Revista Vórtex | Vortex Music Journal | ISSN 2317–9937 | http://vortex.unespar.edu.br/
D.O.I.: https://doi.org/10.33871/23179937.2021.9.2.19

Scheduling and Sequencing Musical Events in Pure Data with Scheme for Pd

Iain C. T. Duncan

University of Victoria | Canada

Abstract: Scheme for Pd is a newly released open-source external for Pure Data, enabling scripting Pd with s7 Scheme Lisp. In this experience report, the author introduces Scheme for Pd and demonstrates its use for scheduling and sequencing musical events in s7 Scheme.

Keywords: Pure Data, Lisp, computer music languages, scheduling

Received on July 1st, 2021. Approved on: November 1st, 2021. Available online on: 30/11/2021. Guest Editor: Dr. Alexandre Torres Porres. Creative Commons CC-BY-NC.

My name is Iain Duncan, and I am a musician, computer programmer, and Music Technology graduate student in Victoria, Canada. In June 2021, I released the first version of Scheme for Pd, an open-source external to enable scripting and live-coding Pure Data (Pd) with Scheme, after working for two years previously on a version of the same for Max. Scheme for Pd, like it's cousin Scheme for Max, embeds a Scheme interpreter in the platform as a regular Pd object, allowing one to program Pd in a high-level, dynamic, text-based language. I believe this opens new ways of working with Pd, and this experience report is meant to serve as a brief introduction to the project and a highlight of one particular problem area that I feel it solves elegantly.

By necessity, this article includes code in s7 Scheme. I hope the reader will bear with me and can follow along well enough to get the gist of things. Interested readers are encouraged to read my s7 Scheme tutorial[1] and the resources linked from the project's page on GitHub[2]. The s4pd external itself can also be downloaded from the GitHub page on the "releases" page. (It is not yet available in the Deken package manager but will be in the future).

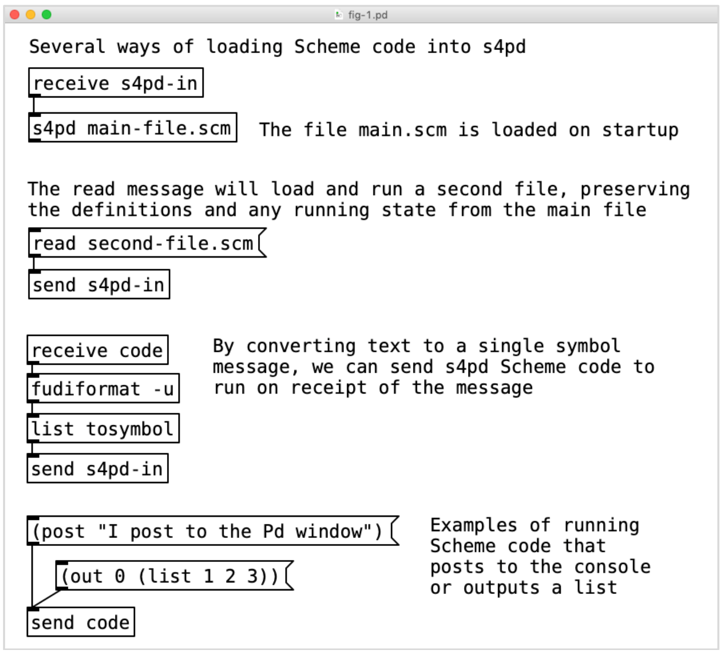
I've long felt that our tools matter in a very serious way. While it's a cliche that it's the artist not the tools, it is also true that the tools shape the artist. Our tools affect what is hard, what is easy, what is stimulating, and what can be done lazily - and I believe this has an unavoidable effect on our choices. In computer music, I think it's safe to say that all the major platforms can now be used to make all different kinds of music, but I think it's also true that in some platforms certain paths are harder, and thus certain decisions are easier or harder to make. In light of this, I've spent a great deal of time over the last 20 years exploring and building tools that help me make the kinds of decisions I want to make when making music. Pure Data and Max have figure prominently in my tool kit, but at various points there have been others too, including text-oriented computer music languages such as Csound, and general-purpose languages such as Python, C/C++, Clojure, and most recently Scheme. Scheme for Pd is my attempt to bring some of what I liked most from these explorations to the rich and productive environment of Pure Data.

