Scheduling Musical Events in Max/MSP with Scheme For Max

ANONYMOUS AUTHOR(S)

Scheme For Max (S4M) is an open source project that enables embedding s7 Scheme interpreters in the Max/MSP visual programming environment, the most widely adopted programming platform for computer music. Of particular note is its appropriateness for flexibly and accurately scheduling future musical events, a capability not adequately supported thus far in Max. This is accomplished by scheduling Scheme procedures integrated with the Max scheduler, and the implementation of this facility is discussed. In addition to scheduling, S4M enables users of Max to script Max generally in Scheme and provides facilities for real-time dynamic code evaluation (a.k.a. "live-coding") within the Max environment.

CCS Concepts: • **Applied computing** → **Sound and music computing**; **Media arts**.

Additional Key Words and Phrases: music computing, event scheduling, live-coding

ACM Reference Format:
Anonymous Author(s). 2021. Scheduling Musical Events in Max/MSP with Scheme For Max. 1, 1 (June 2021), 20 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Max (also known as Max/MSP) is a programming environment for creating interactive music and multi-media programs through a visual programming language accessible to non-programmers. Max was created to make it possible for users to go beyond the limits of commercial music sequencing tools, creating interactive environments of arbitrary complexity and sophistication [16]. Programs, or "patches" in the Max nomenclature, are created by placing visual boxes representing Max objects on a canvas and connecting them visually with "patch cords", a paradigm similar to that of modular synthesizers and familiar to many musicians. First created in the mid 1980's by Miller Puckette while at IRCAM (Institut de recherche et coordination acoustique/musique), Max is now developed and sold by the San Francisco software company Cycling '74, and is widely used in both academic and commercial music contexts as well as in multi-media installations. A rich library of Max objects exists, both provided with Max and available as open-source extensions, enabling users to rapidly create interactive systems with gestural input from physical sources such as on screen widgets, physical electronic instruments, and custom hardware communicating over serial networks.

One powerful feature of Max is its ability to be programmed while the engine is playing music. Patches can be altered without necessarily interrupting patch activity (depending on the design of the program), and this can even be performed live, an activity known as "live-coding", which has emerged since the early 2000's as its own musical sub-culture of live programming performances with tools such as Max, SuperCollider, Csound, and others [10].

XXXX-XXXX/2021/6-ART $15.00
https://doi.org/10.1145/1122445.1122456

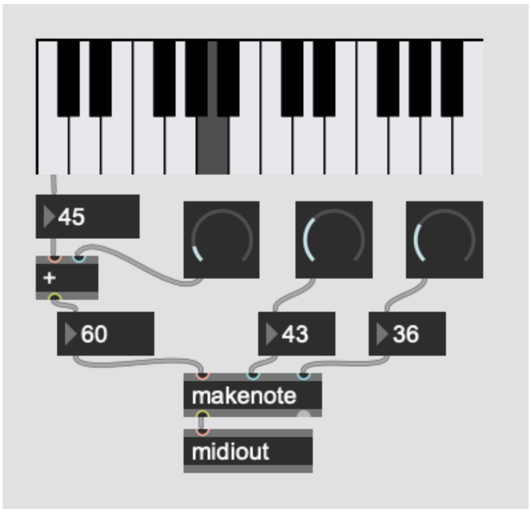


Fig. 1. A Max patch with a keyboard value transposed by a dial and sent to MIDI output.

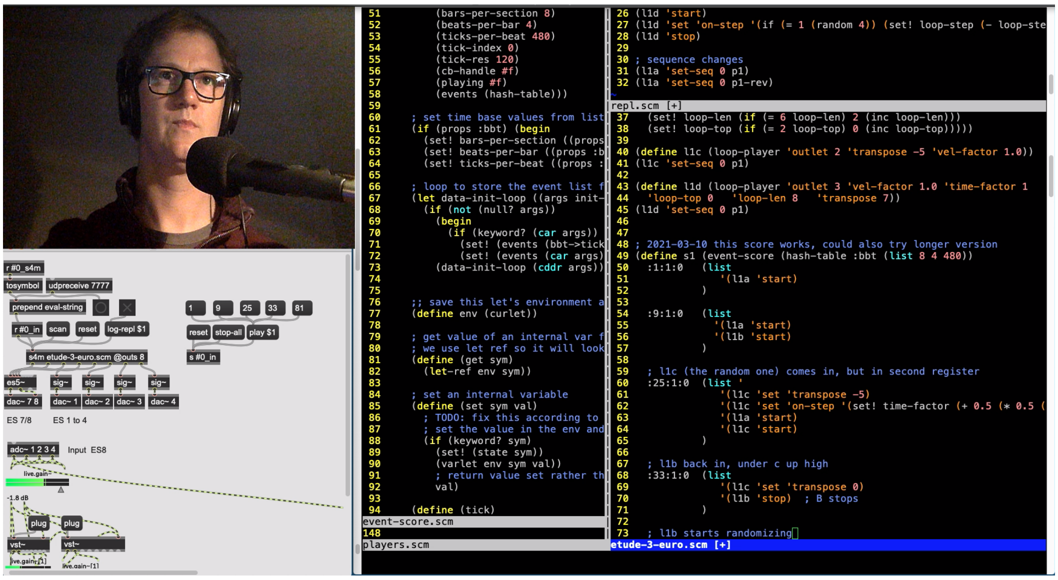
This approach to music making overlaps with the related discipline of algorithmic music, in which programmatic algorithms are used not just for affecting musical parameters such as volume, pitch, and timbre, but also for the generation of musical content such as melodies and rhythms.

In both the fields of live-coding and algorithmic music, the ability for the performer/-composer to schedule events in the future with high temporal accuracy is of major benefit. While this is possible in the Max visual patching language, it is cumbersome and thus the implementation is not optimal for live-coding or algorithmic music where the speed with which the user can accomplish this is an important consideration (i.e., fast enough not to bore an audience or to be frustrating while composing).

In addition to the Max visual patching language, Max can be programmed by creating new Max objects ("externals" in the Max nomenclature) in C or C++ using the Max SDK and API, and can also be programmed in text-based languages through regular Max objects that themselves provide embedded language interpreters. Max provides one such interpreter in the **js** object, which embeds a JavaScript interpreter, and others contributed by third parties exist for languages such as for Lua, Python, and Ruby.

Scheme For Max (a.k.a. S4M) is an open-source external developed by the author that embeds the s7 Scheme interpreter, a Scheme dialect created by Bill Schottstaedt at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University. Originally developed from TinyScheme, s7 is a Scheme dialect designed for use in computer music platforms and is used in various music programs such as the Snd editor and the Common Music algorithmic composition platform [11]. Similar in functionality to the built in **js** object, the **s4m** object enables the user to program Max in s7 Scheme, and provides Scheme functions to interact with the Max engine and environment through a foreign function interface implemented in C using the Max C SDK and API.

