Scheduling Musical Events as Procedures with Scheme for Max

IAIN C.T. DUNCAN, University of Victoria, Canada

We are confident that scheduling musical events as Scheme procedures using Max/MSP and Scheme for Max is the bomb.

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

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

ACM Reference Format:
Iain C.T. Duncan. 2021. Scheduling Musical Events as Procedures with Scheme for Max. 1, 1 (June 2021), 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Here is the introduction about why Max and Scheme are so cool.

2 CONCLUSION

Well that was cool, I wonder how I should conclude such a thing???

3 BACKGROUND - PROGRAMMING MUSIC IN MAX/MSP

3.1 The Max Environment

Cycling 74's Max (previously named Max/MSP) is the predominant visual programming environment for interactive multi-media 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. Created originally by Miller Puckette in year while working at IRCAM, Max was created so that composers of interactive computer music could change their pieces without the assistance of a programmer. Max programs ("patches" in the Max nomenclature) are created by placing visual objects on a canvas ("the patcher"), and connecting them graphically with visual "patch cords". 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 in turn send on messages to further objects. Patch execution can be initiated by various forms of real-time input, such as keyboard or mouse events, MIDI input, and networking events, as well as by scheduled events through objects such as the metronome. Messages are represented visually as lists of Max atoms, which may be symbols, integers, or floating point numbers, and there exists a Max object for visually displaying and altering messages, the message-box. Execution follows a depth first and right-to-left order, enabling the programmer to deterministically control the execution flow through the visual layout of

Author's address: Iain C.T. Duncan, iainctduncan@gmail.com, University of Victoria, , Victoria, BC, Canada,

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 before moving left, rather than spawning two concurrent threads of execution.) When patching, messages can be inspected by sending them to a print object (to print to console), to a message-box object (to update the visual object with the message contents), or though a built in debugger ("probing").

In addition to the event-based message execution model, Max also supports a stream-based digital audio execution model, originally provided separately as MSP, but now included in all Max versions. MSP normally runs in a separate thread from the event and message based Max operations, and uses a a separate class of objects that pass constant streams of digital audio to each other through differently coloured patch cords. MSP objects may additionally receive messages for controlling paramaters. As S4M executes only the event/message level and does not implement DSP operations, MSP is not discussed further here.

3.2 Max Message and Object Implementation

Internally, Max messages are implemented as pairs of data entities consisting of a symbol (required) and an array of Max atom structures (optional). Each atom structure contains a member for the atom type and another for its value, and the type may be any of: integer, float, or symbol. The first element of a message pair is always a symbol, and is called the message selector. Note that while this first symbolic element is always present at the C level, in the visual patcher, the selector may be implicit and normally hidden from the user. For example, let us imagine one creates a message-box object with an integer as its visual contents. The user can send this message to connected receiving objects by clicking on the visual message-box, or by sending to the message-box any message (A process referred to as banging, from the bang object, and essentially meaning "run now"). Let us further imagine the receiving object is another message-box, and we send this message to its right hand inlet, who's job it is to update the message-box's visual contents (its internal state). The actual message created by the action of clicking the first message-box will consist of a hidden selector of the symbol "int", followed by a one item array where the first item is an integer atom. The second box will receive the int message, running the handler bound to the symbol "int", which expects one interger atom as an argument. It will update its internal state, and display a single integer, hiding the implicit symbol "int". Were we to do the same with the first message box-containing "int 99", the results would be identical - we are merely making explicit the integer selector. Similarly, message boxes containing multiple integers in fact send the message selector "list", followed by an array of integers.

XXX: do we even need that shit about implicit and explicit messages??

In order for Max object to do anything, it must thus implemement handler methods bound to the selectors for whatever messages it should respond to. Thus under normal circumstances, in the message/event context, Max execution in an object begins when a selector handler runs, and this handler execution triggers side effects. These are most commonly changes to the object's own state and the creation and output of messages through the object's outlets, but can also be calls to engine functions to trigger actions or update various data structures, or creation and dispatch of messages to objects outside of the visual patching graph.

XXX: fix this, wrap it up at right level of detail

One side effect of interest to us is that a message may be placed on a scheduler queue, to be consumed

3.3 Max externals

While Max itself is a commercial, closed source tool, it includes a software development kit for extending Max by writing a Max “external”. An external is a compiled class that acts as a prototype for new objects that can be used in patcher programming, just as one uses the built in objects. Externals are developed in C or C++ in an object oriented manner: the older C API using data structures and pointers to simulate class-based programming and the more recent C++ API using C++ classes. A typical external will implement a class that provides some (optional) object state for instantiated objects, and methods for receiving and sending Max messages through the object’s inlets and outlets. When the user creates a visual object to the patcher, an instance of the external’s class is instantiated in memory. Thus visual programming in the Max patcher manipulates instantiations of objects, whereas programming at the external level in C is concerned with the creation of classes acting as object prototypes, and the creation and reception of messages, but rarely involves programmatically instantiating objects from Max object classes.

