth, Co. Kildare Ireland

victor.lazzarini@nuim.ie, stevenyi@gmail.com

Abstract. *This paper reviews the current state of the Web Audio API, providing some critical considerations with regard to its structure and development. Following an introduction to the system, we consider it from three perspectives: the API design; its implementation; and the overall project directions. While there are some very good aspects to the API design, in particular its concise and straightforward nature, we point out some issues in terms of usage, Javascript integration, scheduling and extensibility. We examine the differences in browser implementation of builtin nodes via a case study involving oscillators. Some general considerations are made with regards the project direction, and in conclusion we offer a summary of suggestions for consideration and further discussion.*

1. Introduction

The Web Audio API[Adenot and Rodgers 2015] is a framework aimed at providing sound synthesis and processing as part of Javascript engines embedded in World-Wide Web (WWW) browser software. Such functionality had been previously only partially explored via plugin systems such as Adobe Flash. Since the introduction of the audio element in the HTML5 specification, basic streaming audio playback has been possible, but this has not been developed significantly to allow for more complex sound computing applications. These include capabilities provided by game engines, and by desktop audio software (such as mixing, processing, filtering, sample playback, etc.). The aim of the Web Audio API is to support a wide range of use cases, which is acknowledged to be a very ambitious proposition.

In this paper, we would like to raise a number of questions with regards to this framework, and explore some issues that have been left so far unresolved. The WebAudio API has seen some significant changes in the past two years, and is being strongly supported by the major browser vendors[Wyse and Subramanian 2013]. It has also been the main focus of a major international conference (the Web Audio Conference at Ircam, Paris [Ircam 2015]), where a number of projects employing this technology have been showcased (for a sample of software using the API, please refer to [Roberts et al. 2013], [Lazzarini et al. 2014], [Lazzarini et al. 2015], [Mann 2015], [Wyse 2015], [Monschke 2015], and [Kleimola 2015]). While these developments bring some very interesting possibilities to audio programming and to Ubiquitous Music, we feel it is important to consider a number of aspects that relate to them in a critical light.

Our interest in the Web Audio API is twofold: firstly, we hope it will eventually provide a stable environment for Music Programming, and add to the existing choice

of maturely-developed Free, Libre and Open-Source (FLOSS) sound and music computing systems (such as SuperCollider[McCartney 2015], Pure Data[Puckette 2015], and Csound [Ffitch et al. 2015]); secondly, we would like it to provide the supports we need to host efficiently a Javascript version of Csound [Lazzarini et al. 2015]. In the light of this, we would like to explore some of the issues that are currently preventing one or the other, or both, to come to fruition.

The paper poses questions that relate to a number of perspectives. From a technical side, we would like to discuss points of Application Programming Interface (API) design, and the split between builtin, natively-implemented, components (nodes) with Javascript interfaces, and the user-defined, pure-Javascript, elements which include the `ScriptProcessorNode` and the upcoming `AudioWorker`. We evaluate the current API according to requirements to meet various musical use cases, and see what use cases are best supported and what areas where the current API may present problems.

Complementing this analysis, we consider the issue where the Web Audio native components are implemented by the vendors in different ways, based on a specification that is open to varied interpretation. Since there is no reference implementation for any of these components, different ways of constructing the various unit generators can be found. As a case study, we will look at how the `OscillatorNode` is presented under two competing Javascript platforms, Blink/Webkit (Chrome) and Gecko (Firefox). We aim to demonstrate how these not only use different algorithms to implement the same specification, but also lead to different sounding results.

From a project development perspective, we have concerns that there is not a unified direction, or vision, for Web Audio as a system. Extensibility appears to be provided as an afterthought, rather than being an integral part of the system. This is exemplified by how the `ScriptProcessorNode` was provided to users with some significant limitations. These are due to be addressed with the appearance of the `AudioWorker`, whose principles are discussed in this paper. We also observe how the long history of computer music systems and languages can contribute to the development of the Web Audio framework.

2. The API and its design

The Web Audio API has been designed in a way that allows simple connections between audio processing objects , which are called `AudioNodes` or just *nodes* in this context. These connections are simply performed by a single method (`connect()`) that allows the output of one node to be put to another node. These objects all live within a `AudioContext`, which also provides the end point to the connections (physically, the system sound device), the `AudioContext.destination`. Aspects such as channel count are handled seamlessly by the system, and obey a number of basic rules in terms of stream merging or splitting. Such aspects of the API are well designed, and in general, we should commend the development team for the concise and straightforward nature of its specification.