In brief, Scheme for Pd enables one to script Pure Data in s7 Scheme, a Scheme Lisp dialect

[1] The author's s7 Scheme Tutorial: https://iainctduncan.github.io/learn-scheme-for-max/introduction.html
[2] Project Page: https://github.com/iainctduncan/scheme-for-pd

developed by Bill Schottstaedt at CCRMA, and used also in projects such as the Snd audio editor and the Common Music 3 algorithmic composition platform, from which Scheme for Pd draws much inspiration (SCHOTTSTAEDT, William, 2021). The **s4pd** object is able to load Scheme files, similar to how the **js** object uses JavaScript in Max, but is also able to dynamically evaluate Scheme received as real-time messages. This includes code coming from message boxes, or over the network via OSC (Open Sound Control), or loaded dynamically from additional files during run-time, and this can even be done while the existing code is running. Figure 1 shows an example of several ways of running code with Scheme for Pd.

FIGURE 1 - s4pd loading Scheme code



```
Several ways of loading Scheme code into s4pd
receive s4pd-in
s4pd main-file.scm   The file main.scm is loaded on startup

The read message will load and run a second file, preserving
the definitions and any running state from the main file
read second-file.scm
send s4pd-in

receive code        By converting text to a single symbol
fudiformat -u       message, we can send s4pd Scheme code to
list tosymbol       run on receipt of the message
send s4pd-in

(post "I post to the Pd window")    Examples of running
                                    Scheme code that
   (out 0 (list 1 2 3))             posts to the console
                                    or outputs a list
send code
```

Source: the author.

The **s4pd** object includes facilities for converting between Pd and Scheme data types, and allows one to easily send and receive standard Pd messages, as well as write and read from array tables in order to share data between objects. This enables one to use Scheme to interact with any message driven features of a Pd patch. (The **s4pd** object does not generate audio or run in Pd DSP loop.) I've found being able to automate my patches with a flexible, concise, and dynamic programming language to be of significant benefit. In this experience report, I want to briefly highlight one area in particular that I've found has opened doors for how I use Pd and Max.

The patcher paradigm is wonderfully productive for certain kinds of work, but there are others for which it is a bit of a fight. Many years ago, I implemented an entire live show rig purely in Csound, including virtual analogue mono-synthesizers. To replicate a mono-synth properly, one must ensure that, whether or not a note is playing, the oscillators are always running, and the envelopes and ramps are always doing their thing, picking up properly from where they left off. Forcing Csound to act like a set of single-instance monophonic modules requires quite a bit of hacking around its fundamentally polyphonic model. This would have been much easier in Pure Data, though I didn't know it at the time! For those who haven't used Csound, a note event, whether from a pre-written score or sent to the system in real time, becomes a new instantiation of an instrument, with this instantiation lasting in memory as long as the note is active. One therefore has unlimited polyphony - or at least up to the processing limits of the machine - but also each note is unaware and unaffected by neighboring notes, without additional programming effort. To relate this to more mainstream programming terminology, our instruments are factories (what might be classes), and notes from the score become instantiated objects. Every new note brings a new isolated instrument into existence, much the way a digital delay unit creates a completely new instance of a sound on each echo (LAZZARINI et. al, 2016, p. 31-32).

Pure Data and Max share a fundamentally different model - in Pd we program by dragging an object to the canvas, and that visual object on the canvas represents an *instantiation* of an object, which was created by factory functions in the object's C source code. The factories are pre-compiled, and in essence, we program the patcher with the *instances*. This means that the visual representation of our program on the canvas is generally also an accurate representation of active state containers - one object on the canvas corresponds to one object's collection of state data (PUCKETTE, 2002, p. 44-51). Now returning to my old mono-synth setup, this would have been very natural to model in Pd as there are no special steps required to make an object act like an analogue module, with one oscillator that can only be at one particular place in its cycle at a time. The Pure Data paradigm is also very natural for creating tools for interactivity. A dial, a fader, a key on a keyboard, in all cases there is *a single thing* we manipulate, and tracing the flow of data from this thing through a suite of other things is obvious. Having a visual widget that stands for an instantiation of an object with state makes intuitive sense.