The author believes that S4M extends Max in a way that is of significant value to the algorithmic musician and live-coding performer, as well as more generally to to the broader Max community. The ability to update a running Scheme program during playback is a major benefit and is a capability that has been, while technically possible, impractical with existing solutions such as JavaScript. Additionally, the syntax of Scheme bears a convenient similarity to Max message syntax, and thus one can easily use Max visual widgets and messages to generate small Scheme programs in the patcher. Potentially the most interesting benefit is the ease with which the user can create and schedule functions for evaluation in the future, with input into these functions coming from Max patching widgets, and flexible control over the current and future evaluation context of the functions and variables used.



```
            (bars-per-section 8)
            (beats-per-bar 4)
            (ticks-per-beat 480)
            (tick-index 0)
            (tick-res 120)
            (cb-handle #f)
            (playing #f)
            (events (hash-table)))

      ; set time base values from list
      (if (props :bbt) (begin
        (set! bars-per-section ((props
        (set! beats-per-bar ((props :b
        (set! ticks-per-beat ((props :

      ; loop to store the event list f
      (let data-init-loop ((args init-
        (if (not (null? args))
          (begin
            (if (keyword? (car args))
              (set! (events (bbt->tick
              (set! (events (car args)
          (data-init-loop (cddr args))


      ;; save this let's environment a
      (define env (curlet))

      ; get value of an internal var f
      ; we use let ref so it will look
      (define (get sym)
        (let-ref env sym))

      ; set an internal variable
      (define (set sym val)
        ; TODO: fix this according to
        ; set the value in the env and
        (if (keyword? sym)
          (set! (state sym))
          (varlet env sym val))
          ; return value set rather th
          val)

      (define (tick)
event-score.scm

players.scm
```

```
(l1d 'start)
(l1d 'set 'on-step '(if (= 1 (random 4)) (set! loop-step (- loop-ste
(l1d 'stop)

; sequence changes
(l1a 'set-seq 0 p1)
(l1a 'set-seq 0 p1-rev)
~
repl.scm [+]
  (set! loop-len (if (= 6 loop-len) 2 (inc loop-len)))
  (set! loop-top (if (= 2 loop-top) 0 (inc loop-top)))))

(define l1c (loop-player 'outlet 2 'transpose -5 'vel-factor 1.0))
(l1c 'set-seq 0 p1)

(define l1d (loop-player 'outlet 3 'vel-factor 1.0 'time-factor 1
  'loop-top 0   'loop-len 8   'transpose 7))
(l1d 'set-seq 0 p1)


; 2021-03-10 this score works, could also try longer version
(define s1 (event-score (hash-table :bbt (list 8 4 480))
  :1:1:0   (list
              '(l1a 'start)
           )

  :9:1:0   (list
             '(l1a 'start)
             '(l1b 'start)
           )

  ; l1c (the random one) comes in, but in second register
  :25:1:0  (list '
              '(l1c 'set 'transpose -5)
              '(l1c 'set 'on-step '(set! time-factor (+ 0.5 (* 0.5 (
              '(l1a 'start)
              '(l1c 'start)
           )

  ; l1b back in, under c up high
  :33:1:0  (list
              '(l1c 'set 'transpose 0)
              '(l1b 'stop)  ; B stops
           )

  ; l1b starts randomizing
etude-3-euro.scm [+]
```

Fig. 2. Demonstration of real-time algorithmic music by the author, in which S4M is used to control a Eurorack modular synthesizer. https://youtu.be/pg7B8h4yHkU & https://youtu.be/rcLWTjN4qBI

This paper provides an overview of the Max platform and the problem of musical scheduling in computer music, as well as the existing Max solutions to this problem, and a discussion of the limitations of these solutions. It then introduces Scheme For Max and examines how the use of Scheme overcomes these limitations, and provides details of the implementation of scheduling in S4M, both at the Scheme and C programming levels. Finally, it concludes with discussion of the current limitations of, and future possibilities for, Scheme For Max.

2 BACKGROUND - PROGRAMMING MUSIC IN MAX/MSP

2.1 The Max Environment

Max is a visual programming environment for interactive multi-media, used widely in music academia as well as in commercial music circles through Max for Live, a version of Max embedded in the Ableton Live digital audio workstation. Max patches are created by placing visual boxes on a canvas and connecting them graphically with "patch cords", where a box may be any of the Max object types installed on the user's system. A box placed on the patcher results in the instantiation of an object in memory from a prototypical class for the object, with text fields typed in the visual patching box used as constructor arguments. Thus a Max patch consists of a collection of instantiated objects that send messages to each other in a directed graph, producing a data-flow execution model whereby a message from a source object triggers execution in one or more receiving objects, who may in turn send on messages to other similarly connected objects.

Patch activity, in the form of messages moving through the graph, can be initiated by various forms of real-time input, such as keyboard and mouse events, connected electronic instruments, and networking events, as well as by scheduled events through Max objects such as the **metronome**, which sends out messages at regular time intervals. Messages are stored internally as lists of Max "atoms", which may be symbols, integers, or floating point numbers [9]. There also exists a Max object for visually displaying and altering messages called the **message-box**, which allows messages to be typed directly in the object in the visual patcher, and then sent by clicking the box. Execution follows a depth-first and right-to-left order, enabling the programmer to deterministically control the execution flow with the visual layout of the patch cords. (i.e., A source object sending messages out to multiple receiving sub-graphs results in the right-hand message path completing execution to the bottom before moving left, rather than spawning two concurrent threads of execution.) When patching, messages can be inspected by sending them to a **print** object (which prints to the Max console), to a **message-box** object (which will update its visual display of the message) or through a built-in visual debugger using a feature Max calls "probing".

In addition to this event-based message execution model, also called "control messages", Max supports a stream-based digital audio execution model, originally provided separately as a product called "MSP", but now included as part of the single Max product. MSP activity normally runs in a separate thread from the event/message Max operations, and uses a a separate class of objects that pass constant streams of digital audio to each other through differently colored patch cords (though MSP objects may additionally receive control messages for controlling parameters). As S4M executes only at the event/message level and does not implement DSP operations, MSP is not discussed further here.

2.2 Max Message and Object Implementation

Internally, Max messages are data entities consisting of a symbol that acts as a Smalltalk-style message selector and an optional array of Max atoms. Each atom entity contains a member for the atom type and another for its value, where the type may be any of **int, float,** or **symbol**. Note that while the message selector symbol is always present at the C level, in the visual patcher the selector may be implicit and hidden from the user. (i.e., the message originating from a message box that appears to contain only an integer will actually consist of the selector "int" followed by an atom storing the numerical value.) The symbol "bang" is a special symbol that can be used for a one-element message that essentially means "run", and the act of triggering execution by sending a bang message from the **bang** object is called "banging" in the nomenclature. Taken together, this means that in Max there are five kinds of messages: **bang**, **symbol**, **int**, **float**, and **list**. (The bang message is technically a symbol message of "bang", but this is essentially treated as a type of its own in the nomenclature.) [9]

Activity in an object (represented visually by a single visual box in the patch) is triggered by sending the object a Max message, most commonly from an object connected to it through a patch cord running to an "inlet" of the object. In order for the receiving object to do anything, it must be sent a message with a message selector for which it has a bound method, a situation referred to in the nomenclature as "responding to the message". Most objects have a principal activity that typically ends with outputting the result of their calculation, and this is often triggered by sending the object a single **bang** message or by sending a value to their first, or "hot", inlet. In addition, they may respond to other messages to change internal state data or configuration - for example the “set” message is commonly implemented to update internal state without outputting any result. The end result of object methods run on receipt of a message falls broadly into three (non-exclusive) categories: the object may update some internal state, it may send a message or messages out of its outlets, and it may cause a side effect in the broader Max environment, such as printing to the console or updating a global data element such as an audio buffer.