In the API, the audio context is global: it controls the overall running of the nodes, having attributes such as the current time (from a real time clock), the final audio destination (as mentioned above), sample rate, and performance state (suspended, running, closed). This design can be contrasted with the approach in some music programming

systems such as Csound and SuperCollider, where local contexts are possible, on a per-instance/per-event basis. Allowing such flexibility can come with a cost of increased complexity in the API, but at the level at which the framework is targeted, it might be something that could be entertained.

In general, it is possible to equate Web Audio nodes with the typical unit generators (ugens) found in music programming systems. However there are some significant differences. One, which was pointed out in [Wyse and Subramanian 2013], is that there are two distinct types of nodes: those whose 'life cycle' are determined by start and stop commands, and those whose operation is not bound by these. This leads to a two-tier system of ugens, which is generally not found in other music programming systems. In these, the classification of ugens tends to be by the type of signal they generate, and in some cases by whether they are performing or non-performing (ie. whether they consume or produce output signals in a continuous stream). Such differences have implications for programming in that nodes that are 'always-on' can be more or less freely combined into larger components that can themselves be treated as new nodes, whereas the other type is not so amenable to this type of composition. This is not an optimal situation, as ideally, programmers should be able to treat all nodes similarly, and apply the same principles to all audio objects being used.

A related difficulty in the design is the absence of the concept of an instrument, which has been a very helpful element in other music programming systems. In these, they take various forms: patches (PD), synthDefs (SuperCollider), and instruments (Csound). They provide useful programming structures for encapsulating unit generators and their connecting graphs. In some senses, nodes that are activated/deactivated via start-stop methods implement some aspects of this concept, namely, the mechanisms of instantiation, state and performance. But in most other systems, instruments are programming constructs that are user-defined, encapsulating instances of the ugens that compose it. In other words, they sit at a different level in the system hierarchy. While we might be able to introduce the concept via a Javascript class, this is perhaps more cumbersome than it needs to be. The concept of an instrument could also allow the introduction of local contexts.

From another perspective, the Web Audio API does not offer much in terms of lower-level access to audio computation. For instance, users do not have access to the individual data output from nodes (outside the `ScriptProcessor` or `AudioWorker` nodes). It is not possible to control the audio computation at a sample or sample-block level, something that audio APIs in other languages tend to provide (e.g. PyO[Bélanger 2015] or the SndObj[Lazzarini 2008] library for Python). Such access would allow a better mix between natively-implemented nodes and Javascript ones.

2.1. ScriptProcessor and AudioWorker

The `ScriptProcessorNode` interface has been present in the API since the first published working draft (in the form of a `JavaScriptAudioNode`, as it was called then). The main aim of this component was to provide a means of processing sound through Javascript code, as opposed to the natively-compiled bultin nodes. This is currently the only means of accessing the individual samples of an audio signal provided by the API, but it sits awkwardly amongst the other built-in nodes, which are opaque.

More importantly, script processor code is run in the Javascript main thread, and asynchronously to the other nodes. It communicates with the rest of the audio context through `AudioBuffer` objects, and if these are not of sufficient size, dropouts may occur. Higher latencies are then experienced as the result of this. In addition, any interruption by, for instance, user interface events, can result in dropouts. These characteristics render the `ScriptProcessorNode` unsuitable for applications which require a robust system. They limit significantly the extendability of the system. Given that Web Audio is quite limited in terms of its offer of builtin nodes (if compared to other music programming systems), this represents a significant issue at the time of writing.