Objects are not limited to interacting with Max through messages passing in to 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.

Extending Max through externals has been possible since very early versions of Max, and thus a rich library of open-source 3rd party extensions exist, of which Scheme For Max is one.

3.4 Lisp based music platforms

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 rich history of Lisp in music, ranging from to current systems such as Common Music and Extempore, both of which use similar Scheme implementations to Scheme for Max, and Common Lisp based systems such as Incudine and Slippery Chicken. Nor is Scheme for Max unique in providing a Lisp based extension facility to a commercial music platform: the author first (unknowingly!) programmed in Lisp in the early 1990’s as part of the Cakewalk MIDI sequencer, which came with the Cakewalk Application Language (CAL) extension system.

However, Scheme for Max is unique in bringing a fully-fledged Scheme interpreter to the Max environment, which is arguably the most mature and extensive of the truly programmable music platforms in wide use. Systems such as Common Music, Incudine, and Extempore all act as the outermost host, with events originating from them destined for other systems for translation into audio. While these systems are also capable of receiving gestural input, they must, in essence, ”be the boss”: they provide the master timing clock and act as the principal engine around 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.

Where Scheme for Max differs is that it provides an interpreter embedded into Max, with facilities to interact with Max in every way that any other Max external can, including direct interaction with the Max scheduler system and transport mechanism. This means, for example, that in Scheme for Max we are able to say ”do this Max-internal thing based on other internal Max state at such a time, as determined by the Max scheduler”.

3.5 The importance of scheduling events in computer music and Max's implementation of such.

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 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 delayed events satisfactorally, both programmatically and gesturally.

One of the major advantages of Max compared to other computer music platforms is the ease of creating 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, ideally one would be 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 triggering if the event time was specified in musical terms (i.e. the down-beat of the next 8 bar section) and the tempo is variable.

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 the future by sending it to a 'pipe' object. The amount of time by which it is delayed is specified as an argument of (or message 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 time can be set dynamically by sending a numerical message to the right inlet of pipe, and messages sent to the left inlet will then be passed out the pipe outlet(s) after the specified time.

While moving events in to the future with the pipe object functions adequately, it has several limitations. The most immediately noticeable is that it also splits list input 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 thus requires the user to specify how many atoms will be in an incoming list message in advance and to reasssemble the message manually. If the length of the list is unknown, the solution is more complex and requires an exterior storage mechanism be used to allow the outgoing pipe message to refetch the original list from another object after delay. Using pipe for delaying complex operations is thus cumbersome at best.

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 for 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 straight-forward 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 pather 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 additional storage objects for each event. This programming-with-instances paradigm is very convenient when we want to ensure there is only one of a given function, such as in the design of a classical monosynth, but difficult in a poly-phonic time context such as the creation and eventual playback of some unknown number of delayed events.

4 COMPARISON WITH THE MAX JAVASCRIPT OBJECT

We can see that the visual patching language of Max is not well suited to implementing this use case. Max does provide an alternative programming solution through its 'js' object, which embeds a JavaScript (ECMA5) interpreter in the Max environment. The js object is functionally very similar to S4M, and in fact, the desire to work with Max this way, combined with disatisfaction with the limitations of the js object was the impetus for development of S4M. The user is able to load JavaScript code in the js object 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 in C code used to develop a Max external. This does allow one to create functions and variables, and technicallycould be used to solve our use case described above.

However, the js object is not an ideal solution for several reasons. The most serious is that it is limited to running only in the Max low-priority GUI thread. As this thread is used for file i/o and graphic redrawing, the result is that latency is high and timing accuracy is unpredictable and often poor. The human ear is very sensitive to time, and perceives gaps of 20ms as discreet events. Delays of tens of ms are more than 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 sequencers or algorithmic playback devices, a js based solution is not usable.

Secondly, the js object provides no convenient facilities for loading new code during playback. The object reads in a single source file at instantiation time. One could technically add dynamic code evaluation to the JS object by wrapping code to be evaluated in strings and passing this message to a JavaScript function that uses the JavaScript "eval" function. However, compared to dynamic code evaluation facilities in Scheme, this is again cumbersome. This is of interest to algorithmic performers, for example in the "live-coding" oevre, who might want to create a new function and schedule while a piece plays.

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

In contrast, Scheme syntax lends itself well to expression in Max messages. Both Max and Scheme use whitespace as token separators, and the punctuation marks that have significance in Max messages ($,   ) are easily avoided in Scheme. (Note that this restriction applies only to Scheme code built in Max messages, collisions do not happen at all for code loaded from files or strings.)

