M.Tech Thesis

Iain C. T. Duncan

Jul 02, 2023

CONTENTS

Iain C.T. Duncan, University of Victoria

CHAPTER

ONE

INTRODUCTION

It is not an exaggeration to say that music making was completely revolutionized by the advent of personal computers fast enough to render audio in realtime almost 30 years ago. The advances in software synthesis and audio processing in that time have been profound - audio processing capabilities that were the domain of million dollar recording budgets at the advent of the compact disc are trivial on a commodity laptop today and possible with even free software.

A side effect of the rapid progress in realtime audio computation is that many techniques used principally in the context of *non-realtime* computer music have largely fallen out of the public light. However, in the last decade computers have reached speeds such that some of the tools and approaches previously associated only with non-realtime rendering have become feasible even in a realtime scenario. One of these, the one with which I am involved, is the programming of computer music in the Lisp family of languages. With its support for symbolic processing, recursion, and iteractive development, Lisp has always been an elegant and productive way to represent and build music (CTN: Dannenberg 1997, 291). However, the accepted wisdom has been that one does not try to do anything where the same computer is generating significant amounts of digital audio in real time while running a Lisp or similar high-level, garbage-collected language. This is now not just possible, but practical even at latencies usable by serious instrumentalists (i.e., sub 20 ms), allowing it be used on stage as well as in the studio.

I believe we are, as a result, at the cusp of a renaissance of the use of high-level programming languages in music. JavaScript is used as an extension language in Max, Python and Ruby are used for algorithmic composition, and even Haskell is used in live-coding performances (CITE:Lyon 2012, 13-16; Mclean and Dean 2018, 650-651, 597, 258). And a new generation of computer musicians is discovering the elegance, the productivity, and the joy of doing this work with Lisp, a language that I and many others will argue is uniquely suited to expressing musical abstractions in code.

The Scheme for Max project is, among other things, an attempt to make music programming in a Lisp - specifically the Scheme dialect of Lisp - accessible and practical to the “regular” computer musician working in a “regular” studio. Scheme for Max (S4M) provides an extension to the Max audio/visual programming platform that embeds a Scheme interpreter, allowing the user to create fully functional Scheme programs that interact with the rest of the Max environment, and which do so with high temporal accuracy. S4M can be used to embed single-line programs directly in visual Max patches, or to build programs of thousands of lines that do everything from sequencing MIDI to controlling analog hardware and editing audio. And it can be used for music of all sorts, from complex contemporary scored music, to interactive live improvisation, to generative and stochastic algorithmic composition.

While I am far from the only one working on new ways to use Lisp in computer music (even in a realtime context), I believe Scheme for Max achieves something unique. By creating an enviroment thats runs seamlessly and flexibly inside the tremendously successful Max platform (and thus also inside the even more tremendously successful Ableton Live digital audio workstation), Scheme can be productively used in close tandem with both popular, commercial music tools and the vast library of specialized, academic, and research tools available for Max. And by the happy coincidence of Scheme’s syntax compability with Max message syntax, it can be even be used in baby-steps - new programmers can add the power of Scheme to their Max patches with only a few lines of code without even reaching for a text editor. I believe Scheme’s consistent, minimal syntax and interactive development style provide an ideal introduction to textual programming for the curious Max user. With Scheme for Max, one is not forced to choose between the power and flexibility of Lisp and the convenience and accessibility of commercial music software.

This thesis introduces the Scheme for Max project. I begin by discussing the landscape into which it fits and its precendents and inspirations. I cover the motivations and goals for the project, discussing my own musical and technical desires. I examine its design and capabilities. And finally, I conclude by evaluating how well it has succeeded, discussing its limitations, and outlining my future plans for the project.

For reasons of space, I have not covered every feature of the project, nor have I delved into the low-level implementation details and the underlying C code. I have limited myself to discussion appropriate to an interested computer musician, rather than a professional programmer or computer scientist. That said, the project is open-source and freely available, so the the interested reader is encouraged to consult the GitHub project page for further details if desired, where they will find links to comprehensive documentation, help files, demo videos, and source code. Code samples presented here in Scheme should be comprehensible to a new programmer, but if the reader requires more help, I would point them to my "Learn Scheme For Max" online e-book in which I introduce Scheme programming from first-principles, available again from the main project page at (https://github.com/iainctduncan/scheme-for-max).

CHAPTER

TWO

BACKGROUND - COMPUTER MUSIC PROGRAMMING LANGUAGES

2.1 What is Computer Music Programming?

To frame the discussion of the the motivation and goals of the Scheme for Max project, I will first briefly survey the computer music programming landscape, discussing several families of language, their approaches to programming computer music, and the advantages and disadvantages of these for various kinds of user and projects.

To begin, I would like to clarify exactly what I mean by "computer music programming". I am using the term to refer to programming in which both the tools used and the composition itself are programmed on a computer. I say this using the word "composition" in its broadest sense - the composition could be a linearly scored piece of set length and content, or an interactive program with which a performer interacts, but that contains prepared material of some kind. This is distinct, for example, from commercial music tools (such as sequencers or digital audio workstations) in which a computer program is run as the host environment, but the compositional work itself is stored as static data within that program (for example in a binary format or in MIDI files), and would not normally be considered a computer *program* itself.

Using this definition of computer music programming, we can draw a further distinction between two kinds of programming and programmer: we have programming that is specific to a composer or an artistic work; and we have programming of tools and frameworks that will be potentially be used by many people and in potentially many contexts. For the purpose of this discussion, I will refer to the people doing these as the "composer-programmer" - a term also used by Curtis Roads in "Composing Electronic Music" (CTN: Roads 2015, 341) - and the "tools programmer" (my own terminology). Of course, these frequently are the same person at different times. This distinction is important, as the goals of these two programmers, and thus the ideal design of supporting technology, can frequently be very different. The composer-programmer likely wishes the programming tools to be as immediately convenient as possible to an individual, while the tools-programmer may be developing software for use by a larger community and may prefer design decisions that favour long-term code reuse and team development at the expense of immediate convenience. As we will see, the various computer music languages, platforms, and approaches favour one of these hypothetical programmers to a greater or lesser degree. Beyond these distinctions, I use the term "computer music programming" as broadly as possible - I include as computer music programming platforms both graphic platforms with which a beginner can be immediately productive as well as general purpose programming languages that are the domain of experienced computer programmers.

2.2 Requirements for a Computer Music Platform

Prior to examining the programming platforms, let us examine some common requirements of the composer-programmer so that we can assess how the various options meets these needs. While the list of possible requirements is potentially long, I will group them into three high level requirements that are personally important to me as a composer and performer. Note that not all of these are necessarily fulfilled well, or at all, by each of the possible platforms a composer might use - some offer only partial support for a requirement, or perhaps even none at all (CTN: Lazzarini 2013, 97).

2.2.1 Support for Musical Abstractions

To be maximally productive in producing musical works, the computer music language should provide musically meaningful abstractions that support how the composer thinks about music. Examples of these abstractions would be programmatic constructs for representing notes, scores, instruments, voices, tempi, bars, beats, and the like. The more of these that are supported in the language itself, the less the programmer must decide and code themselves in order to make music. Probably the most important of these abstractions are those concerning musical time - a good computer music language must have an implementation of *logical time*, allowing the programmer to schedule events accurately in a way that makes sense to the user but that also works for rendering or playing in real-time to a degree of accuracy appropriate for music (CTN: Dannenberg 2018, 3).

However, it should be noted that the importance of these abstractions being readily provided by a language varies widely amongst composer-programmers: while some composers will be relieved not to have to decide how to implement the concept of beats and tempi themselves, others may be expressly looking for platforms that do not come with any assumptions of how one might think about musical abstractions. Computer music programming languages thus also vary widely in their areas of focus and the problems they attempt to solve for the programmer (CTN: Dannenberg 2018, 4). For example, the Csound language comes with a built-in abstraction of a score system, that in turn depends on abstractions representing beats and tempi (CTN: Lazzarini 2016, 157). In contrast, the Max platform takes the design approach of giving the user an almost blank canvas, deliberately avoiding any assumptions that the user will make music of any particular style or in any particular way (CTN: Puckette 2002, 34).

For my own purposes, I would ideally like to have the ability to score conventional music without necessarily being required to implement all the dependencies of a score system myself, but still have the flexibility of being able to implement my own or greatly alter the abstractions that come with a language when I choose to do so. My personal yard stick is to ask whether I could reasonably program and render a piece such as “Ionization” by Varese, with its shifting meters and cross-rhythms between instruments - an endeavour which would be frustratingly difficult on standard commercial tools.

2.2.2 Support for Performer Interaction

The platform should make it possible for a performer to interact with a a piece while it plays. Various computer music languages support interaction through manipulating a computer directly (through text, network input, or graphical devices), through physical input devices such as MIDI controllers and network-connected hardware, and even through audio input, allowing the perfomer to interact with the program by playing an acoustic instrument.

Again, the importance of this varies with the composer and the platform. In fact, this was not even possible with early computer music languages such as the MUSIC-N family, with which the process of rendering audio from a score file took far longer than the duration of a piece (CTN: Wang 2017, 60-63).

However, since the advent of realtime-capable systems, this has become a standard feature in computer music platforms. In fact, in contrast to the Music-N languages, the Max platform was designed to support performance-time interactivity first, with the ability to render audio only added later as computers became capable of doing both (CTN: Puckette 2002, 33). As someone who comes principally from a jazz and contemporary improvised music background, being able to create complex systems that support realtime input is a high priority to me.

2.2.3 Support for Complex Algorithms

Finally, the platform should provide support for programming algorithms of significant complexity. It should be possible to make the platform, for example, do things that are impractical or impossible for the human perfomer or for the composer working at a piano with score paper. My personal goals include creating music that combines algorithmically generated material with traditionally scored and improvised material. For my algorithmic work, I want to explore techniques and forms that would be impractical if one is individually entering notes, thus instead requiring programmatic implementation of algorithms. However, it is also important to me that a performer be able to interact with the algorithms. That, to quote Miller Puckette on his original design goals for Max, basic musical material should be "controllable physcially, sequentially, or algorithmically; if they are to be controlled algorithmically, the inputs to the algorithm should themselves be controllable in any way" (CTN: Puckette 1991, 68).

While one could certainly come up with further requirements, these three provide the basis on which I will discuss the main families of computer music programming tools, and around which we can discuss how Scheme for Max complements existing options.

2.3 Computer Music Platform Families

For the purpose of keeping this discussion within a reasonable length, I will likewise categorize the historical and currently popular computer music programmimg environments into three general categories: domain-specific textual languages, visual patching environments, and general purpose programming languages that are run with music-specific libraries or within musical frameworks.

I will briefly discuss each of these, listing various examples, but focusing on a representative tool from each family. I will provide my observations and experiences of the advantages and disadvantages of each, drawing both on the literature and on my personal experiences with tools from each category over the last 25 years.

2.3.1 Domain-Specific Textual Languages

A domain-specific language (DSL) for music is a textual programming language intended expressly for making music with a computer (CTN: Wang 2017, 58).

The first historical example of programming computer music (that one might reasonably consider as more than an audio experiment) used a music DSL, namely Max Matthew's MUSIC I language, created in 1957. MUSIC I (originally refered to as simply MUSIC) was a domain specific language written in assembly language for the IBM 704 mainframe at Bell Labs. It was able to translate a high-level textual language with musical abstractions to assembly code, and could (through various intermediary steps) output digital audio. MUSIC I was followed by various refinements by Matthews (Music II through V), and by similar languages by others. Its lineage continues to this day in the Csound language, still under active development and widely used, and one with which I have extensive experience. (CTN: Manning 2013, 187-189).

While the source code of Csound piece, for example, is clearly a computer programs (and would be recognizable as such to one familiar with programming) the way in which it turns code into music would not likely be obvious at a glance to a programmer unfamiliar with music. The language is, to a significant degree, designed around high-level abstractions suitable for particular ways of creating a composition, and has a particular way in which it is run to make the final product. Historically, running such a program meant rendering a piece to an audio file, but with modern computers (and versions of Csound) the rendering can be done in realtime. While originally these programs were not something with which a performer could interact while the music rendered, facilties now exist in Csound for performers to interact with the programs while they play (CTN: Lazzarini 2016, 171-179).

In addition to Csound, some other actively developed examples from this general family of language include SuperCollider, ChucK, and Faust, each of which has a particular focus or approach to the problems of computer music (CTN: Wang 2017, 69-72; Lazzarini 2017, 41-42).

A notable advantage of a using a music DSL is that many of the hard decisions that face the programmer have been made already. The composer-programmer is not starting with a blank slate: the language provides built-in abstractions ranging from macro-structural concepts such as scores and sections to individual notes and beats. Music DSLs thus significantly simplify the task of programming music and reduce how much the composer-programmer must learn and program to begin making music (CTN: Lazzarini 2017, 26). In Csound, for example, a program consists of an "orchestra" file, containing programmatic instrument definitions, and a "score" file, containing a score of musical events notated in Csound's own data format. (These files may be merged into one containing "csd" file, but the distinction still holds.) These are used together to render a scored piece to audio, either as an offline operation or as a realtime operation. A sample of Csound code is shown below, with an instrument playing a short scale driven by the score.

```
<CsoundSynthesizer>
<CsInstruments>

instr 1
  ; take pitch as midi note from param 4
  kfrq  mtof  p4
  ; saw wave
  asig  vco2  0.8, kfrq, 0
  ; an ADSR envelope
  aenv  0.01, 0.3, 0.7, 0.2
  ; apply env and output in stereo
  outs  asig * aenv, asig * aenv
endin

</CsInstruments>
<CsScore>

;  time   dur  midi-note-num
i1 0      1    60
i1 1      1    62
i1 2      1    64
i1 3      1    65
i1 4      1    67

</CsScore>
</CsoundSynthesizer>
```

With their built-in musical abstractions, DSL's are attractive to the composer-programmer, but on the other hand, the tools-programmer is significantly more constrained than when working in a general purpose programming language. This can be frustrating for experienced programmers coming from general purpose languages, who may wonder where their function calls and looping constructs went and how they can express the algorithms with which they are familiar in the unusual abstractions provided by the language. For example, in Csound one can program a form of recursion, but one of the techniques for doing so involves creating instruments that play notes that in turn schedule notes (CTN: Lazzarini 2016, 116). The use of the note as the fundamental unit of computation (where a "note" is an instance of an instrument definition activated at some time, for some duration) requires the tools-programmer to not only understand the concept of recursion, but to also understand how to translate it into this unusual syntax.

That said, music DSLs generally provide ways of *extending* the language with a general purpose language, allowing the tools-programmer to add new abstractions to the DSL itself. In Csound, for example, a tools-programmer may create a new *opcode* (essentially the equivalent of a Csound class or function) using the C language, compiling it such that it can be used in the same way as any built in opcode that comes with Csound (CTN: ffitch 2011b, 581).

It should also be noted that the ease with which composer-programmers can work with DSLs has led to broad popularity in the music community, and this in turn has led to many programmers creating publically available extensions, thus providing a rich library of freely-available tools for the programmer to use. Csound, for example, is still actively used

and developed today, which is remarkable for a language first developed in 1986, and now has thousands of objects available (CTN Manning 2013, 189). If an extension is popular and useful enough, it may even find its way into the main language or into official repositories of extensions.