In order to rectify the problems with the script processor, a new node interface has been introduced in the latest Web Audio API editor's draft [Adenot and Rodgers 2015], the `AudioWorkerNode`. This follows the model defined for the Web Worker specification [Hickson 2014], which describes an API for spawning background threads to run in parallel with the main page code. The Audio Worker has two sides to it: the one represented by `AudioWorkerNode` methods, visible to the main thread; and another that is provided in the actual worker script that processes the audio. This is given by an `AudioWorkerGlobalScope` object, which allows access to the input and output audio buffers and other contextual elements. A script is passed to the Audio Worker on creation, and is run synchronously in the audio thread (rather than in the main thread as the script processor did). In the cases where the WebAudio implementation places this thread on high priority, using the Audio Worker will mean a demotion to normal priority, as for security reasons, Javascript user code is not allowed to run with higher than normal priority. Also, the specification dictates that the processing script cannot access the calling audio context directly. The key configuration parameter of the sampling rate is passed to the script as a readonly element of the `AudioWorkerGlobalScope` interface.

Since no actual implementation of the AudioWorkerNode exists at the time of writing, it is not possible to assess its performance. There are some indications that it might provide a more robust means of extending the Web Audio API, but some aspects of its design (such as the separation between the script context and the calling audio context) may limit it to some use cases. We understand this to be motivated by security reasons (as many of the design decisions in Javascript engine-provided APIs have to be), but inevitably it is a limitation of the current specification.

In providing Audio Workers, the editors of the Web Audio API are marking the `ScriptProcessor` node as deprecated. However, some applications for script processors might still be found, and so it could be advisable to keep providing this interface in future versions of the system.

2.2. AudioParams

AudioParams are exposed as parameters for AudioNodes. AudioParams can have a single value set, can be connected to from other nodes, or also automated with values over time. While the first two ways of setting values seem to align well with the rest of the API, the third option of automating values via function calls is somewhat of an outlier. Since automation times and values are set directly on the AudioParam itself, the curve values can not be shared with multiple params. Instead, if one wants to use the same automation values, one has to set the values for each parameter.

In systems such as Csound and SuperCollider, time-varying values using piecewise segment generators are often done using unit generators designed for that purpose. Within the context of WebAudio, a similar implementation could have been done by creating an AutomationNode. By using a node, the values of the automation could then be connected to multiple AudioParams. In that regards, the design of AudioParams adds another node-like source of values in the graph that is implicitly connected, rather than explicitly done so like other node inputs.

The user is certainly able to create and use their own automation nodes by implementing them in Javascript. This would also allow one to create other types of curves and means of triggering than those provided by the AudioParam API. However, since this appears to be a very basic functionality that could well be encapsulated as a node, it appears that it would be best handled by an addition to the API.

2.3. Scheduling

Scheduling issues are also worthy of note. In many similar systems, an event mechanism is provided or implemented behind the scenes. In Web Audio, there is no event data structure to schedule. Instead, as we have discussed above, the API encourages creating a graph of nodes, then using `start()` and `stop()` functions to turn on and off the nodes at a given time, relative to the `AudioContext` clock. For ahead of time scheduling of events, this requires all future nodes to be realised. This is inefficient in terms of memory, but does give accurate timing. This appears to be a known issue that is being tracked by the development team.

So in this case, it is expected that users will try and implement their own event system. If this is the case, and nodes are used as-designed, it is possible to do this currently in Javascript via the `ScriptProcessorNode`. Scripts run inside these nodes *do* have access to the AudioContext, and so can create new nodes. However, timing is jitter-prone, as the ScriptProcessor is processed asynchronously from the audio thread. Also, the jitter is unbounded; the Javascript main thread can end up completely paused due to other processing or due to things like the page being backgrounded. Chris Rodgers has proposed a solution [Rodgers 2013], which is similar to the one proposed by Roger Dannenberg and Eli Brandt [Brandt and Dannenberg 1999]. However, this is not an accurate solution in that it does not guarantee reproducible results. It might be sufficient for many real-time scenarios, but not when processing may require sample-accurate timing. It is not appropriate for non-realtime scenarios.

As we have seen above, the new `AudioWorker` proposes Javascript-based processing code that is run synchronously in the audio thread. This would allow accurate event system to be written, but the problem is that in this case `AudioContext` is not available to the script run under this mechanism. That means even if you wrote a scheduler, you could not create nodes running in an AudioContext that is external to it. In this scenario, one is probably better off not using any of the nodes in WebAudio, and instead doing everything in Javascript. This abandons using any of the built-in nodes, but trades off for accuracy and reproducibility across browsers (which is not guaranteed with Web Audio code, see section 3). As noted above, there is an element of speculation in this discussion, however, as `AudioWorker` is only a specification at this moment. It is unknown whether the audio context will eventually be made available to AudioWorkers.