As an example, in the patcher screenshot above we see a **message-box** object that will update its state (and visual display) with the symbols “hello” and “world” when sent the message **set hello world**. Near it is a circular **bang** object, which when clicked will send the **message-box** the **bang** message, causing it to output the message **list hello world**. This message will be received by the **zl.len** object, which counts the number of elements in any lists it receives, immediately outputting the count. The three columns in the screen-shot show the patch as it is prior to any clicks, as it is after clicking **message-box** 1, and as it is after clicking both **message-box** 1 and the **bang**.

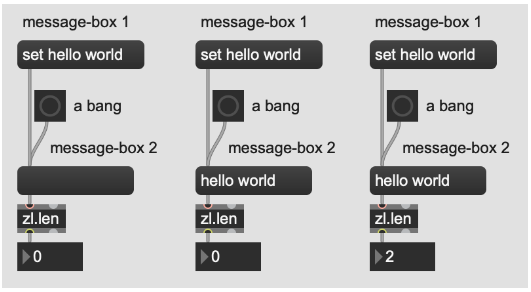


Fig. 3. Message flow at three stages

Objects are not limited to interacting with other objects through messages passing into inlets and out of outlets. A C API exists that enables objects to query and control various engine components (e.g., the transport mechanism), and it is also possible for objects to send messages to other objects directly or through the scheduler queues, without the sending and receiving objects necessarily being connected visually in the patcher. Fundamentally however, the same mechanism is used - when an object calls a method on another object at the C level, this is still done by creating a data structure of a message selector and optional atom arguments, and then sending this to the receiving object through a generic message sending function.

As each visual box in a patcher has state that is retained between messages, we can see that when programming the visual patcher (a.k.a. “patching”), we are in effect programming with object instantiations rather than classes. In fact, in the Max engine, the act of creating a new visual-box and placing it on the patcher canvas does indeed instantiate an object, creating a fresh copy of the prototype’s data structure and adding it to a graph of other objects. This is in contrast to computer music languages such as Csound, where the user programs instruments with functions and the engine creates an instantiated data structure on each note-event sent to the instrument from a score - in effect the instruments act as object builders and note-events become objects that exist for the duration of the note played [6].

2.3 Max Externals

While Max is a commercial, closed-source product, it includes a software development kit for extending Max by writing a Max "external". An external is a compiled plug-in that defines the prototype (data structure and methods) used to create new objects in the patcher. Externals are developed in C or C++ in an object-oriented manner, the C API using data structures and pointers to simulate class-based programming with dynamic binding, and the more recent C++ API using C++ classes [16]. A typical external will implement a class that provides some object state for instantiated objects along with constructor and destructor functions, and methods for sending and receiving Max messages through the object's inlets and outlets. Methods are bound dynamically to the object prototypes. The SDK and API also provide a rich body of functions for interacting with the overall Max environment, thus these methods may also trigger side-effects through these facilities.

As externals are compiled plug-ins, they do not need to be distributed as source-code, and are most typically made available as binaries for Windows and OSX. Extending Max through externals has been possible since very early versions of Max in the late 1980's, thus thousands of 3rd party externals now exist [7], both open and closed source, of which Scheme For Max is one.

2.4 Lisp in Computer Music

Scheme For Max is far from the first Lisp-based computer music tool, or even the first real-time music tool in Lisp. There is a rich history of Lisp in music, ranging from Common Lisp Music, created by Bill Schottstaedt in the late 1980's [15] and Heinrich Taube's Common Music in 1991 [13], to Dannenberg's Nyquist in 1997 [4], and Impromptu (later rewritten as Extempore) [12]. Nor is Scheme For Max unique in providing a Lisp based extension facility to a commercial music platform - the author unknowingly programmed in Lisp in the early 1990's while using the Cakewalk sequencer, which came with an embedded Lisp interpreter in the form of the Cakewalk Application Language (CAL).

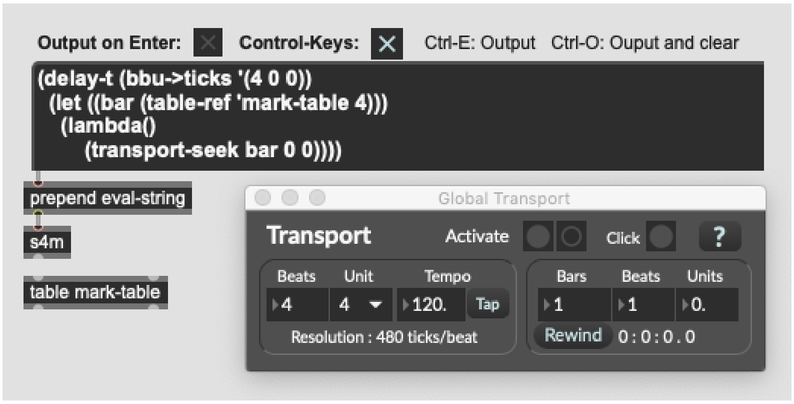
However, Scheme For Max is unique in bringing a fully-fledged Scheme interpreter to the Max environment as a first-class Max object, implemented and running identically to any built-in object. This is significant in that Max is arguably the most widely used truly programmable music platform, and certainly the most widely deployed through its use in the very successful Ableton Live digital audio workstation, as Max For Live.

In contrast with S4M, systems such as Common Lisp Music, Common Music, Nyquist, and Extempore all run as stand-alone applications, with events originating from them either rendered as audio internally or destined for other systems for rendering to audio. While these systems are, like Max, also capable of receiving gestural input, they must, in essence, "be their own boss" - they provide their own scheduler and timing clocks and act as the principal engine from which temporal events originate. If they are to be used alongside another temporal engine, the two must be synchronized and events sent back and forth over network connections of some kind.

There have also been previous Max externals created to allow one to use Lisp in Max in various ways. Brad Garton authored MaxLispJ, an external that provides a Common Lisp interpreter (Armed Bear Common Lisp) embedded through the Java runtime that can be used within Max via the **mxj** object [5]. MOZ'Lib, by Julien Vincenot, takes another approach, in which the Max external acts as a proxy to an externally running Steel Bank Common Lisp process, a similar approach to that used by Cycling '74 for their **Node for Max** object (for

Node.JS/ECMA6), thus enabling the outside process to execute long-running work without blocking Max [14].

In contrast to these, S4M embeds a light-weight Scheme interpreter directly in a Max C external with no intermediate or external runtime necessary. This is made possible by the suitability of Scheme for the creation of small, portable, and self-contained interpreters. The s7 interpreter is implemented entirely in ANSI C, is statically linked during compilation of S4M, and operates entirely in whatever thread it is run in by the C host. This means multiple instantiations of the interpreter are possible, and each **s4m** object has direct C-level access to the Max C API through foreign function interface calls, with the results of said calls being exactly the same as if the functions were called in C, save execution time. The ramification of this is that S4M is unique among Max Lisp projects in being able to operate within the native Max scheduling system, enabling highly accurate timing and retaining deterministic control flow within a patcher. (i.e., There is no hidden networking layer with communication to an outside process, which makes operations fundamentally asynchronous even if this is hidden from the user visually.) This provides the user with opportunities to use Scheme for tasks beyond what is possible with pre-existing options. For example, we are in effect able to say "at a certain time, read some shared state and update an internal engine parameter accordingly". In the example below, a Scheme function is scheduled that will, in 4 bars time, read a rehearsal mark from the table **mark-table**, advancing the global transport to that position. As the table is a shared data object, other Max objects (or patchers) could write to this rehearsal mark table without awareness of S4M.



```
(delay-t (bbu->ticks '(4 0 0))
  (let ((bar (table-ref 'mark-table 4)))
    (lambda()
      (transport-seek bar 0 0))))
```

Fig. 4. S4M patch that moves the transport in 4 bars time to a mark read from a table

3 IMPLEMENTATION OF THE SCHEME FOR MAX EXTERNAL

This section provides a high-level overview of the implementation of the **s4m** object provided by the Scheme For Max external.