So how does a music DSL such as Csound stack up with regard to our three high level requirements? Certainly, we are given many high-level and convenient musically-meaningful abstractions. Creating linear pieces according to a set score is straightforward. Performer interaction is also possible in modern versions, though programming an interactive system is somewhat cumbersome in that tasks that would require simple programming in a general purpose language must be done in an unusual manner to fit in the note-centered paradigm of Csound. For example, making a component to receive, parse, and translate MIDI input according to some arbitrary rules requires making an "instrument" and having the score turn on "always-on" notes (CTN: Lazzarini 2016, 175). Clearly we are bending the built in abstractions to other purposes, at the expense of easily comprehensible code.

Likewise, expressing complex algorithmic processes can be difficult. Being a textual language, expressing mathematical formulae is straightforward. But anything truly complex (for example, building a constraint system incorporating looping, sorting, and filtering) is discouragingly cumbersome. Absent regular functions and iteration, these kind of ideas can be very difficult to express, requiring a great deal of code that is subverting the design of the language.

Returning to our distinction between the composer-programmer and the tools-programmer, one could say that music DSLs are heavily optimized for the composer-programmer and for the process of composing a (relatively speaking) traditional linear piece. Or, to put it another way, Csound and its like are appropriate for making *pieces*, but cumbersome for making *programs*.

2.3.2 Visual Patching Environments

A quite different family of computer music languages comprises the visual "patching" environments, such as Max and PureData (a.k.a Pd). First created by Miller Puckett while at IRCAM in 1985, Max was designed from the outset to support realtime interactions with performers. In a typical use case, the Max program would output messages (which could be MIDI data, but were not necessarily), and these would be rendered to audio with some other tools, such as standard MIDI-capable synthesizers or other audio rendering systems. Later versions of Pure Data and Max added support for generating audio directly, as computers became fast enough to generate audio in real-time (CTN: Puckette, 2002, 34).

In Max and Pure Data, the composer-programmer places visual representations of objects on a graphic canvas, connecting them with virtual "patch cables". When the program (called a "patch") runs, each object in this graph receives messages from other connected objects, processes the message or block of samples, and optionally outputs messages or audio in response. A complete patch thus acts as a program where messages flow through a graph of objects, similar to data flowing through a spreadsheet application. The term "dataflow" has been used to describe this type of program (CTN: Farnell 2010, 149) though it should be noted that Miller Puckette himself asserts that it is not truly "dataflow" as the objects may retain state, and ordering of operations within the graph matters (CTN: Puckette 1991, 70).

As with many textual DSL's, it is possible for the advanced programmer to extend both Max and Pure Data by writing "externals" in the C and C++ languages. In Max, this facility is called the Max Software Development Kit, or SDK (CTN: Lyon 2012, 3). The popularity and extensibility of Max and Pure Data has led to thousands of patcher objects being available for Max and Pure Data, both included in the platforms and as freely-available extensions. These include objects for handling MIDI and other gestural input, timers, graphical displays, facilities for importing and playing audio files, mathematical and digital signal processing operators, and much more (CTN: Cipriani, 2019, XI).

This visual patching paradigm differs significantly from that of Csound and similar DSLs. The program created by a user is best described as an interactive environment, rather than a piece. A patch runs as long as it is open, and will continue to do computations in response to incoming events such as MIDI messages, timers firing, or blocks of samples coming from operating systems audio subsystem (CTN: Farnell 2010, 149).

In contrast to textual DSLs such as CSound, patching environments have comparatively little built in support for musically meaningful abstractions. There is no built in concept of a score, or even a note, and there is no facility for linearly rendering a piece to an audio file from some form of score data store. The programmer must build such things out of the available tools. In this sense, these environments are more open ended than most DSLs - one builds a program

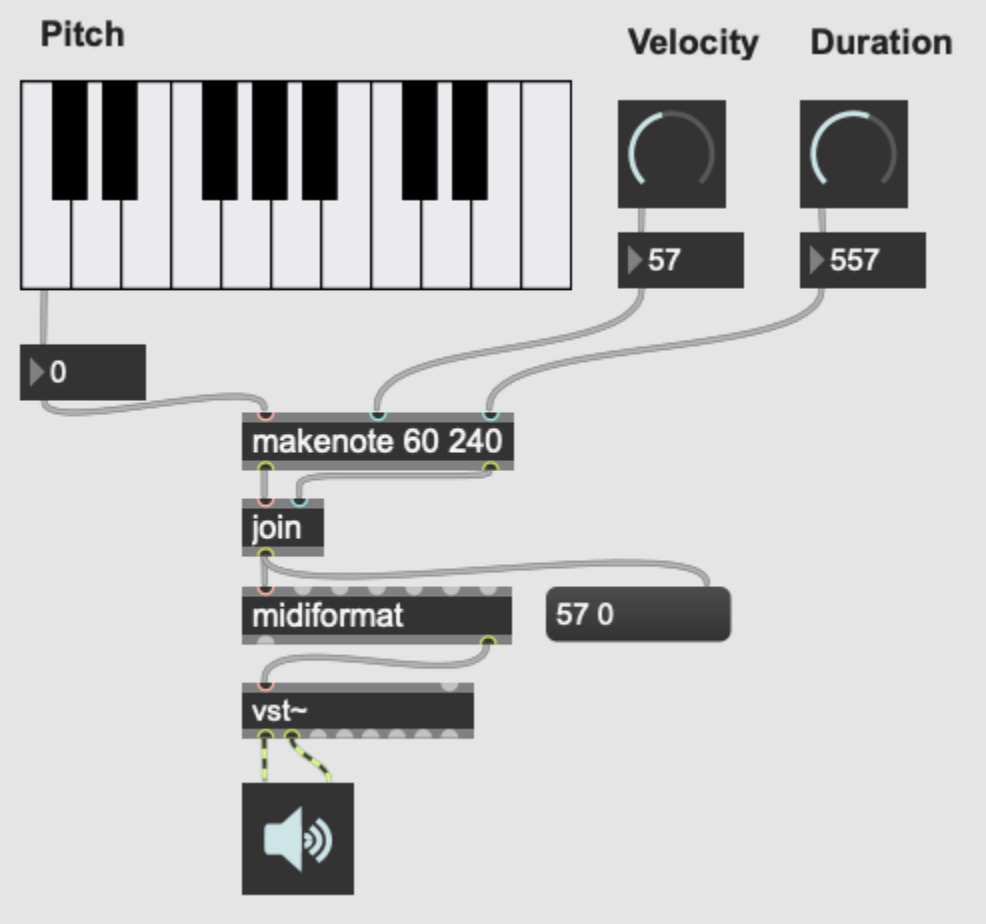

Fig. 1: Figure 1: A Max patch with a keyboard and dial user interaction objects.

(albeit in a visual manner) and this program could just as easily be used to control lighting or print output to a console in response to user actions as play a piece of music. And indeed, modern versions of Max and Pure Data are widely used for purely visual applications as well as music, through the Jitter (Max) and Gem (Pd) collections of objects. There is nothing intrinsically musical about the patcher environments - the environments are much more open ended in this way than the musical DSLs. As Max developer David Zicarelli put it in his paper on the 17th anniversary of Max, it is, compared to most programs, "a program which does nothing", presenting the user with a completely blank canvas (CTN: Zicarelli 2002, 44).

Returning to our requirements, the fundamental strength of patching environments is the ease with which one can create programs mean for performer interaction. A new programmer can realistically be making interesting interactive environments that respond to MIDI input within the first day or so of learning the platform.

However, making something that is conceptually closer to a scored piece is much more difficult than in a language such as Csound. It is most definitely possible, but it requires the programmer to be familiar with the workings of many of the built in objects, and to make a substantial number of low-level implementation decisions, such as how data for a score should be stored, what constitutes a piece (or even a note!), how playback should be controlled or clocked, and so on.

Implementing complex algorithms is also a difficult task in the patching languages. The dataflow paradigm is unusual in that it requires one to write programs entirely using side-effects. Objects do computations in response to incoming messages, which, under the hood, are indeed function calls from the source object to the receiving object, but the receiving objects have no way of *returning* the results of this work to the caller - they can only make new messages that they will pass on to downstream objects, resulting in more function calls until the chain ends. Describing this in programming terminology: the flow of messages creates a call chain of void functions, with the stack eventually terminating when there are no more functions to be called.

While easy to grasp for new programmers, this style of programming makes many standard programming practices difficult to implement, such as recursion, iteration, searching, and filtering. Thus, much like the musical DSLs, but for a different set of reasons, complex algorithms that would be straightforward in a general purpose programming language can require significant and non-obvious programming.

2.3.3 General Purpose Programming Languages

Our third family of computer music programming languages is that of general purpose programming languages (GPPLs), such as C++, Python, JavaScript, Lisp, and the like. The use of GPPLs for music can be divided broadly into two approaches, corresponding to the mainstream software development approaches of developing with libraries versus developing with inversion-of-control frameworks.

In the library-based approach, the programmer works in a general purpose language, much as they would for any software development, and uses third-party musically-oriented libraries to accomplish musical tasks. In this case, the structure and operation of the program is entirely up to the programmer. For example, a programmer might use C++ to create an application, creating sounds with a library such as the Synthesis Tool Kit (CTN: Cook 2002, 236-237), handling MIDI input and output with PortMidi (CTN: Lazzarini 2011, 784-795), and outputing audio with the PortAudio library (CTN: Maldanado, 2011, 364-375). While the use of these libraries significantly reduces the work needed by the programmer, fundamentally they are simply making a C++ application of their own design.

In the second approach, a general purpose language is still used, but it is run from a muscially-oriented host, which could be either a running program or a scaffolding of outer code (i.e., the host is in the same language and code base but has been provided to the programmer). The term "inversion-of-control" for framework-based development of this type refers to the fact that the host application or outer framework controls the execution of code provided by the programmer - the programmer "fills in the blanks", so to speak. Non-musical examples of this are the Ruby-on-Rails and Django frameworks for web development, in which the programmer need provide only a relatively small amount of Ruby or Python code to create a fully functional web application. A musical example of this is the Common Music platform, in which the composer-programmer can work in either the Scheme or Common Lisp programming language, but the program is executed by the Grace host application, which provides an interpreter for the hosted language, along with

facilities for scheduling, transport controls, outputting MIDI, and so on (CTN: Taube 2009, 451-454). The framework-driven approach thus significantly decreases the number of decisions the programmer must make and the amount of code that must be created, while still preserving the flexibility one gaines from working in a general purpose language.

While the framework-oriented approach is less flexible than the library-oriented approach, given the programmer must work within the architectural constraints imposed by the framework, the strength of GPPLs compared to either textual DSLs or visual patching platforms is in both cases flexibility, especially with regard to implementing complex algorithms. With a general purpose language, the programmer has far more in the way of programming contructs and techniques available to them. Implementing complex algorithms is no more difficult than it is in any programming language. Looping, recursion, nested function calls, and complex design patterns are all practical, and the programmer has a wealth of resources available to help them, drawing from the (vastly) larger documentation available for general purpose languages.

Of course, this comes at the cost of giving of the programmer both a great deal more to learn and a lot more work to do to get making music. In the library-based approach, it is entirely up to the programmer to figure out how they will go from an open-ended language to a scored piece, and even in the framework-driven approach, the programmer begins with much more of a blank slate than they typically do with a musical DSL.

General purpose languages are thus attractive to composers wishing to create particularly complex algorithmic music, or to those wishing to create sophisticated frameworks or tools of their own that they may reuse across many pieces. With general purpose languages, the line between composer-programmer and tools-programmer is blurred and managing this division is one of the tricker problems with which the programmer must wrestle.

General purpose languages can also provide rich facilities for performer interaction, but again, at the cost of giving the programmer much more to build. Numerous open-source libraries exist for handing MIDI input, listening to messages over a network, and interfacing with custom hardware. However, the amount of work and code required to use these is significantly higher than doing the same thing in a patching environment. It is worth noting that, *relatively speaking*, the additional work required decreases as the complexity of the desired interaction grows. Given a sufficiently complex interactive installatation, at some point the tradeoff swings in favour of the general purpose language. Where precisely this point is depends a great deal on the expertise of the programmer - to a professional C++ programmer, the savings of using a patching language may be offset by the power of the (C++) development tools with which the programmer is familiar.

2.3.4 Multi-Language Platforms

Finally, we have what is, in my personal opinion, the most powerful approach to computer music programming: the multi-language, or hybrid, platform. As programming tools and computers have improved, it has become more and more practical to make computer music using more than one platform at a time in an integrated system.

This multi-language approach has been explored in a wide variety of schemas. The simplest is that of taking the output from one program and sending it as input to another. With the Csound platform, this is straightforward: instrument input, whether real-time or rendered, comes from textual score statements, and these can be created by programs made in other languages that either write to files or pipe to the Csound engine (ffitch 2011a, 655). In many modern platforms tighter integrations are now possible through application programming interfaces (APIs) that let languages directly call functions in other languages, as they run. One can, for example, run Csound from within a C++ or Python program, interacting directly with the Csound engine using the Csound API (CTN: Gogins, 2013, 43-46). One can also run a DSL such as Csound inside a visual patcher, using open-source extensions to Max and Pd that embed the Csound engine in a Max or Pure Data object (CTN: Boulanger 2013, 189). And one can even run a general purpose languauge *inside* a DSL or visual platform, such as Python inside Csound (CTN: Ariza 2009, 367). or JavaScript inside Max (CTN: Lyon, 13).

Note that a multi-language platform differs from the previously discussed practice of *extending* a patching language or DSL with a GPPL such as C or C++. In the multi-language hybrid scenario, the embedded GPPL is used by the *composer-programmer* to make potentially piece-specific code, rather than solely by a tools-programmer who is creating reusable tools in the environment's extension language. (It should be mentioned, however, that it is feasible to

prototype algorithms in an embedded high-level language such as JavaScript and port them later to a DSL's extension language, should they reach sufficient complexity and stability to warrant the low-level work.)

In the hybrid scenario, the combination of the various platforms provides the programmer with a tremendous amount of flexibility (CTN: Lazzarini 2013, 108). One can, for example, use visual patching to quickly create a performer-interaction layer, have this layer interact with a scored piece in the CSound engine, and simultaneously use an embedded GPPL to drive complex algorithms that interact with the piece.

The cost of this approach is simply that it requires the programmer to learn more - a great deal more. Not only must they be familiar with each of the individual tools comprising the hybrid, but they must also learn how these integrate with each other. This necessitates not just learning the integration layer (e.g., the nuances of the csound~ objects interaction with Max), but likely also understanding the host layer's operating model in more depth than is required of the typical user. For example, synchronizing the Csound score scheduler and the Max global transport requires knowing each of these to a degree beyond that required of the regular Csound or Max user.

Nonetheless, the advantages of the hybrid approach are profound. The hybrid programmer has the opportunity to prototype tools in the environment that presents the least work, and to move them to a more appropriate environment as they grow in complexity. Numerous performance optimizations become possible as each of the components of the hybrid platform have areas in which they are faster or slower. Reuse of code is made more practical - experienced programmers moving some of their work to GPPLs can take advantage of modern development tools such as version control systems, integrated development environments, and editors designed around programming. And finally, the complexity of algorithms one can use is essentially unlimited.