2.4. Offline rendering

As part of the current Editor's draft of the Web Audio specification, we see the presence of new audio context interface, represented by `OfflineAudioContext`. This is a welcome addition, which would allow non-realtime use cases to be addressed. It provides a means of running nodes asynchronously which are not dependent on the need of delivering samples in a given time period, so slow processes could be rendered through this method (and buffered for playback when needed). It writes the output of the process to memory (as an `AudioBuffer` object), and if the final destination of these is a file, then this has to be separately handled by Javascript and HTML5. It appears to provide much needed support for processing that is not designed for realtime audio. However at the time of writing, it is not possible to assess it in a more thorough way since it is still at a specification stage.

2.5. Extensibility

While support for Javascript-based extensions to the system exist, as discussed in section 2.1, there is no indication of plans or proposals for means of extending the system via natively-compiled nodes. Such components would be useful for two reasons: they would allow computationally-intensive processes to take advantage of implementation-language performance; and they would provide a simple means of porting existing code into web applications. Current estimates of difference between optimised Javascript code and native code plugins performing the same tasks indicate a slowdown by a factor of ten [Lazzarini et al. 2015], so the first point above is clearly justified. The second is similarly valid considering the wealth of open-source code for audio processing algorithms that exists in C or C++ forms.

It would be interesting, for instance, if the efforts that have been put in the Native Client (NaCl) [Yee et al. 2009] open-source project could somehow be incorporated into WebAudio via a well-defined interface maybe through a dedicated node. There has been some indication that this might work, as a user-level integration of the two via the script processor has been reported as functional, albeit with some significant issues, for instance in terms of added latencies in the audio path [Kleimola 2015]. The Portable Native Client (PNaCl) plugin system has been proved to be very useful for audio processing, for example, in one of the ports of the Csound system to the Web [Lazzarini et al. 2015].

One of the key aspects of the NaCl system is that it has been shown to be a secure way of extending Javascript applications [Sehr et al. 2010]. Given that many of the constraints to improving the support to lower-level programming in Web Audio appear to relate to security concerns, it appears that NaCl, in its PNaCl form, might provide a suitable environment for extensibility. The provision of an interface for NaCl could therefore provide a very powerful and secure plugin system for the API.

3. Implementation issues

The Web Audio API specification is implemented by browser vendors in different ways. Since the source code for the audio implementation does not stem from a unique upstream repository, such differences can be considerable. In order to explore this issue in a limited but detailed fashion, we have chosen to concentrate on a particular case study. We understand, from informal observations, that the differences discussed here may extend well

beyond this particular example. For instance, we have discovered that a certain browser (Safari) appears to apply a limit of -12dB for full scale audio, whereas other browsers, such as Chrome and Firefox, do not (allowing not only a 0dB full scale, but also not making any efforts to prevent off-scale amplitudes). However it is beyond the scope of this paper to provide a complete assessment of implementation issues. We have chosen two popular browser lines for this test, Google Chrome and Mozilla Firefox, which will provide a sample of the possible differences both in source code implementation and in sonic result.

3.1. Case study: the Oscillator node

In this case study, we have written a very simple Oscillator-based instrument consisting of an OscillatorNode connected to the output, in this case, producing a sawtooth wave:

```
var audioContext;
var freq = 344.53125,  end= 10, start = 1;
var oscNode = audioContext.createOscillator();
oscNode.type="sawtooh";
oscNode.frequency.value = freql
oscNode.connect(audioContext.destination);
oscNode.start(audioContext.currentTime + start);
oscNode.stop(audioContext.currentTime + start + end);
```

All signals had an $f_0 = 344.53125$, which at $f_s = 44100$ means 128 complete cycles in 16384 samples. This was used as the size of our DFT frame for analysis. The above program was run under the Chrome and Firefox browsers. We plotted the magnitude spectra for the sawtooth waves in figs 1 and 2 (Chrome and Firefox outputs, respectively), and their absolute difference in fig 3.