However, there is another area where this paradigm becomes more difficult, and where the Csound model shines, namely scheduling future events. Perhaps we want to experiment with algorithms that create clusters of events, with events variously spawning subsequent events based on choices in each parent event. Perhaps an event stream has a half-life, where it has a 50 percent chance of spawning a new instance sometime in the future, and so on up to a threshold. This sort of thing is certainly possible in the patcher paradigm, using facilities such as the pipe object and spawning new canvases, but it's not particularly easy - it feels like we are fighting against the design model. In this case, programming the *factories*, and letting them create their instances at playback time, is arguably a more natural way to solve the problem (LAZZARINI et. al, 2016, p.115). I believe this is a use case where Scheme for Pd brings something new and valuable to Pd. Scheduling events as dynamically created Scheme functions is simple, effective, and flexible. In this paper we will look at examples of such patches, culminating in a midi delay object that schedules delayed notes across many midi channels while looking up modulation data for these events from Pd arrays.

Scheme for Pd provides delayed scheduling with a very straightforward facility: we simply pass a time and a reference to a zero-argument function to the **delay** function (built in to s4pd), and under the hood this is turned into a Pure Data clock (at the C programming level), resulting in a callback into Scheme to evaluate our stored function at the appropriate time. In a less flexible language, this model might seem overly simplistic. But in Scheme, making a zero-argument function to pass to **delay** is so simple and flexible that this is enough to build sophisticated and complex scheduling and sequencing systems. Let's look at some examples.

To send a list message out of an **s4pd** object's outlets, we use the **out** function. The below will output a list of three numbers out the first outlet:

```
(out 0 (list 1 2 3))
```

The **out** function takes two arguments: the outlet (from zero up), and that which we would like to send out, which can be a number, symbol, or list thereof. For our clock facility to work, we require a zero-argument function (a constraint of how Pd clocks work in C code). We can create this with Scheme's **lambda** form. Lambda returns a function definition, and we'll use this to make an anonymous wrapper function around our call to **out**, saving the resulting function reference to the symbol name **my-callback**:

```
(define my-callback
    (lambda() (out 0 (list 1 2 3))))
```

This means that my-callback is a single function that can be called as `(my-callback)`, and which will output our 3 item list through outlet zero. And now we can use this to schedule execution of our callback for 1000 milliseconds from now:

```
(delay 1000 my-callback)
```

The same expressed more succinctly might dispense with the name for the function, calling lambda to create an anonymous function, and passing the anonymous function directly as the second argument to **delay:**

```
(delay 1000 (lambda() (out 0 (list 1 2 3))))
```

As you can see, the use of the Scheme **lambda** form makes creation of temporary zero-argument functions from more complex functions straightforward.

Now let's do something more interesting by having an event schedule repetitions of itself recursively. To begin, let's look at a recursive loop in Scheme. Recursion has a reputation for being confusing, but in Scheme, it is not so intimidating, the language lends itself well to expressing loops this way. Here we output numbers 4 to 0:

```
; define our recursive counter function
(define counter
    (lambda(count)
        ; first we output the current value of count
         (out 0 count)
        ; then if our count argument is not yet zero, call counter
        ; with count decremented as the argument
        (if (> count 0)
          (counter (- count 1)))))
; now call it for the first time and let it count-down
(counter 4)
```

We can use a similar facility for *temporal recursion*, in which, instead of calling itself right way, a function *schedules* the next iteration of itself using the **delay** function. This is a technique used heavily in algorithmic composition tools such as Common Music, Extempore, and Nyquist, but also even in Csound.

```
; define our self-scheduling event
; it repeats every 1000 milliseconds, outputting a float of half the value
(define event-echo
    (lambda (count amplitude)
        (out 0 amplitude)
        (if (> count 0)
            ; to schedule, we wrap our function in a zero-argument function
            (delay 1000
                (lambda()
                    (event-echo (- count 1) (* 0.5 amplitude)))))))
; start off the process, passing initial count and amplitude
(event-echo 4 1.0)
```

In the above example, our s4pd object will send a float value out outlet 0 five times, with the values 1.0, 0.5, 0.25, 0.125, and 0.0625 respectively.