2.4 Conclusion

It is in this multi-language, hybrid space that Scheme for Max sits. S4M provides a Max object that embeds an interpreter for the s7 Scheme language, a general purpose language in the Lisp family (CTN: Schottstaed n.d.). With S4M, one gets a general purpose language in a visual patcher, and with objects such as the csound~ object, can interact closely with a textual DSL as well.

Given the myriad options existing already in the hybrid space, we might well ask why a new tool is justified, why specifically it ought to use an uncommon language, and why it should be embedded in Max specifically rather than some other platform or langauge. To answer these questions, first we will look at my personal motivations, and following that, at why I chose Max and s7 Scheme to fulfill them.

CHAPTER

THREE

PROJECT MOTIVATION AND GOALS

3.1 Overview

Scheme for Max is the result of many years of exploring computer music platforms for composing, producing, and performing music. The project requirements were born from my experiences, both positive and negative, with a wide variety of tools, both commercial and non-commercial. Broadly, the requirements I set myself for the project are that it should:

- Focus on programming musical events and event-oriented tools
- Support multiple contexts, including linear composition, real-time interaction, and live performance
- Support advanced functional and object-oriented programming techniques
- Be linguistically optimized for the target use cases
- Be usable in conjunction with modern, commercial tools
- Support composing music that is impractical on commercial tools
- Enable iterative development during musical playback

These goals will act as our reference when discussing the sucess of design and implementation decisions.

3.2 Focus on programming musical events and event-oriented tools

While numerous options for programming computer music exist, most domain specific computer music languages are designed principally around rendering audio, essentially acting as higher level langauges for digital signal processing (DSP). Examples of these include SuperCollider, Chuck, Faust, and to a lesser degree, Csound. My goal for the project is instead to focus the design decisions first around the programming of musical *events*, unencumbered by the need to also support DSP programming. The intent is that the tool will be used in conjunction with other software that handles the lower level DSP activity, whether this is by delegating to other programming languages (e.g. Max, Csound, Chuck, etc.) or by sending messages to external audio generators through MIDI or OSC messages. The decision not to have the tool itself handle DSP significantly reduces the amount of ongoing computation, which opens linguistic possibilities. The most notable being that use of higher-level, interpreted, and garbage-collected languages becomes feasible, even in a realtime context. I believe that reducing the scope in this way enables an approach that is more productive for the specific act of *composing*, in that the features of high-level languages afforded by this decision are, in my opinion, tremendously helpful when working programmatically with musical abstractions.

3.3 Support multiple contexts, including linear composition, realtime interaction, and live performance

As previously discussed, on modern computers it is now possible to run complex computer music processes in realtime, where acceptable realtime performance means there are not any audible audio issues from missed rendering deadlines, and the system can run with a latency low enough for the user to interract with the music with an immediacy appropriate to playing instruments (i.e., between 5 and 20 ms). Meeting this goal thus implies that the project will enable one to iteract with a composition while it plays, and that an instrumentalist ought to be able to perform with objects created in the tool with a latency that allows accurate performances. However, there also exist types of music where a predetermined score of great complexity is the most appropriate tool, and where this complexity may exceed the capabilities of human players.

My goal is that the project should satisfy all of these criteria. It should be usable for creating entirely pre-scored pieces, such as the piano etudes of Conlon Nancarrow, where the score is so dense as to be unperformable by human players. It should also be usable in the studio to compose music that is developed through realtime interactions with algorithmic processes and/or sequencers. And finally, it should be usable on stage, for example to create performances in which a human being plays physical instruments or manipulates devices that interact with the program.

3.4 Support advanced functional and object-oriented programming techniques

The project should support advanced high-level programming idioms, many of which are now in the broader programming mainstream, having been adopted into mainstream languages such as JavaScript, Java, Python, and Ruby. This could potentially include support for first-class functions, lexical closures, message-passing object-orientation, tail recursion, higher-order types, continuations, and so on. The support for these programming idioms ought to make it possible for composer-programmers to express their intent more succinctly, with the code better representative of musical abstractions, and should also make the tool more attractive to advanced programmers who might otherwise feel they need to use a general purpose programming language.

3.5 Be linguistically optimized for the target use cases

Support for higher level functional and object-oriented programming idioms can be done in a variety of general programming languages, with the differences between these languages having ramifications on the development process. All language design involves tradeoffs - what is most convenient for a small team of expert users early in the process of development can be a hindrance for a large team of mixed expertise working on a very large code base. Design of the project should take this into account.

I think we can safely assume composers working on pieces are normally working solo or in very small teams. The work of a composer-programmer likely consists of building many smaller projects (smaller relative to the size of a large commercial code base that is), all of which may build on some common set of tools and processes reused across pieces. The linguistic design of the project should take this into account and be optimized for this scenario, favouring whatever is most efficient for the process of composing and interacting with the system while the program runs, and favouring the linguistic tradeoffs appropriate for the solo developer who is likely to be able to hold the entire program in one brain.

3.6 Be usable in conjunction with modern, commercial tools

A problem with many of the existing computer music DSLs is that they were designed with the expectation that the user would be using only (or principally) the lanugage in question - that in effect, the DSL would always get to "be the boss". For example, Common Music enables composing in a high-level language (Scheme), but to be used in real-time, this Scheme code must be run from the Grace host application, where it uses the Grace scheduler for controlling event times (CTN: Taube 2009, 451). Thus combining music coming from Common Music with musical elements coming from a commercial sequencing program such as Ableton Live is not terribly practical if one wants tight synchronization and the "clock-boss" to be Ableton Live. As commercial music software becomes more and more sophisticated, especially in the areas of sound design and emulation of analog hardware, working in a platform or multi-language scenario that doesn't support the use of commercial plugins, or at least synchronizing closely with commercial software, becomes less and less attractive to me. Thus a goal of the project is to ensure that, whatever its design, it lends itself well to composing and producing in scenarios where some musical elements may originate from or stream to commercial tools such as Ableton Live and modern VST plugins. The user of the project should not be faced with a binary choice between using the power of the project or having the convenience and audio quality of modern commercial tools available to them.

3.7 Support composing music that is impractical on commercial tools

While being able to work with commercial tools is a goal, this cannot be at the expense of supporting compositions that are impractical to render purely on commercial platforms. Naturally, commercial music tools are designed around the needs of the majority of users. The visual and interface design of sequencers and workstations such as Ableton Live, Logic, and Reaper make assumptions that do not stand up to many 20th century or new music practices. These assumptions could include: that there will be only one meter at a time, that meter will not change too frequently, that the time scale of composition used across voices is similar, that the number of voices is not in the thousands, that the piece macro-structure is the same across voices, that all voices share the same tempo, and so on. While certainly one can find ways around these assumptions in commercial tools, the work involved can be laborious and discouraging. However, these assumptions do not need to be made for a tool using a high-level textual language.

3.8 Enable iterative development during musical playback

Finally, a goal of the project is to ensure that all of the goals listed so far can be achieved in a way that allows *interactive development* during audio playback. As with a hardware or commercial step sequencer, I should be able to update a looped sequence during playback, hearing the change on the next iteration of the loop, without having to stop and restart playback. This workflow is productive compositionally, and provides the ability to use the ear as the judgement source as ideas are explored. Languages in the Lisp family (and some others) allow this kind of workflow during software development, an idiom know as interactive programming, or REPL-driven development (REPL being a reference to the Read Evaluate Print Loop). In this style of development, code is incrementally updated while the program is running, allowing an exploratory style of development that is ideal during early prototyping and during the composition process (CTN: Taube 2004, 8). For the domain of algorithmic music, interactive development provides the same kind of immediacy one gets with sequencers that allow updating data during playback. Indeed, there exists an entire musical community dedicated to this kind of music programming, known as "live coding", in which the performer takes the stage with minimal advanced material prepared and composes in the programming language in view of the audience, often with the code projected on screen (CTN: Roberts and Wakefield 2018, 293-294). While *performing* live coding is not a personal goal of mine, the ability to live code while *composing* is. The project should support this style of working.

3.9 Conclusion

By explicitly listing the motivational goals and requirements of the project, I can better describe why I made the design choices I made (Max, s7 Scheme), and subsequently evaluate whether the project as a whole is successful.

CHAPTER

FOUR

HIGH-LEVEL DESIGN

In this chapter I will discuss the design of Scheme for Max on a high level and the rationale for the various design decisions, in light of the previously discussed goals.

4.1 Why a Max extension?

As discussed, Scheme for Max is a tool for multi-language computer music programming. This begs the first design question: "Why choose Max specifically as the host platform?".

One driver for choosing Max is the success of Max itself. Having been first created in the 1980's, Max is now one of the most popular platforms world-wide for making computer music, with thousands of Max objects available between those provided by Cycling 74 and the broader Max community (CTN: maxobjects.com 2023). This extensive community and breadth of available objects enables a wide variety of ways of working with both musical abstractions and digital audio, and even video. This includes support for interacting with external hardware through various synchronization and message protocols and hosting commercial software synthesizers and effect plugins in Max itself through the vst~ object.

In addition to being a popular and powerful platform on its own, Max has been available since 2009 as "Max for Live", an embedded runtime within the widely adopted commercial audio workstation and sequencing platform, "Ableton Live" (CTN: Ableton 2009). Max for Live has been so successful as an addition to Live that it led to Ableton aquiring Cycling 74 in 2017 (CTN: Ableton 2017). When run in Max for Live, Max patches are able to processing both audio and MIDI data, can also interact with the host through an application programming interface, the Live API. The Live API provides Max patches the ability to control and query the Live engine, read and write to Live's own sequence and audio data, interact with the mixer and effects, and interact with the global transport. Max for Live is included automatically in the top-tier Ableton offering ("Live Suite") as well as via an add-on to Live. While exact numbers are not published by Ableton, it seems highly likely that, between the standalone Max version and Max for Live, Max has become the most widely deployed advanced computer music programming platform worldwide.

In addition to the popularity of the platform and its integrations with a full-featured commercial sequencing tool, there are implementation related attractions to Max as well. Max supports two kinds of data passed between objects, audio samples and event messages, which in turn run in several different threads of execution. In Max, there are actually two threads handling event messages, the "main thread" and the "scheduler thread". The first is also referred to as the UI thread, or low-priority thread, and the second as the high-priority thread. MIDI data, events from timers and metronomes, and events from audio signals that have been turned into event messages (with various translation objects) all use the high-priority scheduler thread. Events from the user interacting with the GUI run in the low-priority thread, which is also used for redrawing any UI widgets. A Max external can run in any or all of these contexts, and various objects and functions exist to pass events from one to the other (CTN: Cycling 74 2019).

Scheme for Max operates only in the two event oriented threads, receiving and producing only Max messages - that is, it does not render blocks of samples in the audio thread. It can optionally be run in *either* the low or high priority thread, but each instantiation of the s4m object is limited to one of these (chosen at instantiation time). The threading designs of Max and Scheme for Max make it possible for us to run Scheme for Max code only occasionally (i.e, on

receipt of a message rather than on every block of samples), and also to ensure that it runs at a high priority and is not interrupted by low priority activity.

As a result, the choice of creating the project as an extension for Max supports several of the stated goals. There is a clear distinction between event time and audio time, supporting the goal of focusing on the musical event (e.g. "note") abstraction level. By running only at the event time scale, we are afforded the option to use a high-level, garbage-collected language - whereas runing a garbage-collected language in the audio rendering loop would seriously limit the amount of computation possible in real time and likely require us to run with a high degree of latency.

Additionally, using Max supports also the goal of being able to use the project in conjunction with modern commercial music tools. Max runs seamlessly inside Ableton Live, and both Max and Live can host commercial VST plugins. When run in Ableton Live, Max uses Live's transport, and when run as a VST host, Max's transport is visible to the VST plugins. It is thus possible to create music that mixes algorithmic content generated in Scheme with content that is sequenced, rendered, or recorded with commercial tools.

Finally, using Max supports the goal of being usable for real-time interaction and live performance. The Max clocking facilities are highly accurate, with jitter being typically in the 0.5-1ms range when using a typical signal vector size of 32 or 64 samples (CTN: Lyon 2006, 67). (Signal vector size is user configurable in standalone Max, and is locked to 64 samples in Ableton Live.) Max timers are also implemented such that this degree of jitter does not accumulate over time, something I have verified in extensives tests during development. This means we have a way to work with realtime events at levels of latency and temporal accuracy that are appropriate for live performances with highly trained musicians.

Taken together, these three points makes Max a very attractive host platform for the project. We can create music that is implemented in various mixes of Scheme, standard Max programming, and sequencing from Ableton Live. We can use modern commercial effect and synthesis plugins, taking advantage of the dramatic advances in software synthesis in recent years. And timing is reliable and accurate enough that we can use the tool on stage, or in the studio for commercial production of music where high timing accuracy is desired (e.g. electronic dance music). We can even, through Ableton Live, use the tool during the mixing and mastering processes, as all of this can be done in Live, with Scheme for Max used to orchestrate and automate Live devices and VST plugins.

However, the advantages just discussed could be applied to any general purpose programming language hosted in a Max external. Which begs the second question: "Why use a Lisp dialect rather than something a more popular interpreted language as Python, Ruby, JavaScript, or Lua?" Or more pointedly, why bother at all, when Max provides already an object that embeds an interpreter for JavaScript in the form of the **js** object?

4.2 Why not just use JavaScript?

I will discuss in some depth the linguistic reasons for choosing a Lisp language, but first I will outline the reasons I could not simply use the built-in js object to satisfy the project goals.

At first glance, the js objects seems like a comprehensive solution. It runs in Max, it can send and receive Max messages, it has access to Max global data structures such as tables and buffers, and it has a scheduler facility in the Task object (Taylor 2020). Linguistically, it's a high-level language with various modern features such as automatic memory management, objects, lexical scoping, and functional programming techiques such as closures, and it is now one of the most popular languages in the world (Sun 2017).

Unfortunately, the js object in Max has a serious implementation issue - in current versions of Max it *only* executes in the lower-priority main thread. Any messages sent to the js object from the scheduler thread are implicity queued to the main thread and handled on its next pass (CTN: docs.cycling74.com n.d.). The upshot of this is that the short-term timing of events handled in the js object is not reliable - depending on other activity, execution of messages can be delayed, with this delay large enough to be audibly noticeable as errors. While JavaScript is usable for a great many tasks in Max, realtime programmatic note triggering is not one of them. This was simple for me to verify, as any heavy redrawing of the graphic layer (such as by resizing the window) will introduce significant delays to tasks scheduled through the Max JavaScript Task object.

In fact, I began my work combining textual programming with the Max environment by attempting to use the js object to build large-scale sequencing projects. Overcoming the timing limitations of the js object was one of the initial motivations for the Scheme for Max project. Fortunately, there is nothing in the Max SDK (the C and C++ software development kit used for building Max extensions) that requires one to use any particular thread, thus any high-level language with an interpreter that is available as C or C++ code can be used in the scheduler thread safely so long as messages coming from other threads are appropriately queued.

4.3 Why use a Lisp language?

Given that using the js object was not deemed satisfactory, the next design question becomes: which choose a Lisp language? For the purposes of this dicussion I will use "Lisp" when referring to traits shared across the Lisp family of languages (including Scheme, Common Lisp, Clojure, and Racket), and Scheme when referring to the particular choice used in Scheme for Max.