The s7 interpreter is implemented in a single C file (with header file), and is statically linked with the C code of the **s4m** object. The s7 foreign function interface enables one to define new Scheme functions in C for calling into C from Scheme, and to call any Scheme code from C. The interpreter is instantiated in the **s4m** object's constructor (**s4m_new** as per the Max naming conventions), and a reference to it is stored in the **s4m** object's data structure. Additionally, during initialization the external creates a Scheme-side variable that holds a pointer to the instantiated **s4m** object itself. Thus both C functions that are called from Scheme and C functions called in response to Max messages have access to both the **s4m** Max object and its associated s7 interpreter. There is always one and only one s7 interpreter per **s4m** object instance, and as s7 is thread safe, many **s4m** objects can be created in a Max patch, including in both the low and high priority threads. (Max runs both a high-priority "scheduler" thread and a low-priority "UI" thread for event/message flow. Each **s4m** object runs in only one of these, chosen via an optional constructor argument, and incoming messages from the opposite thread are deferred or promoted accordingly.)

Scheme-to-C calls are accomplished by defining a C function, and binding it to a Scheme function name in the initialization routine. The function must take as arguments a pointer to the s7 interpreter instance and an **s7_pointer** that is a reference to a Scheme list of the arguments used in the Scheme call. In the body of the function, C functions from the s7 API are used to get individual arguments from this list, the work is done for the function (which may include additional calls back into Scheme or Max API calls), and finally, an **s7_pointer** is returned, which becomes the return value of the function in Scheme.

C-to-Scheme calls are handled similarly: the C function (normally a method of the **s4m** object that runs in response to a Max message) uses the **s4m** object's reference to its s7 interpreter and the s7 API to build Scheme argument lists and call into Scheme, getting back an **s7_pointer** that is then converted to one or more Max atoms. In both cases, helper functions that convert Max atoms to s7 objects and vice versa are used, implemented as **max_atom_to_s7_obj** and **s7_obj_to_max_atom** respectfully. The code example below shows sample functions for both directions.

```
// a C function that is called from Scheme as (post . args)
// it logs to the Max console through the Max API "post" function
static s7_pointer s7_post(s7_scheme *s7, s7_pointer args) {
    // all s7 functions have this form, args is a list, s7_car(args) is the first arg, etc
    char *msg = s7_string( s7_car(args) );
    post("s4m: %s", msg);
    return s7_nil(s7);
}

// a C function that is called from Max (by a clock) and calls into Scheme
void s4m_clock_callback(void *arg){
    // use the clock info structure to get the delayed function handle
    t_s4m_clock_callback *ccb = (t_s4m_clock_callback *) arg;
    t_s4m *x = &(ccb->obj);
    t_symbol handle = *ccb->handle;

    // create an argument list for calling into Scheme
    // x is the s4m object, x->s7 the interpreter reference
    s7_pointer *s7_args = s7_nil(x->s7);
    s7_args = s7_cons(x->s7, s7_make_symbol(x->s7, handle.s_name), s7_args);

    // call into Scheme
    s4m_s7_call(x, s7_name_to_value(x->s7, "s4m-execute-callback"), s7_args);

    // ... clean up trimmed...
}
```

When an **s4m** object is created in the patcher, it can optionally be given the filename of a Scheme file as an argument. If this is given, the file is found by searching the Max search path, and is then loaded on instantiation using the standard Scheme **load** function. Editing this argument in the patcher box always resets the interpreter, as it forces Max to recreate the **s4m** object entirely. Additionally, the **reset** message can be sent to the **s4m** object, and this results in the interpreter being destroyed and recreated, reloading the argument file, without destroying the Max object instantiation.

To evaluate Scheme code dynamically from the Max patcher, two input facilities are available. If an s4m object receives a Max message consisting of the message selector **eval-string** and a single symbol atom, this is taken as a request to evaluate the symbol argument as Scheme code. Thus in Max, the user can build messages with Scheme syntax or receive code as strings over the network, and by using Max objects to convert this into one single quoted symbol, prepended with the selector symbol **eval-string**, the code can be evaluated dynamically. This enables users to add run-time code that is possibly long (i.e., too long to conveniently type into a Max **message-box**) by sending it from a text editor or command line utility to Max over the local network. In figure 4, the left-hand side shows code being sent from a **text** object, prepared as an **eval-string** message, and sent to the **s4m** object. A **udpreceive** object also receives code as strings on network port 7777. (The author uses a Python script triggered from a macro to send blocks of Scheme code from the Vim editor over port 7777.)

The second facility for dynamic code evaluation in S4M is more unusual. In a Max external, there exists a binding option to catch any previously uncaught messages, so that a message with an unrecognized selector can be handled by a generic dispatching method. In S4M, this is used to catch any message that is not already reserved, with any additional arguments passed in as a Max list of atoms. The **s4m** object interprets the message to mean *"evaluate this selector plus its additional atoms as if it is a Scheme list enclosed by parentheses"*. Because both Max and Scheme uses white-space as separators and very little in the way of diacritical syntax, this enables a wide variety of simple Scheme calls to be made with little ceremony. The individual atoms of the Max message are converted to Scheme tokens, assembled into a list, and this list is evaluated. While seemingly simple, this facility is of significant utility to the Max programmer, as combined with the Max message interpolation facility, it enables the user to very easily assemble Max widgets that trigger Scheme functions. Most anything that can be expressed in a single s-expression without inner nesting can be used. In figure 4, we see a **number-box** outputting to a **message-box** using Max's variable interpolation (the **$1**). Clicking the **bang** above the **number-box** or changing the number in the box will result in a message of **set-volume X** being sent to the **s4m** object, which will then call into Scheme with the code **(set-volume X)**.

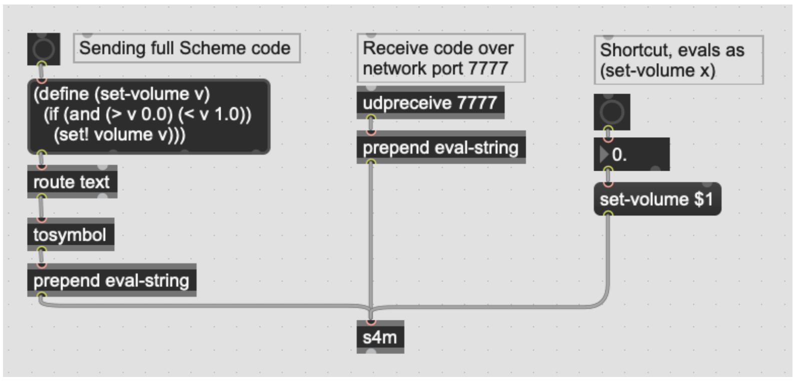

```
(define (set-volume v)
  (if (and (> v 0.0) (< v 1.0))
    (set! volume v)))
```

Fig. 5. S4M patch with 3 ways of sending Scheme code to the interpreter

In addition to both of these, the **s4m** object implements a **read** message, where sending **read filename** to the object will result in the full filename being found on the Max search path and used in a call to the Scheme **load** function, without resetting the s7 interpreter.