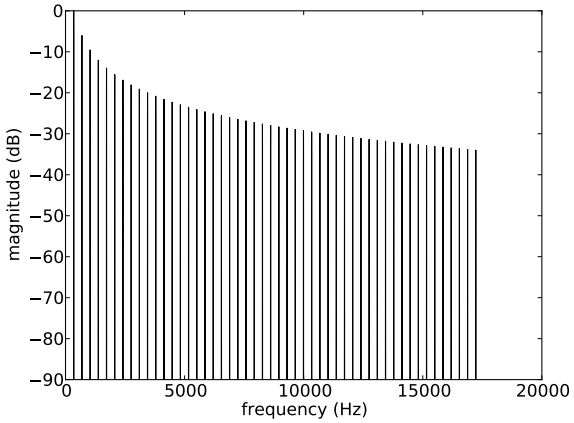

Figure 1. The magnitude spectrum of a sawtooth wave generated by Chrome

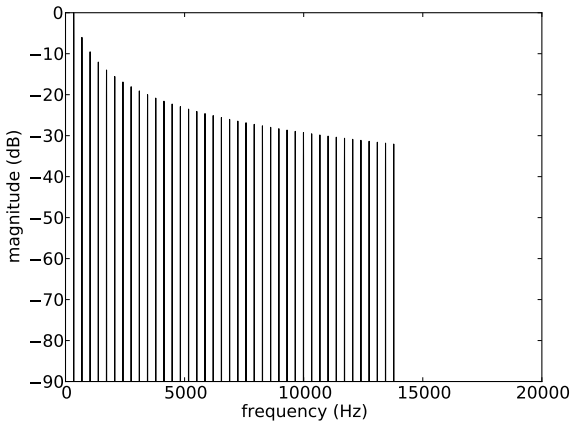

Figure 2. The magnitude spectrum of a sawtooth wave generated by Firefox

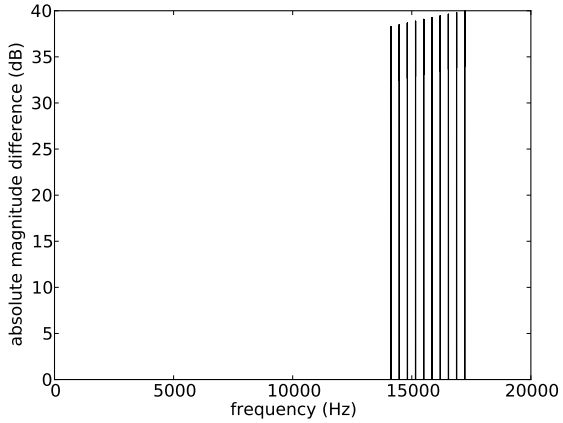

Figure 3. The absolute difference of the magnitude spectra of two sawtooth waves generated by Firefox and Chrome

In addition, we run the same program with `oscNode.type="square"` and plotted the results of the individual magnitude spectra in figs 4 and 5, as well as their absolute difference in fig 6.

From these plots, it is clear that at the high end of the spectrum, we have significantly different signals, as the Firefox output is quite drastically bandlimited, yield a difference of around 37-40dB between the two in the ten highest partials (sawtooth wave, five in the square wave case). Examining the source code for these two implementations of the Web Audio spec, we see that while the Chrome implementation uses a wavetable algorithm for implementing bandlimited versions of classic analogue waves, the bandlimited impulse train (BLIT) [Stilson and Smith 1996] method is used in Firefox. The Chrome implementation is much richer in harmonics, due to its use of three wavetables per octave over twelve octaves, which covers quite a lot of the spectrum up to the Nyquist frequency. In addition to the differences plotted here, we noticed the presence of a very low-frequency component (not visible in the figures above), which is present in the Firefox OscillatorNode signal as an artefact of the way BLIT is implemented.