In the initial research stage of this project (dating back to 2019) I examined various possible high-level languages, and reviewed the use of many various general purpose languages in music. Non-Lisp candidates I evaluated included Python, Lua, Ruby, Erlang, Haskell, OCaml, and JavaScript (i.e. in a new implementation).

Overall, I came to the conclusion that the advantages of working in a Lisp for music outweigh the disadvantages of its relative unpopularity and its unfamiliar syntax (to most programmers today at least!). These advantages include suitability for representing music; suitability for the typical scenarios and needs of the composer-programmer; and suitability for implementing the project in Max specifically.

Compared to the other candidate languages mentioned, Lisps differ in several ways that are germane to this discussion. (To be clear, some of these traits are shared by some of the other candidates, but I would argue that none of the other candidates share all of these traits with Lisps.)

4.3.1 Symbolic computation and list processing

Lisp is unusual in its first-class support for programming with *symbols* and in its simple, minimal, and consistent syntax (CTN: Taube 2004, 8). Programming with symbols, also known as "symbolic computation" or "symbolic processing", means that programs can work directly with not only program *data* but with the *textual tokens* comprising the program itself. For example, as with any high-level language, we may have a variable named "foo", at which we have stored the value 99, allowing us to refer to the contents bound to that variable (99) by the name "foo". When the interpreter encounters the textual token "foo", perhaps in an expression such as "1 + foo", it will automatically *evaluate* this token, replacing it in an internally expanded form with the number 99. But in Lisp, we may also work with the textual token itself, referred to *the symbol foo* just as easily as we work with any other primitive type. We can pass it around, put it in lists, concatenate it to other symbols, and so on. When we want to refer to the symbol part of a variable (the text to which the value is bound), we use a facility of the language called *quoting*, by which we instruct the interpreter to skip evaluating the symbol as a variable (thus expanding to 99) and instead work with the textual token. We can quote by using the **quote** function, or by prepending a symbol with a single quote: **'foo**. This symbolic processing capability is particularly appropriate for music, as we shall see shortly.

In addition to this, Lisp syntax is *entirely* composed of s-expressions, which are parenthetical expressions containing lists of symbols and primitives. We will see why this matters shortly.

For example, below are several ways to return a list of symbols. We can see that all use one or more parenthetical expression as the basic unit of syntax.

```
;; 3 ways of creating a list containing the symbols foo, bar, and baz
;; use the list function
(list 'foo 'bar 'baz)
;; quote the printed representation with a single quote
```

(continues on next page)

(continued from previous page)

```
'(foo bar baz)
;; use the quote function on the printed representation
(quote (foo bar baz))
```

The value returned by the above expressesions is represented on the console by the text (foo bar baz). Note that this looks idential to the source-code for a Lisp function call, specifically it looks like code we would use to call the function **foo** with the arguments **bar** and **baz**. And indeed, if we were to take the lists returned in our example and pass this returned *symbolic* structure to the Lisp **eval** function, that is exactly what would happen - the interpreter would execute whatever function is bound to the symbol **foo**, passing it the arguments bar and baz.

Below is an example of doing just this at a Scheme interpreter. (The lines prefaced by **>** are the text responses from the interpreter that would be printed to a console in an interactive session.)

```
; create a list and save it to the variable my-program
(define my-program (list 'print 99))
> my-program
; now run it, which will print 99
(eval my-program)
> 99
; or all in one step
(eval (list 'print 99))
> 99
```

In the example above, we used quoting to create a list consisting of the symbol **'print** and the number 99, and then we used **eval** to *run this list as a program*. The impact of this is profound: Lisps allow us to easily and elegantly make programs that build lists of symbols and primitives, *and these lists we have built can themselves be executed as programs*.

Now to be clear, we can also build a program with a program in other high-level languages, including Python, Ruby, Lua, and JavaScript. However, in none of these languages is programming *on* the symbolic tokens of the language directly supported the way it is in Lisp. The result is that in these other language this kind of dynamic programming (also known as "meta-programming") is very involved and typically seen as something to be used only sparingly by expert programmers building reusable tools. In Lisp, on the other hand, manipulating lists of symbols, and later evaluating them as functions, is the very stuff of which the langauge is made.

Now, why does this matter for a programming language for music?

As in Lisp code, in music we use lists of symbols to represent functions, relationships, and events. For example, let us say I write a chord progression, such as **I vi ii V7**. We have a *list* of four items, each denoted by a symbol: **I**, **vi**, etc. Each of these symbols represents musical data for a given chord, but by themselves, they don't represent *music* - they need a key *to which the function represented by the chord symbol can be applied*. Thinking computationally, **V7** must be a *function* - it is a description of something we get when we apply a particular algorithm (the intervals within the chord along with the scale-step for the root) to a parameter (the tonic key).

In a Lisp language, this can be represented in code that is visually compatible (almost identical even) to what we would use in musical analysis. (chords->notes 'C '(I vi ii V7)) is a legitimate line of Lisp syntax that could be implemented to be a function that renders a chord progression into a list of notes, given a tonic of C. It could even return something symbolic that looks very familiar to a musician, and *on which more of the program can work*. A potential return value could be represented by the interactive Lisp interpreter as a nested list containing sublists of symbols: '((C E G) (A C E) (D F A) (G B D F))

Further, because this form of symbolic computation is so central to the language - one of the classic texts is even subtitled "A Gentle Introduction to Symbolic Computation" - Lisps include numerous functions for manipulating and transforming lists (CTN: Touretzky 1984). For example, we might transpose a list by applying a transposition function, which itself might be built by a function-building function called **make-transposer**, and we might apply this function

to a list of symbols. This sounds complicated, and indeed, expressing this in most languages is cumbersome, but in Scheme this is both readable and succint:

```
; apply a transposition function that transposes all elements in our chord progression by 2 steps
; the map function maps a function over a list, returning a new list
; (make-transposer 2) creates a function that transposes by 2 specifically
(map (make-transposer 2)
  '( (C E G) (A C E) (D F A) (G B D F)))

; expressed without first expanding our chord progression
(map (make-transposer 2)
  (chords->notes 'C '(I vi ii V7)))
```

This demonstrates thats Lisps are particularly well-suited to expressing musical data, relationships, and algorithms in computer code, and a result of this suitability, there is a rich history of Lisp use in musical programming. Examples of Lisp-based musical programming environments abound, both historical and current. In addition to those already mentioned (Common Music and Nyquist), others include Common Lisp Music, Common Music Notation, MIDI-Lisp, PatchWork, OpenMusic, cl-collider, and many more (CTN: CLiki n.d.).

Thus the choice of Scheme as the language for the project has several important advantages:

- Code representing musical data can be more succint, lowering the sheer amount of code the composer must contend with while working.
- Code working with musical constructs can look remarkably similar to the notation that composers are used to, making the code more readable, and thus more appropriate for use within a piece of music that may be composed of both data and code.
- Programmers have access to a rich historical body of prior work, with code that can be ported to Scheme for Max relatively easily.

4.3.2 Dynamic code loading and the REPL

Previously mentioned as interactive development, or REPL-driven development, Lisp programmers commonly work in an ongoing process of evaluating new code in the interpreter and examining the interpreter's output, *while the program runs*. At any point, the programmer can send new expressions to the Lisp interpreter, which evaluates the expressions, updates the state of the Lisp environment, and then prints the return value of evaluating the expressions. These expressions can define new functions, redefine functions already in use, change state data, or interactively inspect or alter the current environment. While this iteractive style of development is possible to some degrees in other high level languages (such as Python and Ruby), it has been available to a deeper degree in Lisp going back as far as the the 1970's! (CTN: Sandewell 1978, 35-39)

For example, the composer-programmer might separate work into files that contain score data and files that contain functions for altering or creating music, where the functions might be musical transformations of algorithms for generating new content given base score data. The files of functions can be incrementally edited and reloaded, thus updating algorithm definitions, without needing to restart the piece or reset the score data.

In Scheme for Max, the programmer can also trigger interpreter calls from text interface objects in Max, or even from an external text editor by sending blocks of code over the local network into Max. Max has a console window to show messages from the Max engine, and this is used by Scheme for Max for the Print stage of the REPL loop so that the results of dynamic evaluation can be read by the programmer.

I have personally found this capability to be enormously productive while working on algorithmically generated or augmented compositions - the ability to tinker with the algorithms without necessarily restarting a piece is a signficant time saver, and being able to interactively inspect data in the Max console while doing so is similarly helpful.

4.3.3 Macros and Domain Specific Languages

One of the hallmarks of Lisp is the Lisp macro. We have previous discussed the ease with which the Lisp programmer can programmatically create lists of symbols that are then evaluated as syntactic Lisp expressions; the Lisp macro is a linguistic formalization of this process. In use, a macros look to the programmer just like a regular function call, but by virtue of being defined as a macro, they are first called in a special evaluation pass known as the macro-expansion pass. This runs the code in the body of the macro over the *symbolic arguments* passed in to it, returning a programmatically created list structure (the macro-expansion) that is then evaluated. Essentially, macros are code blocks that execute twice - first to build the code, then to evaluate it - though technically they can be nested to repeat the expansion step an arbitrary number of times (CTN: Touretsky 1984, 405-417).

Macros enable programmers to create their *own* domain specific languages - miniature languages within a language that are closer in syntax and sematics to the problem domain than to the host languauge. This makes it possible for code that uses the macros (the "domain code") to be visually aligned with the problem domain, making them easier to read and faster to type. For example, a macro I use for scheduling events in a score looks like the below:

```
(score
  :1:1       (phrase-a :dur 2b :repeat 4)
  :+8        (phrase-b :dur 8b :repeat 4)
  :9:1:120   (..etc))
```

The time argument, :1:1, :+8, and :9:1:120 are converted by the macro layer into musically meaningful time representations, allowing the visual representation of the score code to be more easily read by the composer. The flexibility of macros allow me to use textual representations that are convenient for me as the composer.

But to clarify, this is *not* a separate score language with limited functionality, as is found in Csound. This *is Scheme code* - it can include *any* Scheme functions and even be built by Scheme functions. Thus the use of a language with macro facilities enables the composer to work with different kinds of code - function defining code and score code - in one language, without giving up the expressive power of high-level language facilities. This use of a general programming language that can function additionally *as a readable score language* provides tremendous flexibility to the programmer, breaking the dichotomy between score data and running program (CTN: Dannenberg 1997, 50-60).

4.3.4 Max and Lisp syntax compability

Finally, there is the fortunate coincidence of the Max message syntax being almost perfectly compatible with Lisp syntax. This happy accident (we can assume!) means that a composer-programmer can create and run Scheme code *in Max messages*, and use Max message-building functions to do so. While this compatibility was not something I was expecting when originally embarking on the design of Scheme for Max, it has had a profound effect on the ease with which one can build Max patches that interact with Scheme for Max programs.

A Max message consists of Max *atoms*, which are space-separated tokens that may be integers, floating point numbers, or alpha-numeric symbols. It may also consist of several special characters: the dollar sign, the comma, and the semi-colon. The dollar sign is used as a template interpolation symbol: messages with dollar signs in their text body will output template explansions to downstream objects, injecting arguments they receive in their inlets. A leading semi-colon in a Max message indicates the message is a special message sent to the Max engine itself. Finally, the comma is used to indicate that the message is actually two message, with the two comma-separated halves being sent sequentially.

Notably, the parenthesis, used in Lisp to delimit Lisp expressions, and the colon, used to indicate that a symbol should be a keyword (a special kind of symbol), have no special significance in Max messages. Conversely, the dollar sign has no significance in Lisp, and the semi-colon (used for comment characters) and the comma (used for back-quote escaping) are easily avoided.

The result of this is that rather than require the programmer to create special handlers in their code to respond to Max messages, as one must do when using the js object, the s4m object is able to simply evaluate incoming messages *as if they were Scheme code*, saving the programmer the need to write callback functions for every type of incoming message. This facility is covered in more detail in the Features and Usage chapter, with an accompanying figure.

Having built some complex programs myself in JavaScript in Max prior to building Scheme for Max, I have found this to be a significant advantage of Scheme for Max over the js object.

4.4 Of the possible Lisp languages, why use s7 Scheme?

When beginning the project, after determining that a Lisp-family language was appopriate, I evaluated a number of Scheme and Lisp implementations as candidates. I will discuss now why the s7 implementation in particular was chosen. (Note for the curious: the author has informed me that s7 is intended to be spelled lowercase as it is named after a Yamaha motorcycle!)

4.4.1 Use in Computer Music

s7 was created by, and is maintained by, Bill Schottstaedt, a professor emeritus of the Stanford music centre (CCRMA), and the author of Common Lisp Music and the Snd editor. s7 is used in in Snd editor (essentially an Emacs-like audio editing tool), and in Common Music 3, an algorithmic composition platform created by Henrik Taube (Schottstaedt n.d.). This has meant that there is a significant body of code from Common Music that can be used with very minimal adjustment in Scheme for Max. Indeed, if I were to describe S4M in one sentence, it would be that it is a cross between Common Music and the Max js object.

4.4.2 Linguistic Features

Not suprising, given the author's involvement with Common Lisp (CL) music systems, s7 is, by Scheme standards, highly influenced by Common Lisp. It includes Common Lisp *keywords*, which are symbols that begin with and always evaluate to themselves. s7 also uses Common Lisp style macros (a.k.a. "defmacro" macros), rather than the syntax-case or syntax-rules macros in many other Scheme implementations. To support CL macros safely (without inadvertent variable capture), s7 includes support for first-class environments (lexical environments that can be used as values for variables), and the "gensym" function, which is used to create guaranteed-unique symbols for use in a macroexpansion (Schottstaedt n.d.). Interestingly, and perhaps fortunately for the purpose of adoption, these are features also shared with Clojure, a modern Lisp variant with much in common with Scheme, and with wide use in business and web application circles (Miller et al. 2018).

We can assume these features were chosen by Bill as appropriate for his use case - the solo composer-programmer - and indeed in my personal experience they have been helpful for working on projects in S4M. For example, the ability to use keywords allows us to have symbols in Max messages that will be preserved as symbols when the message is evaluated by the s4m interpreter, and these are easily differentiated visually in Max messages.

4.4.3 Ease of embedding

Of the Lisp dialects, Scheme in particular has a further pragmatic advantage. Due to its minimal nature, it is eminently appropriate for embedding in another language, and there thus exists a wide variety of embeddable Scheme intepreters. A functional Scheme interpreter can be created in a very small amount of code - there is even an implementation named SIOD, for "Scheme In One Defun" (but also referred to as "Scheme in One Day"). SIOD was a project by computer science professor George Carrette, started in 1988, intended to make the smallest possible Scheme interpreter that could be embedded in a C or C++ program (CTN: Carrette 2007).

The s7 project in particular is a Scheme distribution intended expressly for embedding in C host programs, and designed to make that use case as simple as possible. The core s7 interpreter is distributed as only two files, s7.h and s7.c, that can simply be included in a source tree. The foreign function interface (FFI) is very straightforward, making adding Scheme functions to S4M simple. And, importantly, s7 is fully thread-safe and re-entrant - meaning that there is no issue having multiple, isolated s7 interpreters running in the same application, a situation common in a Max patch where many s4m object may coexist, but a feature not common across all candidate implementations (CTN: Schottstaedt n.d.).