Taking the above together, we see that users have a wide variety of code input options: they can change the main file in the **s4m** box if they want a rebooted interpreter, they can send **read** messages if they want to load files without losing interpreter state, and they can send code itself either as strings or as Scheme one-liners implemented as Max messages. The result is that it is straightforward for users to keep parts of their Scheme program active and running while working on other parts that are being redefined on the fly (perhaps while music plays!), a workflow that is very convenient to algorithmic musicians, but impractical with existing Max options. Of particular note, this enables the user to conveniently create new functions and schedule them for future evaluation, as discussed in the next section.

4 EVENT SCHEDULING IN COMPUTER MUSIC AND MAX

Music is fundamentally a temporal art form - there may be music with static pitch or static amplitude, but absent rhythm, there is no music. Thus a core problem in computer music programming environments is that of providing a flexible means of triggering events in time. Further, in platforms intended to support both live and algorithmic musical interaction, a viable solution must enable the performer to interact with scheduled events easily, both programmatically and gesturally.

One of the major advantages of Max compared to other computer music platforms is the ease with which the user can create interactive environments in which the performer's actions change musical parameters in real-time in complex ways, enabling musically rich performances. It is, for example, trivial to add GUI elements to change musical parameters, and only slightly less trivial to add handlers for MIDI input, so that users can connect to their programs physical devices such as piano-style keyboards, and mixing board knobs, faders, and buttons.

For performers and composers exploring the intersections of live performance and algorithmic composition, one is ideally able to use gestural inputs not just to affect what is happening *now* (as with a traditional musical instrument), but also what will happen *in the future*. For example, a performer may have two physical dials, and may wish to schedule an event in the future that will use parameters derived from these dials, but may wish one parameter to be used as the dial is now (at the time of event dispatch) and the other to be used as the performer will have the dial at the scheduled time. Further, the absolute time of the event may not even be known at the time of scheduling, as could be the case if the event time was specified in musical terms (i.e., on the down-beat of the next 8 bar section) and the tempo might be changed in real-time.

The author proposes that this problem is one not well solved in the Max environment prior to Scheme For Max. Max does provide facilities to delay Max messages by some amount of time. In the context of visual patcher programming, users can schedule a Max message (number, symbol, or list of both) for future processing by sending it through a **pipe** object. The amount of time by which it is delayed is specified as an argument to the **pipe** object, and can be expressed in milliseconds or in a tempo relative format with the actual time determined by the Max master transport tempo. The **pipe** object's delay time can also be set dynamically by sending a numerical message to the right inlet. Any messages sent to the left inlet will be passed out the outlet(s) after the specified time.

While moving events into the future with the **pipe** object is simple to program in the patcher and functions adequately, it has several limitations. The most immediately noticeable is that it also splits list messages into individual elements, sending them out individual outlets. If the user wishes to delay a complex event with many parameters, something easily expressed in list format, it requires the user to specify how many atoms will be in an incoming list message in advance and to reassemble the message manually. If the length of the list is unknown (i.e., the event uses an arbitrary number of parameters), the solution is more complex and requires an exterior storage mechanism to be used to allow the outgoing message to re-fetch the original list from another object after delay. Using the **pipe** object for delaying complex events is thus cumbersome.

A larger problem is that of how to capture gestural values for use in the delayed events. Max's visual patching language is fundamentally modelled similarly to a modular synthesizer - there is one instance in memory of each visual object, and messages can only pass through them one at a time. Creating visual programs where execution of delayed events will trigger cascades of messages through objects that are also being used in the interim becomes notoriously complex and there exists no straightforward facility to express *"make a new container for this variable as it is now so that we can fetch it later"*. A naive solution would use a visual patcher object to store the variable's state at trigger time for use later, however this only works if there is only one scheduled event - adding more scheduled events requires adding an additional storage objects per concurrently scheduled event, and as each object in the visual patcher represents an object instantiation rather than a class, this rapidly becomes baroque. This "programming-with-instances" paradigm is very convenient when we want to ensure there is only ever one instance of a given function, such as in emulation of a hardware synthesizer, where each oscillator represents one "always-on" physical device that is playing regardless of whether one can hear it. However, it is difficult in a polyphonic time context such as the creation and eventual playback of some arbitrary number of delayed events.

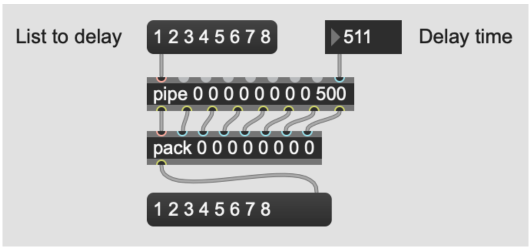


Fig. 6. Delaying a list message with the pipe object requires reassembling the list with the pack object

It should also be mentioned that there exists within Max a facility for creating new instances of Max sub-patches on demand, designed with the express purpose of overcoming this barrier. One can use the **poly** object to create polyphonic patches. However, this involves complex patching, so we will not examine this further here, save to comment that this again becomes very cumbersome when dealing with a potentially large and unknown number of events, each with arbitrary numbers of parameters.

In the C API however, Max does provide a highly accurate facility for delaying the execution of a C callback function, through the Max **clock** API. Scheme for Max builds on this to provide an accurate, flexible, and high-level means for scheduling future events, without the problems described.

5 COMPARISON WITH THE MAX JAVASCRIPT OBJECT

We can see that the visual patching language of Max is not well suited to implementing our use case of a performer wanting to schedule large numbers of events with arbitrary numbers of parameters and gestural inputs meant variously for present and future use. However, Max does provide an alternative programming solution through its **js** object, which embeds a JavaScript (ECMA5) interpreter in the Max environment. The user is able to load JavaScript code from a file given as an argument to the object, and this code can interact with the Max environment and with object inlets and outlets, similar to how one can do so in C code in a Max external. This does allow one to create functions and variables, and does have a facility for delayed execution through the JavaScript **Task** object. Thus technically it could be used to solve our use case. In many respects the **js** object is functionally similar to the **s4m** object, and in fact, the desire to work with Max this way, combined with dissatisfaction with the limitations of the Max JavaScript implementation were the reasons for the development of S4M.

The **js** object in Max is not an ideal solution to our problem for several reasons. The most serious is that it is limited to running only in the Max low-priority GUI thread [1]. As this thread is also used for Max file i/o and graphic redrawing, the result is that latency is potentially high and indeterminate, meaning musical timing accuracy is unpredictable and often poor. The human ear is very sensitive to time, with delays of tens of milliseconds enough to sound like errors in playback of highly rhythmic music. This means that it takes little other activity in the GUI thread to delay our scheduled events enough to be audibly incorrect. For some purposes, this is acceptable, but for the creation of accurate musical sequencers or algorithmic music engines, the **js** object is unreliable.