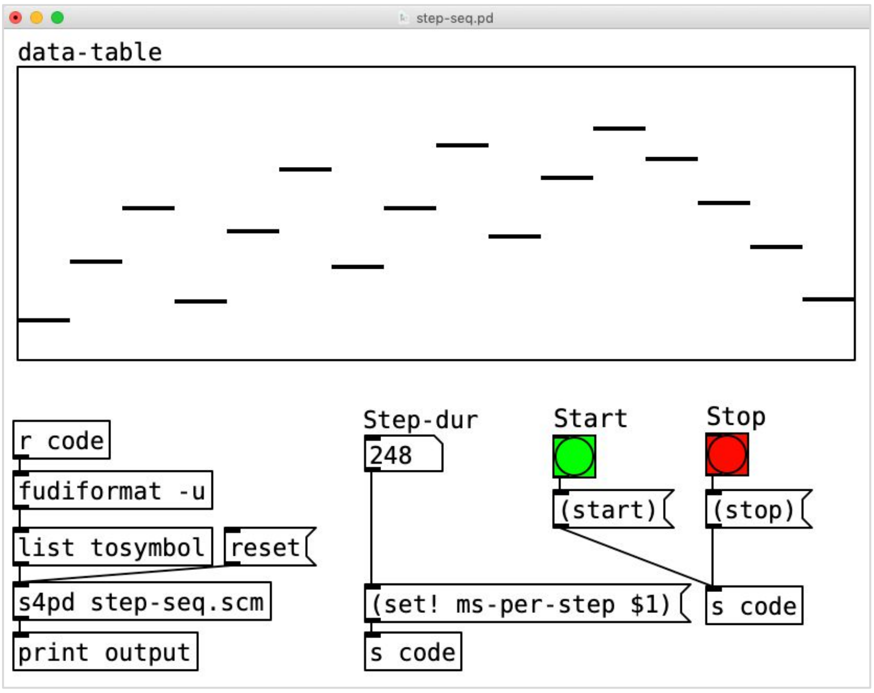
This may seem a lot to digest to a new Scheme programmer, but you can see that once one is familiar with the idioms of the language, this kind of algorithm can be expressed succinctly. And we are not using much of the language to achieve this, just function and variable definitions, branching, and function calls. A running function thus becomes an *instance* of an event, with a flexible way to schedule further instances of itself.

Let's look at an example of using this self-scheduling technique to implement a step sequencer. In the example below, we have the simplest possible step-sequencer, in which our s4pd object reads from a Pd array, sending the value read out its outlet. There is a main function, **run-step**, which executes on each step and checks the **playing** variable to see if it should schedule its next iteration after updating the **current-step** counter. Data is read from a named table, using the current-step variable as a table index, and we have a couple of helper functions, **start** and **stop**, to kick-off the process of the self-scheduling **run-step** functions or to unset the **playing** flag respectfully. (In this example we will use the shorter form syntax for defining a Scheme function, sometimes known as "defun" syntax from its similarity to Common Lisp's **defun** form.)

```
; global variables that will be used for sequencer state
(define num-steps 8)
(define ms-per-step 200)
(define current-step 0)
(define playing #f)

; define our step function, named 'run-step' and taking no arguments
(define (run-step)
  ; get data by reading from the array 'data-table' and send it out
  (out 0 (table-read 'data-table current-step))
  ; increment step for next round, rolling to zero at num-steps
  (if (< current-step (- num-steps 1))
      (set! current-step (+ 1 current-step))
      ; else rollover to zero
      (set! current-step 0))
  ; and schedule the next step, unless playing set to #false
  (if playing
      (delay ms-per-step run-step)))

; functions to start and stop
(define (start)
    (set! playing #t)
    (run-step))

(define (stop)
    (set! playing #f))
```

The Pd patch for the above contains an array for the data table, our s4pd instance, and some widgets that send code to set the step duration and call the start and stop functions.