4.4.4 License

Finally, s7 uses the BSD license, a permissive free software license. The BSD license imposes no redistribution restrictions the way the GPL family of licenses do, thus user-developers wishing to use s7 in a commercial project are free to do so with no obligations (CTN: Schottstaedt n.d.).

This is a point in s7's favour as many Ableton Live device developers sell devices, and many Max developers sell standalone Max applications, thus I would also like to allow use of S4M in these contexts.

CHAPTER

FIVE

FEATURES AND USAGE

In this chapter I will discuss some of the principal features of s4m from the perspective of a composer-programmer using s4m to create musical works. In the interest of space, I will not cover all of S4M's functionality, however the interested reader can consult the online documentation in which all the capabilities are covered. (https://iainctduncan.github.io/scheme-for-max-docs/)

This chapter assumes some familiarity with the Max platform, though readers unfamiliar with Max should be able to follow along. Where I refer to a message sent to the s4m object, I am referring to a Max message, such as would occur when a message object is connected to an inlet of the s4m object and clicked or "banged".

5.1 Installation

Scheme for Max is released as a Max Package which contains: the s4m and s4m.grid Max externals; a collection of Scheme source files; a Max help patch demonstrating features and use; and some example patches and Max for Live devices using S4M. In order to use S4M, the Max user must download the package file and uncompress it in their Max "Packages" directory, after which it will be possible to create an **s4m** object in a Max patch and to open the s4m help patch for assistance.

5.2 Object Initialization

5.2.1 Bootstrap files

When the s4m object is created in a Max patch, it will initialize itself by loading the bootstrap file, **s4m.scm**. This file contains Scheme code on which the documented s4m functionality depends, and also loads several other Scheme dependencies. This boostrap file is available for inspection and alteration by the user, and it is expected that advanced users will alter their bootstrap file, allowing them to automatically load additional files that they would like to have available automatically.

By default, the bootstrap file additionally loads the file **s74.scm**, which contains Scheme definitions that are not specific to Max. s74 is intended to be an extension to s7 to provide convenience features that I assume most users will want available. It adds various higher-level functions taken from, or inspired by, less minimal Scheme implementations, such as Chez and Chicken, as well as from the related Lisp dialects of Clojure, Racket, and Common Lisp.

The bootstrap file also loads several files that are packaged with s7 itself but are optional: **stuff.scm**, **loop.scm**, and **utilities.scm**. These files come as optional extensions int s7 distribution itself, and define several macros borrowed from Common Lisp and used in Common Music, such as **loop**, **dolist**, and **dotimes**.

Once the s4m object has loaded s4m.scm and its subsequent dependencies, it is ready to be used.

5.2.2 Source Files

The user can load source files into the s4m object in several ways. The primary way is to provide a file name as the first argument in the s4m object box in Max, similar to how this done in many other Max objects that load files, such as the js and buffer objects. S4M will search the Max file paths (user configured paths for source code search) to find the named file, and will load it if found. This file is then considered the main file for the s4m object instance. Double clicking the s4m box will print the full path to the main file in the console as a convenience to users.

Sending the s4m object the **reset** message will re-initialize the object, recreating the s7 interpreter and reloading the bootstrap files and the main file. Sending s4m a **reload** message will reload the main file, *without* resetting the interpreter. The difference here is that if one has made definitions in the interpreter from outside the main file (how this happens will be covered shortly), these will not be erased on a reload, but will be on a reset.

Sending s4m a **source some-file.scm** message will load some-file.scm and set it as the main source file. This can be useful in cases, such as in a Max for Live device, where it may be convenient not to have to edit the Max patch to change the main file. This use case is not uncommon as the commercial licenses required to *use* Max for Live and to *edit* in Max for Live are different. For example, one might create a device where a text box can be updated and is interpolated into a **source** message, allowing a user of the device, who may not have the ability to edit devices, to change the main file.

The message **read some-file.scm** will load a file (again searching on the Max file path) without resetting the interpreter or changing the s4m main file. This is useful when working on a program or piece while it runs: a user can put state variables and score data in one file and algorithms in another, allowing them to reload the algorithms file after changing it, while leaving the state and score data alone.

In the event that the user has multiple similarly-named files on their Max search path, Max will load the first one it finds, and print a message to the Max console indicating that multiple source files were found and which one it loaded. (This is a feature of Max, and has nothing to do with s4m specifically - it comes with the use of the SDK functions to load files from the search path.)

5.2.3 Inlets and Outlets

By default, the s4m object will be created with 1 inlet and 1 outlet. The **@ins** and **@outs** attribute arguments can be used at instantiation time to create additional inlets and outlets, to a maxium of 32 outlets. While these are implemented as Max *attributes*, they cannot be changed in the Max object inspector as their number must be set before object initialization. They can only be set as @ arguments in the object box.

5.3 Input

5.3.1 Inlet 0 Scheme Expressions

Input to the s4m object works differently depending on whether one uses the main left-most inlet (a.k.a. inlet 0) or subsequent inlets. A common pattern in Max objects is for objects to accept "meta" messages in inlet 0 - messages that configure the object, but are not calls to execute the objects main functionality. S4M follows this pattern, and supports a number of meta messages, such as the previously mentioned **reset** and **source** messages. While these message have an effect on the Scheme interpreter, they are handled by the s4m object's C functions, rather than being passed to the Scheme interpreter for evaluation. I refer to messages that are handled this way as "reserved messages", as they are not meant to be used as function names in Scheme (technically, there is nothing preventing this, but it is not recommended as tracing which component is handling the message will not be obvious).

Any messages to inlet 0 that are not reserved messages are evaluated as expressions by the Scheme interpreter. As previously discussed, S4M adds implicit enclosing parentheses around any non-reserved messages that do not already start and end with parentheses, and then passes the message to the s7 interpreter for evaluation. This convenience feature allows users to make calls from Max messages to Scheme more visually readable - for example, a message

of **my-fun 99** will be treated as **(my-fun 99)**. This also make it possible for users to programmatically build Scheme expressions with Max objects, such as by using the **prepend** object to insert a symbol at the beginning of some list. The return value from evaluation is normally printed to the Max console, though S4M provides various facilities for controlling how much is printed to the console (see the documentation).

This facility makes it very straightforward for a user to add input mechanisms to their programs. For example, if they want a number box to update a Scheme variable, they can use Max's dollar sign interpolation facility in a message such as **set! my-var $1**, connecting a number box or dial to this message, and connecting the message box to inlet 0 of the s4m object. Typing the number 99 into the number box will now result in Scheme calls to set the my-var variable to 99, as the s4m object will receive the message **set! my-var 99** and will treat this as **(set! my-var 99)**, evaluating accordingly. This capability significantly reduces the amount of code the user must write to make interactive patches when compared to the Max js object, as the js object requires explicit handler methods to be made for any input (Cycling 74 n.d.).

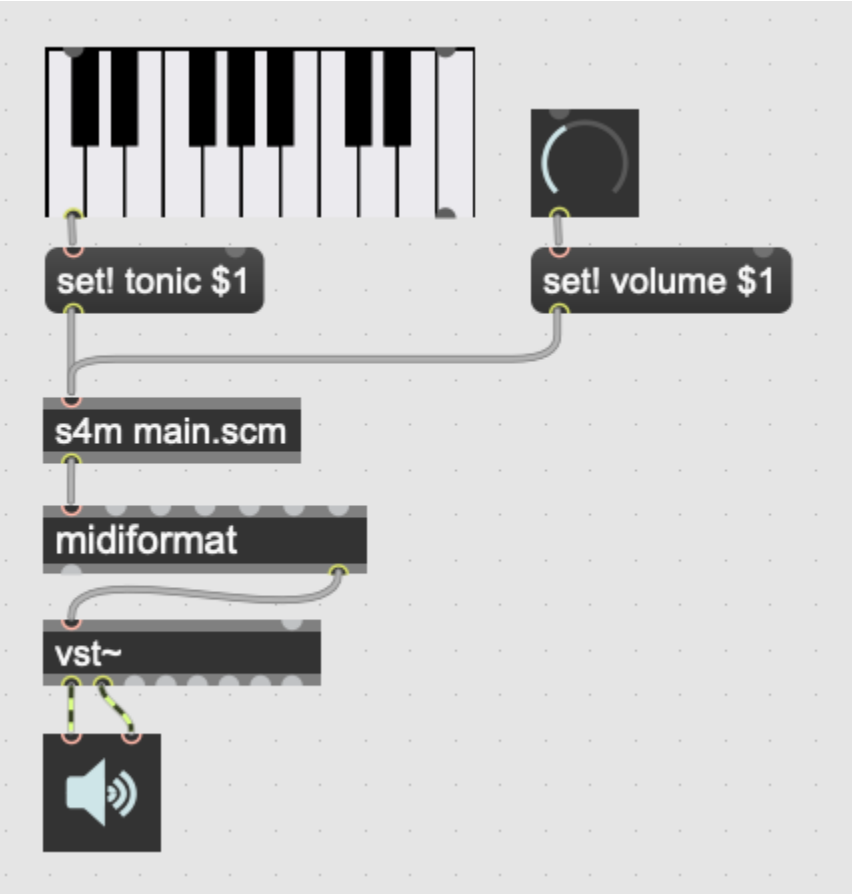

Fig. 1: Figure 2: Setting Scheme variables using Max message interpolation.

A result of this input facility is that when one uses a symbol in a Max message sent to inlet 0, the interpreter will take symbol to be a variable name in the running Scheme programs top-level environment. Should the user wish to pass in a *symbol* (i.e., not refer to a variable), they can use the standard Scheme leading single quotation mark to quote the symbol. They can also use an s7 *keyword* (a symbol beginning with a colon, that always evaluates to itself), in which case evaluation does not change the fact that the keyword is a symbol. Rather fortunately for us, Max does not assign any special meaning to either single quotation marks or colons, thus this presents no issue from Max messages. One can, for example, even name various Max objects such as buffers with colon-prefixed names.

For the majority of use cases, this is the easiest way to send input to the Scheme interpreter. When one wants to do something with an argument from Max, one can use message interpolation or the **prepend** object to turn the incoming

argument into a Scheme expression, and have the interpreter evaluate it.

There do exist, however, several convenience functions in case users want to handle input with even less boilerplate in their Max patch, at the cost of more boilerplate on the Scheme side. The **f-int**, **f-float**, **f-bang**, and **f-list** handlers are automatically invoked when the s4m object receives an integer, float, bang, or list respectively in inlet 0. If the user has defined such a function, it will be invoked, if they have not, the default handlers will be invoked, which simply print an error message. (These are named **f-{{type}}** simply to avoid the inconsistency that would result had we used **int**, **float**, and **bang**, as **list** is a built in Scheme function.)

5.3.2 Inlet 1+

There are times when it is not desirable that the incoming symbols in Max messages be taken as Scheme variable names. An example of this is dealing with incoming Open Sound Control (OSC) messages, where one may not have full control over the text formatting of the incoming message, and thus inserting single quotation marks to indicate symbols is not possible. For this kind of situation, messages to inlets over 0 are not automatically evaluated as Scheme code. This means that in order to accept input in inlets over 0, one must create a handler function and register it with Scheme for Max using the **listen** function. The call to **listen** takes arguments for the inlet, type of incoming message, and the handler function, where the type of incoming message can be one of: integer, float, symbol, or list. The handler function must be a single-arity function as it always receives its arguments as a single bundled list. This allows handlers to be generic and also allows the same handler to be registered for multiple types of message. An incoming Max list message made of Max symbols will be treated as a list of incoming quoted symbols. (Experienced Lisp programmers can think of these as being automatically quoted arguments.) It is up the handler to unpack the arguments from the list passed in.

Below is an example of defining a listener for a message consisting of an integer, and a second for a list.

```
;; handler message, all arguments are bundled into the args variable
(define (my-int-handler args)
  (let ((int-arg (args 0)))
    (post "s4m got the int:" int-arg)))

;; register it to listen for integers on inlet 1
(listen 1 :int my-int-handler)

(define (my-list-func args)
  (let ((list-length (length args))
        (first-arg   (args 0)))
    (post "s4m received a" list-length "item list, first item:" first-arg)))

;; register it to listen for lists on inlet 1
(listen 1 :list my-list-handler)
```

5.4 Output

The s4m object can output a Max message from any of its outlets using the **out** function. This is accomplished by passing the **out** function an outlet number and either a single value or a Scheme list of output values. Output values must be either integers, floats, symbols, or strings. Other value types (such as hash-tables or nested lists) will produce an error. Code to output various messages from outlet 0 is shown below.

```
;; output number 99
(out 0 99)
;; output a max list of ints
```

(continues on next page)

(continued from previous page)

```
(out 0 (list 1 2 3))
(out 0 '(1 2 3))
;; output a bang
(out 0 'bang)
;; output the value of my-var
(out 0 my-var)
;; output the max symbol "set"
(out 0 'set)
;; output the max message "set 99"
(out 0 (list 'set 99))
```

(Note that in Max, the special message type “bang”, such as one gets by clicking on a bang object, is synonymous with a message of a single symbol atom consisting of the symbol “bang”.)

5.5 Sending Messages

In addition to outputing messages via Max patch cables through the s4m object’s outlets, the s4m object can also send messages directly to Max objects that have been given a Max **scripting name**. On instantiation, and additionally on receipt of a **scan** message, s4m objects iterate over all objects in the same patcher and recursively through any descendent patchers. On finding any object with a scripting name, a reference to the object is placed in a registry in the s4m object, implemented as a Scheme hash-table with scripting names as keys and object references as values. The **send** function can then be used to directly send messages to these objects by using a symbol argument similarly named. Attempting to send to an unrecognized object will produce an error.

This uses the message sending functionality in the Max SDK, and is functionally no different from sending a message to a destination object via a patch cable. As with regular patch-cable messages, execution will pass to the receiving object and will not return to the caller until all subsequent message handling has finished. A variant of send exists, **send***, which flattens all arguments to allow conveniently sending list messages.

Code to send messages to a named destination is shown below:

```
;; update the contents of a number box that has scripting name "num-target"
;; by sending it a numeric message
;; we quote num-target below as we want the symbol num-target, not the
;; value of a variable named num-target.
(send 'num-target 99)

;; send a message box a message to update to the contents to "foobar 1 2 3"
(send 'msg-target 'set 'foobar 1 2 3)

;; or if we had the list ('foobar 1 2 3) in a variable named "msg":
(send* 'msg-target 'set msg)
```

This facility allows one to orchestrate complex activity in a Max patch without having predetermined connection paths. The results of messages so sent (as with patch-cable messages) are determined entirely by the semantics of the receiver.

5.6 Buffers & Tables

Max contains two types of globally-accessible objects for storing arrays of numerical data: the **buffer** and the **table**. Buffers are typically used to store floating-point sample data while tables are typically used to store integers, but either can be used for either. Both provide the programmer the ability to use indexed collections, and can have names, allowing objects that are not connected to a given buffer or table object to interact with them. The main use for buffers is as a container for audio data that can be played back in various ways, as well as manipulated programmatically by reading from and writing to them. An interesting feature of buffers is that the abstraction of the buffer of samples can be accessed by multiple Max objects by referring to the buffer by name, the name being provided as an argument to the **buffer** object that instantiates the buffer, allowing, for example, many objects to access the same audio sample.