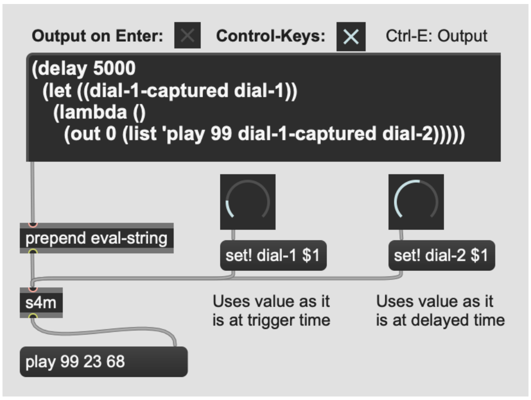
Secondly, the **js** object provides no convenient facilities for loading new code during playback such as was previously described for S4M. The object reads in a single source file at instantiation time. One could technically add dynamic code evaluation by wrapping code to be evaluated in strings and passing this message to a JavaScript function that uses the JavaScript **eval** function, much as S4M does with the **eval-string** message. However, this pattern is not common and well-supported in the JavaScript milieu the way it is in Lisp languages, as it poses a serious security risk in a web development context, which is the use case driving JavaScript linguistic evolution, literature, and tooling.

Finally, the **js** object gives us JavaScript's implementation of anonymous functions and closures, which while usable, are verbose in terms of syntax, and require diacritical syntax that is not easily used in Max messages. When compared to the syntax of Scheme, JavaScript is thus impractical for generating code in Max messages.

This is of particular significance to algorithmic performers, for example in the live-coding scene, who might want to not just schedule a pre-written function, but create a new function and schedule it while a piece plays.

6 SCHEDULING EVENTS IN SCHEME FOR MAX

Given the above, we can see that the problem of elegantly implementing a delayed event system that can differentiate between current and future values for variables derived from real-time input gestures is well suited to solutions using Scheme. Scheme is notable for the conciseness with which one can create an anonymous function and store a reference to this function (taking along its environment) in some data store to be retrieved at an arbitrary point in time. Further, it is straightforward to differentiate in the function between variables that should be used with their value as they are at function definition time versus as they are at eventual evaluation time. The underlying host system is required only to implement the ability to execute a callback at some time in the future, so long as the callback has a means to retrieve some reference to the function stored. Scheme For Max brings this facility to the Max environment.



```
(delay 5000
  (let ((dial-1-captured dial-1))
    (lambda ()
      (out 0 (list 'play 99 dial-1-captured dial-2)))))
```

Fig. 7. Delaying a mixed list message, with dial values used with present and future values

6.1 Clock Implementation in Max

To discuss the implementation of scheduled functions in S4M, we must first examine briefly the implementation of Max externals in C and the aforementioned facilities for delaying functions.

At a high level, a Max external must implement the following:

- A data structure to hold state used by the object
- A class building function used to create the class in C
- An instance constructor function called when objects are added to a patch
- Any methods that will be bound to messages as event handlers

A sample of code for a minimal external is shown below, for an object called **mynum**. The object holds an integer as state, updates the integer on receipt of an **int** message, and posts the value to the console on receipt of a **bang** message.

```
// data structure of instance fields for our class
typedef struct _mynum {
    t_object obj;      // obligatory member for the mynum instance
    long value;        // state variable for the integer
} t_mynum;

// global pointer to our class definition that is setup in ext_main()
static t_class *mynum_class;

// ext_main, the obligatory setup function that builds the mynum class
void ext_main(void *r){
    t_class *c;
    c = class_new("mynum", (method)mynum_new, (method)NULL, sizeof(t_mynum), 0L, 0);
    // bind handlers for the messages we want to be able to receive
    class_addmethod(c, (method)mynum_int, "int", A_LONG, 0);
    class_addmethod(c, (method)mynum_bang, "bang", NULL, 0);
    class_register(CLASS_BOX, c);
    mynum_class = c;
}

// constructor for our object
// the value returned by the below gets stored in the obj field of our t_mynum struct
void *mynum_new(){
    // pointers to the object defined are traditionally called "x"
    t_mynum *x = (t_mynum *)object_alloc(mynum_class);
    x->value = 0;
    return x;
}

// a method handler for int messages that updates the internal state
void mynum_int(t_mynum *x, long n){
    x->value = n;
}

// a method handler for bang messages, post the internal int to console
void mynum_bang(t_mynum *x){
    post("value_is_%ld",x->value);
}
```

We can see that the pattern for adding functionality to our class is to add state variables to the **t_mynum** structure, and add methods as functions expecting a pointer to our object as the first argument, traditionally named "x". In the example above, a **bang** message causes our object to run, with the side effect of posting the stored value to the console. A more realistic example would likely output the stored value, but this adds more code and is not necessary for our demonstration of scheduling.

Let us imagine instead that the **bang** message should delay the activity (posting to the console) by 1000 ms. We can use the Max **clock** facility for high-accuracy delay as it accepts floats for the ms time value. (Note that the actual delay time will be offset to the nearest signal vector boundary, determined by the users "Signal Vector Size" audio setting, but subsequent clock calls adjust for this, maintaining long term temporal accuracy at sample rate resolution.)

We will create a **clock** object, which takes as arguments a void pointer and a function reference, with the void pointer normally used to hold a reference to our instantiated object (x). The standard method for doing this in Max (taken from the SDK documentation) is to add a **clock** object to our classes data structure, and to add the act of starting the clock to the **bang** handler.

```
// typical Max clock use

typedef struct _myint {
    t_object  obj;          // member for the actual instance
    long      value;        // state variable for the integer
    void      *clock;       // will hold clock pointer
} t_myint;

// update the constructor to make a clock
void *myint_new(){
    t_myint *x = (t_myint *)object_alloc(myint_class);
    x->value = 0;
    // create a clock bound to the myint_callback method
    x->clock = clock_new(x, (method)myint_callback);
    return x;
}

// update the bang handler to start the clock
void myint_bang(t_myint *x){
  clock_fdelay(x->clock, 1000);
}

// the callback that will be triggered by the clock
void myint_callback(t_myint x){
    post("in_the_future,_the_value_is_%ld",x->value);
}
```

From the above we can see that, while accurate, the clock functionality is limited - the callback must be a single-arity function expecting a pointer, and this is normally to the object.

6.2 Implementation in Scheme For Max

Scheme For Max builds on the clock facility provided by the Max API to allow scheduling Scheme functions. The delayed functions are limited to zero-arity signatures, but as creation of lambda functions in Scheme is trivial, this is of no practical significance to user. To the user of S4M, scheduling the execution of a function is simple:

```
; schedule a function for 1000 ms in the future
; it will send the int 99 out outlet 0 of the s4m object
(delay 1000 (lambda()(out 0 99)))
```

In addition to scheduling the function, **delay** also returns a unique symbolic handle representing the scheduled instance, and this can be used to cancel the execution of the scheduled function.

```
; delay and store the handle
(define handle
  (delay 1000 (lambda()(out 0 99))))

; cancel it
(cancel-delay handle)
```

The Scheme implementation of this is straightforward:

- The **delay** function (in Scheme) creates a unique symbolic handle, and stores the function passed to it in a hash-table, keyed by this handle.
- It then calls **s4m-schedule-delay**, which is implemented in C and takes as arguments the delay time and the symbolic handle. It will handle clock creation.
- When the clock callback runs after the time has elapsed, it calls (from C) the Scheme function **s4m-execute-callback**, which uses the handle received as an argument to retrieve the delayed function from the Scheme hash-table and call it.