FIGURE 2 - A simple s4pd step sequencer



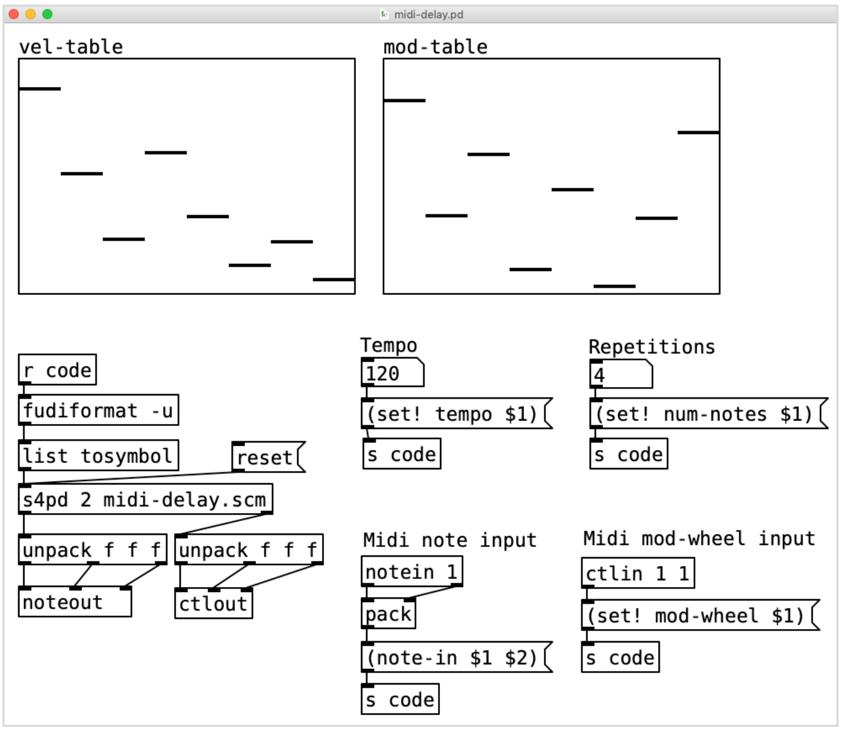
```
r code
fudiformat -u
list tosymbol
reset
s4pd step-seq.scm
print output
248
(set! ms-per-step $1)
s code
(start)
(stop)
s code
```

Source: the author.

You can see that extending this would be quite straightforward, and we can interact with it by adding further Pd widgets to send messages that set Scheme variables or call Scheme functions. In my own work, I use this pattern to build sequencers that can alter their data algorithmically as they play, and to allow me to alter the sequencer engines themselves by redefining their internal functions as they are playing.

Finally, I would like to end with a more substantial example to illustrate how much functionality can fit in small amount of code. We'll make a midi delay unit, but with some special features. In this example, delayed notes will fan out over separate channels to allow the synthesis settings of each note repetition to be different (the first note 1 will be on channel 1, the second to channel 2, and so on). We will also have two data tables, one for velocity scaling factors and one for mod-wheel scaling factors. Our delayed midi notes will use these, using the repetition number as the table index, and determining the output velocity and mod-wheel output for the repetition's midi channel by using the retrieved values as scaling factors. To do this, we will also need to make our own equivalent of the Pd **makenote** object, as the built in one does not handle interleaving channels (I believe we would need one per channel to work properly). For illustration, we'll do this in Scheme and have a midi output function that automatically sends a note off after our duration has elapsed. (This is admittedly a naïve implementation, but to keep the example readable it will do for our purposes.)

FIGURE 3 – A MIDI delay patch



```
r code
fudiformat -u
list tosymbol
reset
s4pd 2 midi-delay.scm
unpack f f f
unpack f f f
noteout
ctlout

120
(set! tempo $1)
s code

4
(set! num-notes $1)
s code

notein 1
pack
(note-in $1 $2)
s code

ctlin 1 1
(set! mod-wheel $1)
s code
```

Source: the author.