Scheme for Max provides a collection of functions for reading and writing to and from buffers and tables, as well as convenience functions for getting the length of table or buffer and verifying if there exists a particular named buffer or table (**buffer?**, and **buffer-samples**, **table?**, **table-length**).

The simplest way of using these is to read or write a single data point using **buffer-ref** and **buffer-set!**. However, in the case of buffers, at the C level, Max locks the buffer before a read or write operation to ensure thread-safety in case other objects (that may be running in other threads) attempt to access the same buffer. Similarly, Max provides an ability to **notify** on a buffer update, so that objects sharing the buffer (such as visual display objects) can update their displays accordingly. Consequently, interacting with a collection of samples from the same buffer with a Scheme loop that makes repeat calls to buffer-ref or buffer-set! is slower than necessary, as locking, unlocking, and notifying will happen on every loop interation. For these scenarios, s4m functions exist to copy blocks of samples between Scheme vectors (Scheme's basic array type) and buffers, in which optional starting index points and sample counts are provided as arguments. At the C level, these lock, unlock, and notify only once, running direct low-level memory copies for all samples in between locking and unlocking.

```
;; example buffer operations
;; return true if buffer-1 is a buffer
(buffer? 'buffer-1)

;; get number of samples in buffer
(buffer-size 'buffer-1)

;; read value at index 2
(buffer-ref 'buffer-1 2)

;; write 0.5 to index 3
(buffer-set! 'buffer-1 3 99)

; make a vector
(define my-vector (vector 0.125 0.25 0.375 0.5))

;copy vector into buffer in one operation
(buffer-set-from-vector! 'buffer-1 0 my-vector)
```

While buffers (and to a lesser degree, tables) are implemented around the primary use case of storing sample data, they can in fact be used for storing numerical data in arrays for any purpose. The s4m facilities thus provide a complement to the Max functions, enabling iterative array manipulation with more convenient looping constructs than are built in to Max.

5.7 Dictionaries

Another higher-order data abstration provided by Max is the **dictionary**, a key-value store in which one can store a wide variety of Max data types as values, and use integers, floats, symbols, or strings as keys. Max provides a rich API for working with dictionaries, including the ability to refer to them by name across many objects, serialize them to JSON, update them from JSON files, and even send references to them between objects. There are a number of Max objects that have the ability to dump their contents to dictionaries, and various display handlers.

The Scheme equivalent of a dictionary is the **hash-table**, a key-value store that can hold any valid Scheme object, either as a key or value. S4M provides functions to interact with Max dictionaries and to convert between Max dictionaries and Scheme hash-tables. Notably, these are recursively implemented: converting a Max dictionary to a Scheme hash-table will convert all values in the dictionary, including nested dictionaries, regardless of the depth of nesting. Interesting, Max supports numerically indexed arrays of heterogenous type as values in dictionaries, even though there is no convenient way of directly working with arrays of heterogenous types in the visual patcher (one can though in JavaScript). Thus, using a dictionary as a container is one way to have simple arrays in regular in Max programming. If these are encountered during the conversion from a Max dictionary to a Scheme hash-table (or vice versa), S4M converts the nested arrays to Scheme vectors, where these vectors may contain a mix of types, including further nested dictionaries and arrays.

Similar to Common Lisp and Clojure, s7 Scheme (but not all Schemes) provides a **keyword** data-type, which is a symbol starting with a colon that always evaluates to itself. These are commonly used as keys in hash-tables. This is a convenient practice in Max, as one does not have to worry about quoting or unquoting as data passes through evaluation boundaries, such as when messages from Max go through inlet 0 of an s4m object.

S4M provides the functions **dict-ref**, **dict-set!**, **dict->hash-table**, **hash-table->dict**, and **dict-replace** for working with dictionaries. Of note is that these provide some convenience capabilities for dealing with nested dictionaries without having to nest calls to dict-ref and dict-set!, as shown below.

```
;; get a value from max dict named "test-dict", at key "a"
(dict-ref 'test-dict 'a)

;; get value at key "ba" in nested dict at key "b"
(dict-ref 'test-dict (list 'b 'ba) )

;; get the value at index 2 in the nested vector at key "c"
(dict-ref 'test-dict '(c 2) )

;; set a value in max dict named "test-dict", at key "z"
(dict-set! 'test-dict 'z 44)

;; set a value that is a hash-table, becomes a nested dict
(dict-set! 'test-dict 'y (hash-table :a 1 :b 2))

;; set value at key "bc" in nested dict at key "b"
(dict-set! 'test-dict (list 'b 'bc) 111)

;; set a value that is a hash-table, creating an intermediate hash-table automatically
(dict-replace! 'test-dict (list 'foo 'bar) 99)

;; create a hash-table from a named Max dictionary
(define my-hash (dict->hash-table 'my-max-dict-name))

;; update a Max dict from a hash-table
;; if the Max dictionary does not exist, it will be created
```

(continues on next page)

(continued from previous page)

```
(hash-table->dict (hash-table :a 1 :b 2) 'my-max-dict-name)
```

5.8 S4M Arrays

While in Max one has access to arrays of heterogenous type through dictionaries, and homogenously typed arrays of integers and floats through buffers and tables, there is no direct equivalent of the simple statically sized and homogenously typed C array (that is to say, buffers and tables are much more complex, coming with various forms of overhead). Scheme for Max fills this gap by providing its own internal implementation of arrays, the **s4m-array**, which provides an interface to static C arrays. These are created with the **make-array** function, providing a name, size, and type, where type may be **:int**, **:float**, **:char**, or **:string**. These arrays are stored by name in a global registry in the Scheme for Max code, allowing multiple s4m objects to use them to share data between instances. As the arrays are created in the s4m global registry, these persist beyond the life of a single s4m object, and are, at this point, only freed upon a restart of Max.

S4M provides functions for working with these point-by-point, (**array-ref** and **array-set!**) as well functions for copying blocks of data to and from Scheme vectors (**array->vector**, **array-set-from-vector!**).

```
;; create a 128-point array of integers, naming with a keyword
(make-array my-array :int 128)

;; copy a value from one array to another
(array-set! destination-array dest-index
  (array-ref source-array source-index))

;; update a block of data from a Scheme vector
(array-set-from-vector! display-array 0 #(0 1 2 3 5 6 7 8))
```

Unlike Max buffers, s4m-arrays do not include any thread protection. They are intended to be used in cases where speed of access is the top priority, leaving synchronization issues (and safety!) up to the programmer.

The motivating use case for s4m-arrays is that of driving graphic displays of tabular data as quickly as possible, such as one one would when making a visual display for a step sequencer. In this scenario, one might have one s4m instance that contains a sequencer engine which works with sequence data stored in vectors, and a second instance, running in the low-priority thread off a timer, that drives a graphic display showing this data.

In this scenario, we have an implementation of a **producer-consumer** pattern: we know that only the sequencer will produce data, writing to the s4m-array, and only the consumer will read the data. We also know that if the consumer should get partially updated data (perhaps its thread runs part way through an update from the producer), this is not a serious problem - some ripple in the display as data refreshes is acceptable to the user in the name of realtime performance. Given our strict producer and consumer scheme, and our acceptance of ripple, the s4m-array is preferable to using data structures such as buffer or table, which will run more slowly on account of the thread-synchronization code that they run.

5.9 The s4m.grid object

The missing piece for the scenario just discussed is a display element, and for this purpose Scheme for Max provides a graphical display object, the **s4m.grid**. The grid provides a visual grid on which we can draw values in each cell. It is implemented as a Max UI object, built in the C SDK, and has attributes that may be changed in the Max inspector window for controlling spacing, font size, striping, conversion to MIDI note names, vertical versus horizontal orientation, and whether a value of zero should be drawn or remain blank.

The grid can be updated in two ways. The first is to send it a Max list message. On receipt of a list, the grid will update each cell from the list, iterating either by rows then columns or vice versa, depending on the orientation attribute. The second update method is to read directly from a named s4m-array, on receipt of the **readarray** message. In the second case, the grid iterates through the s4m-array using direct memory access (again according to the orientation attribute), updating each cell. Updating from an s4m-array has the speed advantage that no Max atoms or message data structures need to be created and then parsed for each item of data - the numerical data are read directly from contiguous memory by the display function. When driving a large grid from a timer, this has a significant impact on the processing load created. The result of this is that it is practical to have several large grids updating multiple times per second without creating problematic loads.

The intended workflow is that the programmer will have a component of their sequencing system acting as a view driver. This can be code that is run on a periodic timer (perhaps every 100 to 200 ms), queries the desired Scheme structures (such as reading the sequence data vectors from a Scheme sequencer), and writes the data which we want to view into an s4m-array, thus acting as the producer half. On a separate timer (or the same timer if desired), a grid element running in the UI thread will be sent the display message with the name of this array, acting as the consumer and triggering a redraw of the contents.

In this workflow, the s4m-array acts as a framebuffer, a data structure that virtually represents a display element, and the entire system acts as an immediate-mode GUI. An immediate-mode GUIs decouples the display from the data model, making it possible for the display to accurate reflect the current state of sequencing data, regardless of how it was set. This is desirable in an algorithmic music platform as one cannot assume that the state of the sequencing data originates from GUI actions - it could come from autonomous processes, network requests, MIDI input, and the like. The disadvantage of an immediate mode GUI is the processing cost: it is constantly running data queries and updates regardless of whether data has changed. Thus, the low-level speed optimizations of the s4m.grid and s4m-array facilities make immediate-mode GUIs practical where previously they were not. In my personal experiments, comparison with the Max built in jit.cellblock (the built in tabular display element) showed very significant speed increases - from unusable with one 64 x 16 grid, to usable with four 64 x 16 grids with minimal CPU impact.

5.10 Scheduling Functions

Arguably the most important feature of Scheme for Max is its scheduling and timing features, and their integration with the Max threading and transport subsystems. On a surface level, they are quite straightforward: s4m provides functions that allow one to schedule execution of a zero-arity Scheme function at some point in the future, the simplest of these being the **delay** function.

In the example below, an anonymous function is created (in order to make a zero-arity function) and put on the scheduler to execute in 1000 milliseconds. The call to delay returns a handle that can be used to cancel the scheduled function.

```
;; create a lambda function that prints to the console,
;; and schedule it for 1 second in the future, saving the handle
(define my-handle
  (delay 1000
    (lambda () (post "Hello from the future!"))))
```

(continues on next page)

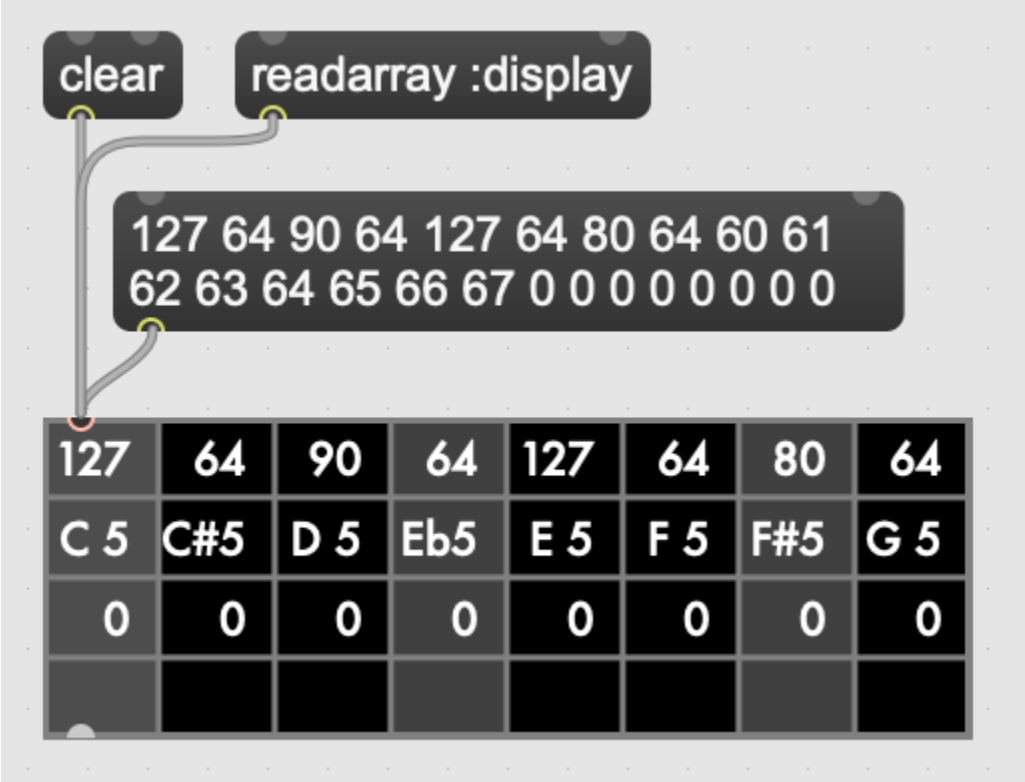

Fig. 2: Figure 3: The s4m.grid display object.

(continued from previous page)

```
;; cancel its execution
(cancel-delay my-handle)
```

The delay function has variants that allow one to schedule in ticks (based on the Max global transport, at 480 ticks per beat), and in quantized ticks, where execution time is forced to align to a tick boundary regardless of at what time the call to delay was made. The quantized tick delay functions will thus only execute if the Max transport is playing, making it possible to synchronize scheduled functions accurately with other Max sequencing tools or with the Ableton Live built-in sequencers.

```
;; schedule my-function for 1 quarter note from now
(delay-t 480 my-function)

;; schedule my-quantized-function for 1 quarter note from now, but forcing now
;; to be interpreted as on the nearest 16th note boundary from the time
;; of the scheduling call, given a running transport
(delay-tq 480 120 my-quantized-function)
```

At an implementation level, these use the Max SDK's **clock** functions, which allow one to precisely schedule execution of a callback function. It is important to note that in modern versions of Max these functions are designed to preserve long-term temporal accuracy regardless of immediate jitter. Jitter, in this context, refers to the difference between the scheduled time and the actual execution time as one would see if analyzing recorded audio.

For optimal real-time audio performance in Max, the recommended settings are to have "audio in interrupt" and "overdrive" enabled. When both of these are turned on, the Max engine alternately runs a DSP pass (calculating a signal vector of samples), and a scheduler thread pass (CTN: Cycling 74 n.d.).

This means that real time of events stemming from the scheduler thread execution can be off by up to a signal vector of samples, resulting in small timing discrepencies. At a signal-vector size of 64 samples (the default for Ableton Live) and a sample rate of 44100 samples per second, this is 1.5 milliseconds, and is thus a musically acceptable discrepency. Note though that the clock functions in current versions of Max compensate for this in the long run such that this discrepency does not accumulate. Tests I made during development confirmed that even after long playback times, clock driven functions did not accumulate jitter, and that if one sets the Max signal vector size to 1 sample, the timing on the clock functions is sample accurate.

The Scheme for Max functions use these clock facilities by putting a reference to the Scheme callback function (the function passed to the delay function) into a special internal registry, keyed by their handles. When the C clock callback runs, the stored handle is retrieved and used to retrive the Scheme callback, which is then exectued.