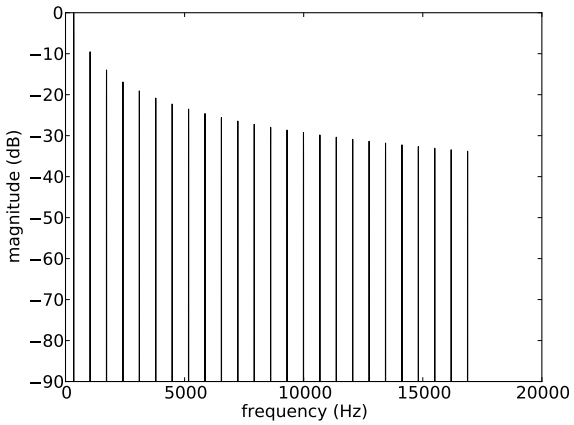

Figure 4. The magnitude spectrum of a square wave generated by Chrome

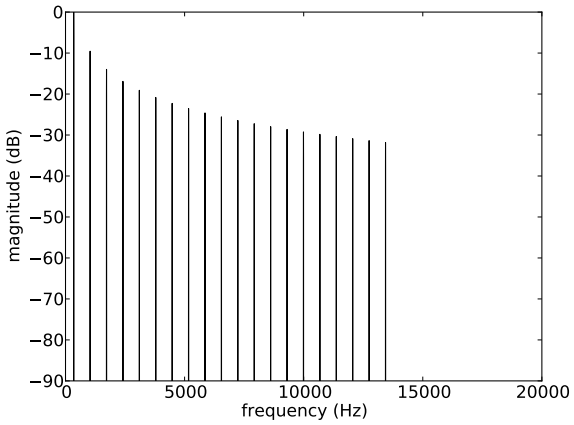

Figure 5. The magnitude spectrum of a square wave generated by Firefox

The differences discussed here stem from these implementations being, in sound and music computing terms, two clearly distinct unit generators. In a system such as Csound, with over 1,800 such components, they are assigned two different names (in this case, `vco` and `vco2`, also with slightly different parameters reflecting the particular methods used). The WebAudio specification is not definitive enough to prevent such deviations, and maybe not wide enough to accommodate them in a more suitable way. While we understand the desire to be succinct, we also note that the experience of the existing systems could have been used to inform the design of the API. Clearly, if we are to allow different implementations of bandlimited oscillators (and there are many of them), then we need to provide ways that users can distinguish between them. The development of Computer Music has been one in which precision and audio quality were always first-class citizens, and it is reasonable to expect these standards to be maintained in such an important software project.

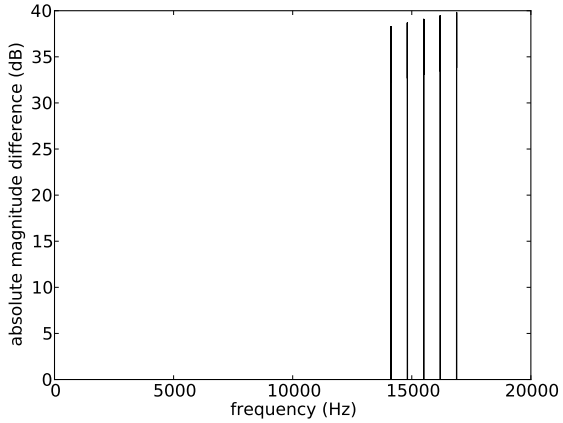

Figure 6. The absolute difference of the magnitude spectra of two square waves generated by Firefox and Chrome

As builtin nodes can differ, it is not possible to create consistent results across browsers. An alternative to this of course is to use Javascript-programmed audio code (either directly or via systems like Csound) to ensure the same results everywhere. It is also important to note that issues such as this are not confined to Web Audio, as differences in interpretations are not new to web applications. For instance, on the graphics side, browsers have long been known to render web pages differently (types, in particular, are an issue[Brown 2010]). However, this is widely acknowledged to be a less than desirable scenario.

4. Project directions

The Web Audio project is clearly a very significant project, which has been managed in an open way, through accessible code repositories, and a well-supported issue tracking system. Discussions on its directions have been carried out in open fora, and the main team members seem to take heed of user suggestions. On the other hand, the points made in this paper may indicate a certain lack of awareness of the fifty years of computer-based digital audio technologies. The history of computer music languages is rich in examples of interesting ideas and concepts [Lazzarini 2013], and these could be very useful to the design of WebAudio. Interestingly, developers seem to be well aware of commercially-available closed-source music software. Proprietary multitrack and MIDI programs Logic and GarageBand, for instance are name-checked in the Web Audio specification document[Adenot and Rodgers 2015], even though the functionality and use-cases of the API are closer to FLOSS music programming systems.