In addition to our tables and our s4pd object, we have made widgets for changing the tempo and the number of repetitions (1 to 8 inclusive). And we have two widgets that receive midi input and turn it into Scheme functions calls: one for note input, and one for modulation wheel input. The Scheme code for this builds on the techniques demonstrated in the previous examples. We receive midi input in our **note-in** function, which in turns calls the scheduling function **play-note (**ignoring note-off messages). The **play-note** function uses the data in the tables and the current triggering velocity and mod-wheel setting to calculate the velocity and mod-wheel settings to use for actual note output. These are sent as lists out outlet 0 and 1 respectively, and we use standard Pd objects to convert this to midi output. After sending the output, our **play-note** function schedules its next execution, with the delay time based on the **tempo** variable (it is locked to 8th note delays for simplicity).

```
; global vars that get updated from the patch
(define tempo 120)
(define num-notes 5)
(define mod-wheel 64)

; handler function for midi note input
(define (note-in note-num vel)
    (if (> vel 0) ; ignore note-offs
        (play-note 0 note-num vel)))

; function to play a note and schedule recursive repeats
(define (play-note count note-num vel)
    (let* (; get velocity and mod-wheel scaling from arrays
           (vel-out (* vel (table-read 'vel-table count)))
           (mod-out (* mod-wheel (table-read 'mod-table count)))
           ; calculate 8th note delay, dotted 16th duration
           (del-ms    (* 500 (/ 60 tempo)))
           (dur-ms    (* 0.75 del-ms)))
        ; send mod wheel (cc 1) out outlet 1
        (out 1 (list mod-out 1 (+ 1 count)))
        ;send the note values by calling midi-note
        (midi-note dur-ms count note-num vel-out)
        ; schedule the next iteration with adjusted values
        (if (< count num-notes)
            (delay del-ms (lambda()
                  ; on next call to play-note, count is incremented
                  (play-note (+ 1 count) note-num vel))))))

; midi output function that schedules a note-off after dur-ms
(define (midi-note dur-ms channel note-num vel)
    ; send a note on out outlet 0, adjust channel to 1+
    (out 0 (list note-num vel (+ 1 channel)))
    ; after dur-ms, send out a note-off (midi velocity 0)
    (delay dur-ms (lambda()
        (out 0 (list note-num 0 (+ 1 channel))))))
```

I hope this overview has piqued interest by demonstrating how Scheme for Pd can be used for various kinds of scripting and automation, and especially for manipulating timed events elegantly in the Pure Data environment. For those interested in exploring Scheme for Pd further, the project is available on its GitHub page, with binary releases for Mac and Windows, and source code available for all platforms.[3]

You will also find links on the GitHub page to beginner friendly tutorial material, along with demonstration videos and sample code. I am happy to answer any questions from readers and would love to hear how Scheme for Pd helps you with your projects.

REFERENCES

LAZZARINI, Victor et. al, *Csound A Sound and Music Computer System.* Switzerland: Springer International Publishing, 2016.

PUCKETTE, Miller. Max at Seventeen. *Computer Music Journal,* vol 26 issue 4, p.44-51, 2002 dec.

PURE DATA. Version 0.51-1. Developed by Miller Puckette. Available at <https://puredata.info> Accessed on: 21 nov. 2021

S7 SCHEME. Developed by William Schottstaedt. Available at: <https://ccrma.stanford.edu/software/snd/snd/s7.html> Accessed on: 21 nov. 2021

ABOUT THE AUTHOR

Iain C.T. Duncan is a musician and software developer in Victoria BC, and a student in the MMus in Music Technology at the University of Victoria, Canada, studying under Dr Andrew Schloss and Dr George Tzanetakis. He first began using computer music programming platforms with Csound and Max in 1996, and has since worked in numerous languages and platforms. He is the author of Scheme for Pd (beta) and Scheme for Max, and also plays jazz saxophone. ORCID: https://orcid.org/0000-0002-3185-854X. E-mail: iduncan@uvic.ca

[3] https://github.com/iainctduncan/scheme-for-pd