There is a powerful but not immediately obvious capability granted by the combination of this facility and the nature of Scheme's lexical scoping. This is that Scheme for Max makes it possible to easily specify whether a scheduled function should use values taken from the environment at the time of scheduling, or at the time of execution for which it is scheduled. This is not possible in regular Max patching, and while it is technically possible using JavaScript, it is of limited practical use given the problematic levels of jitter one may have with js object. (As previously discussed, this is because it is always executing in the low-priority thread.)

This facility makes musical algorithms and real-time interaction possible in interesting ways. For example, one might create a patch in which dials or hardware change some musical value. This can be captured, so to speak, at scheduling time, such that when the function executes in the future, the value *where the dial was* is used. Alternatively, one can use a function that explicitly looks in the global environment for settings at run time. Below is an example of a function that uses both of these facilities. The value read from **dial-1** will be used as it was at scheduling time, while the value from **dial-2** will be looked up in the future.

```
;; capture the value of g-dial-1 and use it in the function
;; look up the value of g-dial-2 in the future
(delay-t 480
  (let ((dial-1-capture g-dial-1))
    (lambda ()
      (let ((dial-2-now (eval 'dial-2)))
        (post "dial-1 was:" dial-1-capture)
        (post "dial-2 is:" dial-2-now)))))
```

In combination with s4m's capability of updating code interactively while programs run, this scheduling flexibility enables the programmer-performer to edit algorithms used in a performance in interesting ways, even once they have already been scheduled.

Finally, these facilities enable a workflow known as "self-scheduling" or "temporal recursion", in which a repeating function schedules the next pass of itself (CTN: Lazzarini 2016, 115-116). This enables the composer to create evolving processes, as each pass of a function can change the data, (or even the code!) of the next pass of the function. One might think at first glance that this would result in an accumulation of timing jitter, but the implementation of Max clocks does indeed make this possible while preserving temporal accuracy over long periods of time, something I have tested extensively.

Below is an example of a function scheduling itself. The first iteration of this function would be kicked off by a call to the **start** function, and the temporal recursion will stop when the variable **playing** is set to false.

```
;; a variable to turn on and off playback
(define playing #f)

;; a function that schedules itself to run on every quarter note
;; and keeps track of how many times it has run
(define (my-process runs)
  (post "run number:" runs)
```

(continues on next page)

(continued from previous page)

```
  (if playing
    (delay-t 480
      ;; create an anonymous function that wraps the next call to my-process
      ;; this is necessary as we can only schedule zero-arity functions
      (lambda ()(my-process (+ 1 runs))))))

;; a function to start the process
(define (start)
  (set! playing #t)
  ;; kick it off with the first call
  (my-process 0))

;; a function to stop the scheduling chain
(define (stop)
  (set! playing #f))
```

The above can, of course, be combined with the previously mentioned lexical scoping capabilites, enabling implementations of complex, interactive, algorithmic process music in succint and flexible code. The Scheme for Max online documentation and example repositories contain examples of interactive algorithmic sequencers implemented in this way.

5.11 Garbage collector functions

I have previously referred to the fact that, as a high-level, dynamically-typed language, Scheme includes a **garbage collector** (a.k.a. a **gc**). The garbage collector is a language subsystem that finds and frees memory which has previously been allocated by the program but is no longer needed. Garbage collection spares the programmer the tedious work of manually allocating, tracking, and freeing the memory used by variables in the language. It is a standard feature of most modern high-level programming languages, such as Java, C#, Python, Ruby, JavaScript.

The problem with garbage collection in soft real-time work (such as music, where missed deadlines are undesirable, but not catastrophic) is that the gc must periodically do its work, in which it scans over the program memory, looking for unused memory allocations and freeing them, and this can be a computationally expensive process when the program is large or uses large amounts of data. Further complicating things, garbage collection is of indeterminate duration, as the work that the gc must do is heavily dependent on the particular algorithms and data structures used in the program over which it is running That is to say, a program of some given size and memory use may require more or less garbage collection processing, depending on how precisely it is written. (CTN: Deutsch and Bobrow 1976, 522-523)

For theses reasons, the use of garbage- collected languages is not common in real-time audio programming, where the program must be doing constant calculations to produce streams of samples. Scheme for Max, however, is intended to be used at the *note level*, rather than the *audio level*, thus the typical time between blocks of computation is potentially much higher (i.e., the temporal gap between notes rather than between blocks of samples), giving us potentially adequate time for a garbage collector to run. Modern audio workstations allow a user to configure the output audio buffer size, corresponding to the number of samples the program pre-computes in one block, and thus also corresponding to the latency of real-time operation. This essentially provides the program with a buffer of time during which it can catch up on “bursty” work. While the s7 garbage collector will cause issues if attempting to run Scheme for Max programs in a host with very low output buffer and latency settings (e.g., 64 samples or less), on a modern computer and moderately sized program, the gc is able to run within the latency period of an output buffer of 128 samples or more (depending on the program). This is sufficiently low for playable latency in many situations.

Nonetheless, a heavy Scheme for Max program can run out of time for the garbage collector, resulting in audio underruns and audible clicks. For these cases, Scheme for Max provides some additional facitilites for controlling whether and when the gc runs.

The first of these, perhaps counterintuitively, allows one to control when the gc runs on a timer, allowing it to run *more* frequently than is the case if one does not force a gc run. This increases the overall work the gc does (as it runs more frequently), but lowers how much work it must do on each pass, allowing each pass to complete more quickly.

Sending the **gc-disable** message to the s4m object disables automatic running of the gc, leaving one to explicitly force a run by sending the **gc-run** message, which can be triggered off a timer such as a Max metronome. In my experience, setting this to somewhere between 200 and 300 ms works well and provides better real-time performance than is posssible using the automatic gc, which may wait many seconds between runs.

A second facility is the ability to change the starting heap size of the Scheme for Max object. The lower the heap size, the faster the gc runs, as it must run over less memory. The s4m object takes an initial **@heap** attribute to set the starting heap size. This works well so long as one checks whether the heap allocated will be big enough. If it is not, a *heap reallocation* will be required when s4m is out of memory, which is likely to cause audio issues. Users can use s7's built in gc reporting by turning on the **gc-stats** flag, which will result in output to the console on each gc pass, including the amount of memory it must run over. This can be used to ensure the initial heap size is adequate by running some tests over a given piece or Scheme program to determine the lowest feasible heap size.

Finally, if the performance of a piece is of a reasonable duration, the user may elect to disable the garbage collector altogether. This is done again by sending the **gc-disable** message, but this time without following it by any forced gc runs. In this case, the heap will likely need to be rather large, as the memory use of the program will grow as it runs, with unused memory never getting freed. In programming parlance, this is referred to as a "memory leak", and is normally considered a bug. However, given that the the size of audio sample libraries and personal computer RAM is now commonly in the gigabytes, it is certainly not unreasonable for one to pre-allocate a larger heap and let a program grow in memory on the order of megabytes.

5.12 Summary

This covers the main features and capabilities of Scheme for Max in version 0.4. Additional functions and variations on those discussed here are covered in both the official online documentation and in the Max help file. Additionally, various tutorials with examples are available, and linked from the main GitHub project page.

CHAPTER

SIX

CONCLUSION

To conclude, I will evaluate the success of the Scheme for Max project against the stated criteria and goals, discuss the limitations of Scheme for Max in its current state, and introduce planned and potential areas of future work.

6.1 Evaluation

The Scheme for Max project was started in 2019, with a first usable version released in early 2020. I have thus been using S4M in musical work for three years, and have done so in a variety of contexts: producing mainstream electronic music within the Max for Live environment, creating algorithmic process music in standalone-Max, and creating large Scheme-based frameworks for improvised sequencing. This work has encompassed using Scheme to control virtual synthesizers, voltage controlled analog synthesizers, Max audio objects, the Csound engine (hosted in Max), and the Ableton Live environment through the Live API. While there is certainly room for improvement, and there are limits in the current incarnation with regard to realtime performance, the project has been successful overall in meeting my personal goals for a productive, flexible, and interactive computer music programming platform.

I will briefly discuss the success of the project in the context of the previously discussed project goals.

6.1.1 Focus on programming musical events and event-oriented tools

This goal has been met well by working in Scheme, and in the event-domain of Max. Lisp has made creating my own abstractions of musical events simple to do and has made exploring various algorithmic music techniques fruitful and enjoyable. I am able to work in traditional musical constructs (notes, bars, sections, etc.) but also able to concoct programmatic support for whatever other musical abstractions I choose to think in.

6.1.2 Support multiple contexts, including linear composition, real-time interaction, and live performance

Scoring compositions with Max is an area I have been recently exploring by creating scoring tools with Lisp macros. I am able to score high and low level time scales in Scheme, and this has been productive, flexible, and importantly, resulted in readable code that is pleasant to use in the composition process. Doing so is particularly flexible, as I can change how the macros execute for certain tasks, allowing me to, for example, fast-forward through scores by cueing all called functions up to some playback point. Representing hierarchies of sections of music as functions (as opposed to just notes) has also proven to be very productive and makes rearranging material particularly efficient. Of particular note is that working in text files makes working with multiple time bases very simple.

Meeting the goal of realtime interaction has also been successful. I am able to rapidly build programs for interacting with the system over MIDI hardware. Having done this at various points with all of Max, JavaScript, Csound, and C++, I can personally attest that working in Scheme has sped up development immensely. This comes from both the ability to update only certain sections of code without restarting pieces or programs, and from the flexibility of the

language and its support for symbolic computing and macros. There do exist limits to realtime interaction stemming from performance issues, which I will discuss below, but these have largely been surmountable.

Live performance is feasible with some caveats at this point. However, it should be clarified that this has not yet been tested in actual stage situations. The program is remarkably stable; while crashes were common during the original stages of development, in my current work this does not happen anymore. I would not hesitate to run S4M on stage, provided programs were adequately tested. However, running *large* Scheme programs while also rendering audio does, in the current incarnation, require running with some significant latency (for garbage collection), and certain program behaviour, such as over a hundred delay calls per second, can lead to garbage collection issues after some period of running. This would likely be an issue for users wishing to perform particularly long pieces that use large Scheme sequencing programs and are rendering audio on the same computer. These issues are discussed further below.

6.1.3 Support advanced functional and object-oriented programming techniques

This is an area in which the project has been successful beyond my expectations. While there is some work required of the programmer to learn advanced programming in Scheme, resources for doing so are widely and freely available. The flexibility of being able to work on algorithms as the program runs is helpful, and reimplementing patterns and algorithms that I have previously used in other languages (Max, JavaScript, C, Csound) has been pleasant and fast. Scheme's multi-paradigm approach has been particularly welcome - I use a mixture of imperative, functional, object-oriented, and language-oriented development as the situation warrants.

6.1.4 Be linguistically optimized for the target use cases

Having now developed a substantial body of work in Scheme, both music and tools for music, I personally feel that the linguistic tradeoffs of Lisp, and of Scheme in particular, are eminently suited to balancing the needs of the composer-programmer and tools-programmer. In particular, Lisp macros enable the tools-programmer to create high-level abstractions that require little in the way of code from the composer-programmer, and for which the client code used in pieces matches visually the way one wants to think about music as a composer.

While it would be interesting to also try this kind of programming in a typed language of the ML-family (e.g., OCaml, Haskell, SML), I suspect from my personal experiences that the trade-offs between Lisp and ML languages make the Lisp family more productive and satisfying for the immediate needs of the composer-programmer.

6.1.5 Be usable in conjunction with modern, commercial tools

In this area, the project has again been resoundingly successful. I have spent much of the last year (2022-2023) developing a large-scale platform for creating music in Ableton Live in which S4M is used for improvised sequencing, hardware control, algorithmic processes, mix automation, and scoring. This has not only worked well, but I am able to combine it with material coming from regular Ableton Live clips with near-perfect synchronization (infact, as close to perfect as it is possible to get when using message-events in Max for Live of any kind). Scheme for Max introduces no further timing discrepencies than does Max for Live itself, where jiter of up to one signal vector (approximately 1.5 ms at 44100 samples per second) is possible, and is usually musically neglible.

Of particular note, being able to control the Live environment through the Live API with S4M has been particularly successful. The Live API provides a way for Max patches to programmatically control most elements of the host in real-time, including mixer settings, device settings, the system transport, sequencer data, and much more. It uses an object model, the Live Object Model (or LOM), that is based on hierarchal lists of symbols. Users construct lists of numbers and symbols, and send these to special Max objects to query and control elements of the API (CTN: Cipriani 2020, 676-680). Given that the fundamental model is of lists of symbols, it thus lends itself well to implementation in Lisp. I have created a low-level interface object for working with the Live API, **live-api**, and an accompanying Max subpatch (provided with S4M), which together provide the functions **send-path** and **call**, to which LOM paths can be passed. Building on this, adding specific functions to accomplish various tasks with the Live API requires minimal code:

```
;; Live API functions to start and stop clips and get/set device params
(define (fire-clip track slot)
  (live-api 'send-path (list 'live_set 'tracks track 'clip_slots slot 'clip)
    '(call fire)))

; as above, but using back-tick lisp syntax
(define (stop-clip track slot)
  (live-api 'send-path `(live_set tracks ,track clip_slots ,slot clip)
    '(call stop)))

(define (get-device-param track device param value)
  (live-api 'send-path `(live_set tracks ,track devices ,device parameters ,param)
     `(get value)))

(define (set-device-param track device param value)
  (live-api 'send-path `(live_set tracks ,track devices ,device parameters ,param)
     `(set value ,value)))
```

6.1.6 Support composing music that is impractical on commercial tools

I have found Scheme for Max particularly appropriate for composing and programming works that are not practical or are difficult on mainstream sequencers (e.g., Live, Logic, Reaper). By using Scheme as the top-level orchestration layer, whether through score facilities or algorithmic processes, implementing pieces with complexities such as shifting or multiple concurrent meters is straightforward, as is manipulating time across multiple scales at once, such as gradually changing the tempi of different voices at different rates.

Similarly, S4M is well suited to exploring spectral music and other techniques in which the line between a component of a sound and a note from an instrument is blurred. For example, if one wants to apply spectral composition techniques such as controlling many partials of many sounds independently, this is straightforward by combining Scheme for Max with the csound~ object, and far simpler than would be the case with plain Max. Scheme programs can create programmtic loops that send Csound score messages representing activations of sine waves. Having previously experimented with this using Max, Csound, and the combination of the two, I have found the addition of S4M to be a tremendous improvment.

Overall, I feel that the achievement of this goal is one of Scheme for Max's strongest points, and that S4M has the potential to be a significant contribution to the computer music tool landscape in this area.

6.1.7 Enable iterative development during musical playback

The support for interactive development has been another area area in which Scheme for Max has succeeded beyond my expectations. For my personal work configuration, I have created two small scripts in Python and Vim respectively, which enable me to send Scheme code to Max directly from my text editor. This is achieved by having Vim commands send a selected area (the enclosing parenthetical expression) to standard input (STDIN) of a short Python program, which in turn sends the text over the local network as an Open Sound Control message to the Max **udp** object, from where it is passed to an s4m object for evaluation.

I am thus able to work on code in my editor, and in two keystrokes, send blocks of it to Max to run. I have used this to create hotkeys for starting and stopping Live, reloading my project, and resetting the interpreter, and have created short convenience functions that I can evaluate from the editor to cue works to certain places, mute tracks, arm devices, and the like. The results of these operations (whatever I make the functions return) are printed on the Max console, and I am also able to use the Max console to inspect data structures interactively. Of particular note is the ability to change functions even once they are scheduled. This capability is something I have found exceptionally valuable while working on algorithmic music.