Cancelling a delay function consists of merely replacing the callback registered in the hash-table with the value **#false**, letting the clock fire harmlessly. The s7 Scheme code for this is shown below. It uses an s7's **gensym** function to create the symbolic handle that is guaranteed to be unique to this instance of the interpreter.

```
; registry of delayed functions, by handle
(define s4m-callback-registry (hash-table))

; function to register a callback by a handle and return handle
(define (s4m-register-callback cb-function)
  (let ((key (gensym)))
    (set! (s4m-callback-registry key) cb-function)
    key))

; fetch a callback from the registry
(define (s4m-get-callback key)
  (let ((cb-function (s4m-callback-registry key)))
    cb-function))

; internal function to get a callback from the registry and run it
; this gets called from C code when the Max clock fires
(define (s4m-execute-callback key)
  ; get the func, note that this might return false if was cancelled
  (let ((cb-fun (s4m-get-callback key)))
    ; de-register the handle
    (set! (s4m-callback-registry key) #f)
    ; if callback retrieval got false, return null, else execute
    (if (eq? #f cb-fun)
        '()
        ; call our cb function, catching any errors here and posting
        (catch #t
          (lambda () (cb-fun)) (lambda err-args (post "ERROR:" err-args))))))

; public function to delay a function by time in ms (int or float)
; returns the gensym callback key, which can be used to cancel it
(define (delay delay-time fun)
  ; register the callback and return the handle
  (let ((cb-handle (s4m-register-callback fun)))
    ; call the C FFI and return the handle
    (s4m-schedule-delay delay-time cb-handle)
    cb-handle))
```

The functions prefixed with **s4m-** are internal; the S4M user need only understand how to call the **delay** function.

The implementation in C is more involved to work around the signature limitations of the clock facilities in Max. When the **s4m-schedule-delay** function is run, receiving the symbolic handle and delay time as arguments, the following occurs:

- A **t_s4m_clock_callback** data structure is dynamically allocated and used to store a reference to the **s4m** object and to the symbolic handle passed from Scheme.
- A Max **clock** is created, passing in a void pointer to clock callback structure and binding the clock to a generic clock callback function in C. The clock's timer is started
- The clock callback struct is also stored in a C hash-table (Max's hashtab implementation), keyed by the handle, so that cancellation or object deletion can use this to find all clocks.
- When the generic clock callback function runs (time has elapsed), it uses the pointer argument to get the delay handle and the instantiated **s4m** object, through which it can also get the s7 interpreter pointer.
- The s7 interpreter and handle are then used to call the Scheme function **s4m-execute-callback,** which will run our delayed Scheme function as previously explained.
- The generic C callback then removes the clock reference from the C hash-table, deletes the clock, and frees the memory allocated for the callback struct.

Additionally, there exists clean up functionality triggered on a **reset** message or **s4m** object destruction which fetches from the hash-table all outstanding clocks, cancels them, and frees the memory allocated. This is not shown as it does not materially change the process used.

```
// the struct for the s4m object, with most elements removed
typedef struct _s4m {
    t_object obj;
    // pointer to the s7 interpreter (initialization of which is not shown)
    s7_scheme *s7;
    // a Max hash table for storing clocks (for reset clean-up)
    t_hashtab *clocks;
} t_s4m;

// the clock callback struct
typedef struct _s4m_clock_callback {
   t_s4m obj;
   t_symbol *handle;
} t_s4m_clock_callback;

// schedule delay FFI definition, called from Scheme as s4m-schedule-delay
static s7_pointer s7_schedule_delay(s7_scheme *s7, s7_pointer args){
    // as this is called from Scheme, we must find x by fetching it
    // from the Scheme variable set in initialization
    t_s4m *x = get_max_obj(s7);

    // get the arguments we need (time and handle) from the s7 args list
    // that represents the arguments passed to s4m-schedule-delay in Scheme
    // first arg is float of time in ms
    double delay_time = s7_real( s7_car(args) );

    // second arg is the symbolic handle
    char *cb_handle_str;
    s7_pointer *s7_cb_handle = s7_cadr(args);
    cb_handle_str = s7_symbol_name(s7_cb_handle);

    // allocate memory for our clock_callback struct and populate
    // NB: this gets cleaned up by the receiver in the clock callback above
    t_s4m_clock_callback *clock_cb_info =
        (t_s4m_clock_callback *)sysmem_newptr(sizeof(t_s4m_clock_callback));
    clock_cb_info->obj = *x;
    clock_cb_info->handle = gensym(cb_handle_str);

    // make a clock, setting our callback info struct as the owner, as a void pointer
    // when the callback method fires, it will retrieve this pointer as an arg
    // and use it to get the handle for calling into scheme
    void *clock = clock_new( (void *)clock_cb_info, (method)s4m_clock_callback);

    // store the clock ref in the s4m clocks hashtab (used to get at them for reset cancelling)
    hashtab_store(x->clocks, gensym(cb_handle_str), clock);

    // schedule it, this is what actually kicks off the timer
    clock_fdelay(clock, delay_time);

    // return the handle to the Scheme caller
    return s7_make_symbol(s7, cb_handle_str);
}

// the generic clock callback, this fires after being scheduled with clock_fdelay
// it gets access to the handle and s4m obj through the clock_callback struct
// this is the C callback that runs for every delayed Scheme function
void s4m_clock_callback(void *arg){
    t_s4m_clock_callback *ccb = (t_s4m_clock_callback *) arg;
    t_s4m *x = &(ccb->obj);
    t_symbol handle = *ccb->handle;

    // call into Scheme with the handle, where Scheme will call the registered delayed function
    // we must build a Scheme list through the FFI to use as the arguments
    s7_pointer *s7_args = s7_nil(x->s7);
    s7_args = s7_cons(x->s7, s7_make_symbol(x->s7, handle.s_name), s7_args);

    // call the Scheme s4m-execute-callback function
    // s4m_s7_call is a simple wrapper around s7's s7_call with error handling and logging
    s4m_s7_call(x, s7_name_to_value(x->s7, "s4m-execute-callback"), s7_args);

    // remove the clock(s) from the clock registry and free the cb struct
    hashtab_delete(x->clocks, &handle);
    // free the memory for the clock callback struct
    sysmem_freeptr(arg);
}

// clean up methods for s4m reset and delete not shown
```

7 LIMITATIONS AND FUTURE IMPROVEMENTS

The scheduling system describe does have several limitations worth noting. The s7 Scheme implementation includes a tracing garbage collector that must run occasionally, pausing other processing [8]. The garbage collector can, however, be paused and run on demand. Tests on the author's system, with the garbage collector forced to run every 100 ms to prevent significant build up, showed the garbage collection taking an average of 1 to 2 ms per pass when run in a substantial application (a 16 track sequencer written entirely in S4M with the computer also generating audio for the tracks with commercial software synthesizers). The audio rendered was recorded and examined examined in a digital audio workstation to check for timing discrepancies. The results of these preliminary tests were that timing was unaffected by the garbage collector so long as Max was configured with a large enough output buffer to allow the collector to run within the latency period of this buffer. Using Max with an output buffer of 256 samples produces an output latency of approximately 6-8 ms (depending on the Max signal vector as well), which was large enough for the collector to run without issue, but is low enough to be acceptable for real time performance. With software synthesis loads at closer to the maximum possible on the author's computer, an output buffer of 512 samples (approx. 11 ms output latency) was necessary to prevent any timing errors. This is encouraging as these buffer sizes are not atypical for commercial music production on digital audio workstations and are widely considered as being in acceptable ranges for real-time music systems [3].