One way in which the project could take advantage of the wealth of ideas in FLOSS Computer Music systems is to develop a reference implementation for unit generators/nodes, based on source code that is openly available and well documented. This could be a way of addressing the issues raised in section 3, and a means of making good use of existing technology. Furthermore, a review of such systems could inform the decisions taken by the team in terms of steering the future directions of the API. Contributors to the discussion fora have already been bringing ideas that stem from academic research in the area, in an informal way. This could be enhanced by structured and systemic study that could be carried out as part of the development work.

5. Conclusions

The Web Audio framework is a very welcome development in audio programming, as it provides a number of potential applications that were previously less well supported. However, there are some key issues in its current implementation, and in its design, that need to be addressed, or at least, considered. On one hand, users of the framework should be made aware of these so that they can make informed decisions in the development process; on the other, developers might want to pay attention to the ones that can be addressed in some way. Our aim with this paper is to be able to contribute to the debate in the area of programming tools, so that support for a variety of approaches in music systems development is enhanced. From this perspective, we would like to offer the following summary of suggestions:

- The introduction of an *instrument* interface to enhance composability (section 2)
- Further flexibility for Audio Worker code (e.g. some form of access to the calling audio context) (2.1)
- New nodes, in particular one for handling control curve generation (2.2)
- More precise and flexible scheduling (2.3).
- Extensibility enhancements via native plugins (2.5).
- More precise definitions to minimise implementation differences (3).
- A reference implementation based on existing computer music systems (4).

References

Adenot, P. and Rodgers, C. (2015). Web Audio API, W3C Editor's Draft. `https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html`. Accessed: April 16, 2015.

Brandt, E. and Dannenberg, R. B. (1999). Time in distributed real-time systems. In *In Proc. Int. Computer Music Conference*, pages 523–526.

Brown, T. (2010). Type rendering: web browsers. Accessed: April 17, 2015.

Bélanger, O. (2015). PyO: dedicated Python module for digital signal processing. `http://ajaxsoundstudio.com/software/pyo/`. Accessed: April 17, 2015.

Ffitch, J., Lazzarini, V., Yi, S., Gogins, M., and Cabrera, A. (2015). Csound. `http://csound.github.io`. Accessed: April 16, 2015.

Hickson, I. (2014). Web Workers, Editor's Draft. `http://dev.w3.org/html5/workers/`. Accessed: April 18, 2015.

Ircam (2015). The 1^{st} Web Audio Conference . `http://wac.ircam.fr`. Accessed: April 16, 2015.

Kleimola, J. (2015). Daw plugins for web browsers. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Lazzarini, V. (2008). Interactive audio signal scripting. In *Proceedings of ICMC 2008*.

Lazzarini, V. (2013). The development of computer music programming systems. *Journal of New Music Research*, 42(1):97–110.

Lazzarini, V., Costello, E., Yi, S., and ffitch, J. (2014). Csound on the Web. In *Linux Audio Conference*, pages 77–84, Karlsruhe, Germany.

Lazzarini, V., Yi, S., Costello, E., and ffitch, J. (2015). Extending csound to the web. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Mann, Y. (2015). Interactive music with tone.js. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

McCartney, J. (2015). SuperCollider. `http://supercollider.github.io`. Accessed: April 16, 2015.

Monschke, J. (2015). Building a collaborative digital audio workstation based on the web audio api. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Puckette, M. (2015). Pure Data. `http://puredata.org`. Accessed: April 16, 2015.

Roberts, C., Wakefield, G., and Wright, M. (2013). The Web Browser As Synthesizer And Interface. *Proceedings of the International Conference on New Interfaces for Musical Expression*.

Rodgers, C. (2013). A Tale of Two Clocks. `http://www.html5rocks.com/en/tutorials/audio/scheduling/`. Accessed: April 17, 2015.

Sehr, D., Muth, R., Bifie, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. (2010). Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*.

Stilson, T. and Smith, J. (1996). Alias-free digital synthesis of classic analog waveformss. In *In Proc. Int. Computer Music Conference*, page 332?335.

Wyse, L. (2015). Spatially distributed sound computing and rendering using the web audio platform. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Wyse, L. and Subramanian, S. (2013). The Viability of the Web Browser as a Computer Music Platform. *Computer Music Journal*, 37(4):10–23.

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. (2009). Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 IEEE Symposium on Security and Privacy*.