I feel that this is also an area where Scheme for Max can contribute significantly to the computer music landscape, providing a live-coding platform that does not need to be insulated from mainstream tools.

6.1.8 Evaluation Summary

To conclude the evaluation, I feel the project has been almost entirely successful in meeting its stated goals. The one area of concern that remains is suitability for live performances that use realtime interaction with large programs and would benefit from being able to run with lower latency. However, as the current s7 interpreter was not designed for realtime use (indeed upon the first release of S4M, its success in this regard was received with suprise and enthusiasm by its author), I believe this is an area in which future work on optimizing s7 and Scheme for Max for realtime performance will bear fruit.

6.2 Limitations and Future Work

Finally, I will discuss the limitations of Scheme for Max in its current incarnation and the planned and potential work on and with the project.

6.2.1 Limited Integrations

At present, Scheme for Max provides new facilities to Max, but does not integrate with other Max extensions. As a result, many users who would benefit from S4M are not aware of its capabilities - it is the kind of thing they need to find on their own. A notable item of planned work that will help address this is implementing an integration with the Bach project.

Bach (the Bach Automated Composer's Helper) is a long-standing open-source project that provides Max objects for accomplishing computer-assisted composition tasks similar to those available in Lisp-based platforms such as Patchworks and OpusMondi. Bach does this by supporting what the project calls "llls" – Lisp-like linked lists – a high-level data type corresponding to the Lisp list in its ability to nest and to hold heterogenous data. In addition, the Bach project, and its extensions such as Cage and Dada, provide a wide variety of objects for working with these lists, including sophisticated graphical elements such as staff notation displays and piano rolls. Bach uses llls in a similar fashion to how Max uses dictionaires and S4M uses s4m-arrays: the data is stored in a global Bach-controlled registry, and objects can pass references to these between them (CTN: Agostini 2015, 11-27). However, while being inspired by Lisp data structures and Lisp-based platforms, Bach is notably missing an interactive Lisp interpreter itself. Were Scheme for Max also able to work with Bach llls, the capabilities of both Bach and S4M would be significantly increased, and the number of users interested in Scheme for Max would likely also increase significantly.

One of the next major initiatives planned for S4M development is developing an integration layer for Bach, and I have met with Andrea Agostini, one of the Bach developers, to discuss plans already. This work is planned for the summer and fall of 2023.

6.2.2 Real-time Scheduling

As previously mentioned, there is an issue that manifests itself when programs making particularly large numbers of delay calls are run for long periods, especially while the computer is also doing significant other work (e.g., rendering audio in plugins). This manifested itself on my system only after I began working on pieces in Ableton Live in which 16 different Scheme sequencers were running concurrently, each making a new delay call on each 16th note, thus producing on the order of 100 delay calls per second (depending on the tempo). After some period of time of running without a reset of the interpreter, such as 10 minutes or so, CPU use becomes too high for realtime rendering. The behaviour is similar to what happens when the audio latency is too low or the heap size is too high, both situations where the garbage collector cannot finish in time. It thus seems likely (though at this point this is speculation) that the memory over which the GC is running has inadvertenly grown, and there is a bug in my implementation of the

scheduled function callback handling that prevents the garbage collection of already scheduled functions. This is the most serious limitation at the moment and is something on which I will be actively working in the summer of 2023.

6.2.3 Garbage Collection

In addition to the bug in my implementation, there is the fact that the s7 garbage collector is not designed for realtime use. There has been significant work in recent years on garbage collection algorithms, including the development of various approaches for soft-realtime gargage collectors such as incremental collectors. An incremental collector does not finish all its work on every pass, and would likely perform better in an audio situation as the work can be distributed over time. Audio computation is, by its nature, "bursty", with much work happening during the computation of the audio blocks corresponding to times with many note onsets. Allowing the gc to leave unfinished business until a subsequent pass, and giving the user the opportunity to configure how this is done, has the potential to significantly lower the latency at which Scheme for Max can be used. This, however, will require major development work, and should be considered a long-term potential area of exploration.

6.2.4 Thread Limitations

At present, the user can choose between running the s4m object in the low-priority main thread or high-priority scheduler thread, but cannot run the interpreter in the audio thread. Were it possible to run an instance in the audio thread, S4M could be used to produce audio signals at single sample temporal accuracy. The previously discussed jitter of event onsets in Max is only an issue for Max *event messages*. Generating timing data as part of an audio stream is not affected. (CTN: Lyon 2012, 121-179) This could be useful for those wishing to sequence synthesizers controlled by control voltages, as this is done in modern audio workstations by outputing control voltage signals as audio streams. While Scheme, as a high-level language with a garbage collector, is unlikely to be appropriate for heavy digital signal processing, control voltage signals do not necessarily need to be created at the same bit-depth or sample rate as regular audio to be useful. For example, in the Csound language, it is common to use *k-rate* signals, generated at a divisor of the sample rate, to control many attributes of synthesis. These can be generated at lower resolution, and one can use interpolation when a smoother output signal is needed (CTN: Smaragdis 2000, 126-128). It is thus possible that creating control rate signals for purposes such as control-volt gates (controlling note onsets), envelopes, and low frequency oscillators could all be practical in Scheme.

This would require creating a variant of the Scheme for Max object that would run the Scheme interpreter within the Max audio rendering loop, and use some form of thread-safe queuing to pass Max messages in and out of the scheduler or main thread. It is likely that this would be more practical when used in conjuction with an improved garbage collector. While control rate signals generated from Scheme are unlikely to be possible with the same latency as those generated from C (given the unavoidable extra computation), the convenience of doing so may well make the endeavour worthwhile, especially as computers continue to become faster.

Running in the audio thread could also make it possible to create objects that combine Scheme for Max and other audio systems in one Max object. This could be used, for example, to create a Scheme-capable Csound object, in which Scheme functions that directly access the Csound API could interact with Csound at a deeper and more temporally accurate level than is currently possible with the scenario of a separate s4m and csound~ object.

6.2.5 Difficulty of Extension

Scheme for Max is open-source software, licensed under the permissive BSD license, enabling any one to extend it if desired. This is potentially attractive to users who would like to integrate Scheme code with processes that will be faster to execute in C. The s7 foreign function interface makes this quite straightforward - it does not require much in the way of code to add a C function that can be called from Scheme and vice versa, and this was indeed one of the motivations for choosing s7. However, the programming logistics around doing so are prohibitively cumbersome: one must go through all the setup necessary to create a Max extension with the Max SDK, and one must also navigate and alter the main s4m.c file.

A potential area of work to address this would be the creation of plugin system or automated compilation system for Scheme for Max extensions. This could even use other languages that compile to C, such as Zig or OCaml. While I feel this would be a powerful additional piece of functionality, the target user base for this feature is likely very small. This is thus a long-term potential area of exploration.

6.3 Conclusion

In conclusion, I believe the Scheme for Max project has been successful and has the potential to make a significant contribution to the landscape of computer music programming. It succeeds in making programming in Lisp accessible and convenient, and enables the programmer to work in a productive, flexible, and exploratory manner alongside commercial and research-oriented tools alike. I believe it provides much needed capabilities to both patching platforms and to textual DSLs as an orchestration layer, and makes the development of sophisticated and complex music more attainable. Scheme's flexibility and power make it an ideal glue language in a multi-language environment, allowing users to bridge previously separated tools, approaches, and techniques. And finally, I believe, and certainly hope, that the addition of Scheme to Max and the Ableton Live platform will introduce many new and potential programmers to the joy of programming music in Lisp.

CHAPTER

SEVEN

REFERENCES

Ableton. 2009. "Ableton releases Max for Live." Press Archives. Last modified November 23, 2009. https://www.ableton.com/en/press/press-archive/press-archive-mfl-release/

Ableton. 2017. "Ableton and Cycling '74 Form a New Partnership". News. Last modified June 7, 2017. https://www.ableton.com/en/blog/ableton-cycling-74-new-partnership/

Agostini, Andrea, and Daniele Ghisi. 2015. "A Max Library for Musical Notation and Computer-Aided Composition." *Computer music journal* 39, no. 2 (2015): 11–27.

Ariza, Christopher. 2009. "Sonifying Sieves: Synthesis and Signal Processing Applications of the Xenakis Sieve with Python and Csound." In *International Computer Music Conference Proceedings.* Vol. 2009. Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

Boulanger, Richard. 2013. "Introducing Csound for Live." In *Ways Ahead Proceedings of the First International Csound Conference*, edited by Joachim Heintz, Alex Hofmann, and Iain McCurdy, 188. Newcastle: Cambridge Scholars Publishing.

Carrette, George. 2007. "SIOD: Scheme in One Defun." Last modified April 5, 2007. http://people.delphiforums.com/gjc//siod.html

Cipriani, Alessandro, and Maurizio Giri. 2019. *Electronic Music and Sound Design : Theory and Practice with Max/MSP Volume 1*. Rome: ConTempoNet.

Cipriani, Alessandro, and Maurizio Giri. 2020. *Electronic Music and Sound Design : Theory and Practice with Max/MSP Volume 2*. Rome: ConTempoNet.

CLiki. n.d. "Music." The Common Lisp Wiki. Last accessed July 2, 2023. https://www.cliki.net/music

Cook, Perry R. 2017. *Real Sound Synthesis for Interactive Applications.* Abingdon: CRC Press.

Cycling 74. n.d. "Javascript and Threading." Max v8.5.5 Documentation. Last accessed July 2, 2023. https://docs.cycling74.com/max8/vignettes/jsthreading

Cycling 74. n.d. "Max JS Tutorial 1: Basic JavaScript." Max v8.5.5 Documentation. Last accessed July 2, 2023. https://docs.cycling74.com/max8/tutorials/javascriptchapter01

Cycling 74. n.d. "Scheduler Settings." Max v8.5.5 Documentation. Last accessed July 2, 2023. https://docs.cycling74.com/max8/vignettes/scheduler_settings

Cycling 74. 2019. "Min Guide to Threading in Max." Min Development Kit. http://cycling74.github.io/min-devkit/guide/threading

Dannenberg, Roger B. 1997. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis." *Computer music journal* 21, no. 3 (1997): 50–60.

Dannenberg, Roger B. 2018. "Languages for Computer Music." *Frontiers in digital humanities* 5 (2018).

Dannenberg, Roger B., Peter Desain, and Henkjan Honing. 1997. "Programming Language Design for Music." In *Musical Signal Processing*, edited by Curtis Roads, 291. London: Routledge.

Deutsch, L. Peter and Daniel G. Bobrow. 1976. "An efficient, incremental, automatic garbage collector". *Communications of the ACM* 19, 9 (Sept. 1976), 522–526.

Farnell, Andy. 2010. *Designing Sound.* Cambridge, Mass: MIT Press.

Gogins, Michael. 2013. "The Csound API: Interview with Michael Gogins." In *Ways Ahead Proceedings of the First International Csound Conference*, edited by Joachim Heintz, Alex Hofmann, and Iain McCurdy, 43. Newcastle: Cambridge Scholars Publishing.

ffitch, John. 2011. "Using C To Generate Scores." In *The Audio Programming Book.*, edited by Richard Boulanger and Victor Lazzarini, 655. Cambridge, Mass: MIT Press.

ffitch, John. 2011. "Understanding an Opcode in Csound." In *The Audio Programming Book.*, edited by Richard Boulanger and Victor Lazzarini, 581. Cambridge, Mass: MIT Press.

Lazzarini, Victor. 2011. "MIDI Programming with PortMIDI." In *The Audio Programming Book.*, edited by Richard Boulanger and Victor Lazzarini, 783. Cambridge, Mass: MIT Press.

Lazzarini, Victor. 2013. "The Development of Computer Music Programming Systems." *Journal of new music research* 42, no. 1 (2013): 97-110.

Lazzarini, Victor. 2016. *Csound A Sound and Music Computing System.* Cham: Springer International Publishing.

Lazzarini, Victor. 2017. *Computer Music Instruments: Foundations, Design and Development*. Cham: Springer International Publishing.

Lyon, Eric. 2006. "A Sample Accurate Triggering System for Pd and Max/MSP." In *International Computer Music Conference Proceedings.* Vol. 2006. Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

Lyon, Eric. 2012. *Designing Audio Objects for Max/MSP and Pd.* Middleton, Wisconsin: A-R Editions.

Maldonado, Gabriel. 2011. "Working With Audio Streams." In *The Audio Programming Book.*, edited by Richard Boulanger and Victor Lazzarini, 329. Cambridge, Mass: MIT Press.

Manning, Peter. 2004. *Electronic and Computer Music*. New York: Oxford University Press.

maxobjects.com. n.d. "Max Objects Database." Last modified May 4th, 2023. http://maxobjects.com/

McLean, Alex, and R. T. Dean. 2018. *The Oxford Handbook of Algorithmic Music.* Edited by Alex McLean. New York, NY: Oxford University Press.

Miller, Alex, Stuart Halloway, Aaron Bedra, and Jacquelyn Carter. 2018. *Programming Clojure.* Edited by Jacquelyn Carter. Third edition. Place of publication not identified: Pragmatic Programmers.

Puckette, Miller. 1991. "Combining Event and Signal Processing in the MAX Graphical Programming Environment." *Computer music journal* 15, no. 3 (1991): 68–77.

Puckette, Miller. 2002. "Max at Seventeen." *Computer music journal* 26, no. 4 (2002): 31–43.

Roads, Curtis. 2015. *Composing Electronic Music : a New Aesthetic.* Oxford: Oxford University Press.

Roberts, Charlie, and Graham Wakefield. 2018. "Tensions and Techniques in Live Coding Performance." In *The Oxford Handbook of Algorithmic Music.*, edited by Alex McLean, 293. Oxford: Oxford University Press.

Sandewall, Erik. 1978. "Programming in an Interactive Environment: The Lisp Experience." *ACM computing surveys* 10, no. 1 (1978): 35–71.

Schottstaedt, William. n.d. "s7." Accessed July 2, 2023. https://ccrma.stanford.edu/software/snd/snd/s7.html

Smaragdis, Paris. 2000. "Optimizing Your Csound Instruments" In *The Csound Book*, edited by Richard Boulanger, 123. Cambridge Mass: MIT Press.

Sun, Kwangwon, and Sukyoung Ryu. 2017. "Analysis of JavaScript Programs: Challenges and Research Trends." *ACM computing surveys* 50, no. 4 (2017): 1–34.

Taube, Heinrich. 2009. *Common Music 3. International Computer Music Conference Proceedings.* Vol. 2009. Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

Taylor, Gregory. 2020. "JavaScript: A Brief Resource Guide for Max Users." Cycling74 Articles. Last modified Nov 10, 2020. https://cycling74.com/articles/javascript-a-resource-guide-for-max-users

Touretzky, David S. *LISP: a Gentle Introduction to Symbolic Computation*. New York: Harper & Row, 1984.

Wang, Ge. 2017. "A History of Programming and Music." In *Cambridge Companion to Electronic Music*, edited by Nick Collins and Julio D'escrivan, 58-74. Cambridge: Cambridge University Press.

Zicarelli, David. 2002. "How I Learned to Love a Program That Does Nothing." *Computer music journal* 26, no. 4 (2002): 44–51.