In addition to output latency, there is the factor of internal latency. Max, like many computer music programming environments, generates audio in vectors of samples, with each vector generated from one processing pass, and (normally) control rate passes conducted also a maximum of once per audio vector pass [9]. The size of this signal vector thus has a large effect on performance, with larger vectors requiring significantly less CPU use. As audio rendering of note onsets originating from control messages (the normal scenario) must thus be aligned with signal vector boundaries, this produces delays of note onsets by up to the time period of one vector, a phenomenon known as "timing jitter" in the nomenclature. However, the more recent scheduling facilities in Max, including the clock facilities used by S4M, are implemented to compensate for this jitter, maintaining long term temporal accuracy despite note onset delays. The mechanism by which this is done is unfortunately not publicly documented or available in open-source code, but one can assume that it is some variant of the common pattern whereby scheduled events are aware of when they are supposed to be running ("logical-time"), and if they in turn schedule subsequent events, the time for these subsequent events will be shifted to compensate for the jitter, based on the logical time of the scheduling event, rather than its actual (post-jitter) real time [2]. The author also conducted personal tests of this compensation, comparing both high and low signal vector sizes to produce variable delays in note onsets, and it was determined that so long as the overall output latency is again large enough to allow the GC to run in time, self-scheduling events in a long piece do stay accurate in that jitter does not accumulate over the piece - note onsets stay accurate to within the delay of one signal vector of samples, regardless of the length of the piece of music.

These tests demonstrated that Scheme For Max is usable as a highly accurate timing source for processing-intensive music production so long as some reasonable latency is used, but that at low latencies the garbage collector will interfere with timing by generating audio under-runs. Improving or replacing the s7 garbage collection implementation for lower latency use is a potential area for future work.

A second limitation is that S4M does not implement DSP calculations at this time, running only in the Max event/message thread. Max does have an audio configuration option whereby the event thread and DSP thread are shared in a round-robin execution pattern. It is worth exploring whether the Scheme interpreter could be run productively in the DSP loop in this configuration. While it is unlikely that S4M will be suitable for processing-intensive audio synthesis (requiring filling the whole vector per pass), it is possible that other productive work could be done once per DSP pass. This could be used to create the equivalent in Max of the "control-rate" as it is implemented in Csound, where rather than control rate calculations being triggered solely by events, there is also the facility to create lower resolution continuous signals that can be used to modulate audio signals. This can be useful for automating parameters that sound acceptable at lower sample rates, such as pitch curves, and is an area that will be explored in subsequent work on S4M.

8 CONCLUSION

The Scheme For Max project brings a new way of working to the Max platform, especially with regard to flexible and accurate scheduling of future events. The author believes this will be of significant benefit to those working in the fields of algorithmic music and live coding, and will also be useful to the more general Max user base as a way to script Max in a high level language that runs in the high-priority thread. In the author's experience, working in Scheme within Max has been highly productive, with the language well suited both to expressing musical concepts and to working with the Max platform. The ability to update code during play-back has been an especially productive workflow for building complex sequencing and algorithmic composition tools.

Future plans for the project include a version for the open-source Pure Data platform, similar to Max in many ways and created also by Miller Puckette, the original author of Max. **Scheme For Pure Data** is now in beta release, available as source code, with binary releases planned for July 2021. In addition to this, experimental work on running Scheme for control rate DSP is planned, as well as the creation of portability layers in Scheme to allow one to run the same Scheme code on S4M, S4Pd, or Common Music (which also uses s7). There have been successful ports of s7, Pure Data, and Csound to Web Assembly, thus it seems likely that work from the project will be useable for web audio as well. It is hoped that through these additional run-times, the project can become a valuable addition to the larger computer music community, enabling use in contexts where the commercial Max platform is not practical or financially accessible.

Scheme For Max can be obtained on GitHub, with links to documentation, demonstration videos, and various tutorials linked from the main page.

REFERENCES

[1] Cycling '74. [n.d.]. *Max JS Tutorial 3: JavaScript Tasks, Arguments, and Globals.* https://docs.cycling74.com/max8/tutorials/javascriptchapter03
[2] David P. Anderson and Ron Kuivila. 1986. Accurately Timed Generation of Discrete Musical Events. *Computer Music Journal* 10, 3 (1986), 48–56. https://doi.org/10.2307/3680259
[3] Eli Brandt and Roger Dannenberg. 1998. Low-Latency Music Software Using Off-The-Shelf Operating Systems. In *International Computer Music Conference.*
[4] Roger B Dannenberg. 1997. The Implementation of Nyquist, A Sound Synthesis Language. *Computer Music Journal* 21, 3 (1997), 71–82.
[5] Brad Garton. 2011. *maxlispj.* Retrieved Jan, 2011 from http://sites.music.columbia.edu/brad/maxlispj/
[6] Victor Lazzarini. 2013. The Development of Computer Music Programming Systems. *Journal of New Music Research* 42, 1 (2013), 97–110. https://doi.org/10.1080/09298215.2013.778890
[7] V.J. Manzo. 2021. *maxobjects.com.* Retrieved Jan 1, 2021 from maxobjects.com
[8] Kjetil Matheussen. 2020. *Re: [CM] S7s GC (CM-dist mailing list archive).* https://www.mail-archive.com/cmdist@ccrma.stanford.edu/msg05595.html
[9] Miller Puckette. 2002. Max at Seventeen. *Computer Music Journal* 26, 4 (Dec. 2002), 44–51. https://doi.org/10.1162/014892602320991365
[10] Charles Roberts and Graham Wakefield. 2018. Tensions and Techniques in Live Coding Performance. In *The Oxford Handbook of Algorithmic Music*, Alex McLean and Roger T. Dean (Eds.). Oxford University Press, New York, NY, Chapter 16, 293.
[11] Bill Schottstaedt. 2021. *s7.* Retrieved June 5, 2021 from https://ccrma.stanford.edu/software/snd/snd/s7.html

[12] Andrew Sorensen and Henry Gardner. 2010. Programming with time: cyber-physical programming with impromptu. *ACM SIGPLAN Notices* 45, 10 (2010), 822–834. https://doi.org/10.1145/1932682.1869526

[13] Heinrich Taube. 2002. Common Music: A Music Composition Language in Common Lisp and CLOS. *Computer Music Journal* 15, 2 (2002), 21–32. https://doi.org/10.2307/3680913

[14] Julien Vincenot. 2017. LISP in Max: Exploratory Computer-Aided Composition in Real-Time. In *International Computer Music Conference*, Vol. 2017. Michigan Publishing, University of Michigan Library, Ann Arbor, MI, 87–92. http://hdl.handle.net/2027/spo.bbp2372.2017.012

[15] Ge Wang. 2017. A History of Programming and Music. In *The Cambridge Companion to Electronic Music*, N. Collins and J. D'Escrivan (Eds.). Cambridge University Press, Chapter 16, 55–85. https://doi.org/10.1017/9781316459874.006

[16] David Zicarelli. 2002. How I Learned to Love a Program That Does Nothing. *Computer Music Journal* 26, 4 (Dec. 2002), 44–51. https://doi.org/10.1162/014892602320991365