The advantages of being able to build Scheme code in Max messages themselves is that it becomes very simple to add functional input to a Scheme system, even while it is playing. Max provides a facility whereby messages can act as templates, with the $ sign used to indicate placeholders. This allows one to attach an input element, for example a dial, to a message-box and have the box output the results of template interpolation whenver ever the dial is moved. The example below shows a Max and S4M patch in which a Max dial widget updates triggers a message of "set! foobar dial-setting", which is then passed on to an S4M object for evaluation. One of the convenience facilities provided by S4M is that of treating unhandled messages as Scheme expressions as if they were wrapped in parantheses, enabling easy creation of Scheme one-liners in Max messages. Thus the patch below is all that is required to capture input from a dial widget and update a variable in the Scheme environment.

While the above is convenient to the visual programmer, S4M is not limited to one-line code. Should we wish to handle code in Max messages with complete Scheme syntax, we must only convert the message to a quoted symbol and pass the resulting string to the S4M object as the body of an 'eval-string' message.

5 SCHEME FOR MAX AND SCHEDULING EVENTS

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 representing input gestures is one perfectly suited to expression in S4M. Scheme is notable for the ease and 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 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, where the callback is able to retrieve some indicator of the correct function stored. Scheme for Max brings this facility to the Max environment, enabling users to dynamically define functions during playback of a piece, and schedule these in musical terms for execution in the future.

5.1 Clock Implementation in Max

To discuss the implementation of scheduled functions in S4m, we must first show the implementation of objects in C and the facilities for delaying functions.

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

- a data structure to hold state used by the object
- a main function used to create the class in C
- a new function that acts as a constructor for instances
- any methods that will be bound to messages as event handlers

A sample of a minimal external that holds an integer and updates the integer on receipt of an 'int' message is below.

```
// simplified example from the Max SDK documentation

// data structure for our class, contains instance data
typedef struct _simp {
    t_object s_obj;     // member for the actual instance
    long s_value;       // state variable for the integer
} t_simp;

// global pointer to our class definition that is setup in ext_main()
```

```
static t_class *s_simp_class;

// ext_main, the setup function that builds the s_simp class
void ext_main(void *r){
    t_class *c;
    c = class_new("simp", (method)simp_new, (method)NULL, sizeof(t_simp), 0L, 0);
    // bind handlers for the messages we want to be able to receive
    class_addmethod(c, (method)simp_int, "int", A_LONG, 0);
    class_addmethod(c, (method)simp_bang, "bang", 0);
    class_register(CLASS_BOX, c);
    s_simp_class = c;
}

// constructor for our object
// the value returned gets stored in the s_obj field of our t_simp struct
void *simp_new(){
    t_simp *x = (t_simp *)object_alloc(s_simp_class);
    x->s_value = 0;
    return x;
}

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

// a method handler for bang messages, post the internal in to console
void simp_bang(t_simp *x){
    post("value is %ld",x->s_value);
}
```

We can see that the pattern for adding functionality to our class is to add state variables to the t_simp structure, and add methods that are functions expecting a pointer to our object as the first argument, normally named 'x'. In the example above, a 'bang' message causes our object to run, with the side effect of posting the stored value. 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) by 1000ms. We can use the Max clock facility for high-accuracy delay. (An older, and more flexible, facility exists as well but as it is less accurate temporally, it is not used in S4M)

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

```
// Max standard clock use
// data structure for our class, contains instance data
typedef struct _simp {
    t_object  m_obj;          // member for the actual instance
    long      m_value;        // state variable for the integer
    void      *m_clock;       // will hold clock pointer
} t_simp;

/ update the constructor to make a clock
void *simp_new(){
    t_simp *x = (t_simp *)object_alloc(s_simp_class);
    x->s_value = 0;
    // create a clock bound to the simp_callback method
    x->m_clock = clock_new(x, (method)simp_callback);
    return x;
}

// update the bang handler to start the clock
void simp_bang(t_simp *x){
  clock_fdelay(clock, 1000);
}

// our callback that will run from the clock
void simp_callback(t_simp x){
    post("value is %ld",x->s_value);
}
```

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

5.2 Implementation in Scheme for Max

Scheme for Max builds on the clock facility provided by the Max API to allow scheduling Scheme procedures. These 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 procedure is simple:

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

In addition to scheduling the function, the delay function also returns a unique symbolic handle that can be used to cancel a delayed function before it runs.

```
; 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 procecure creates a unique symbolic handle, and stores the procedure in a hash-table, keyed by the handle
- it then calls an internal Scheme procedure, 's4m-schedule-delay', which is implemented in C using the s7 foreign function interface (FFI) and takes as arguments the delay time and the symbolic handle
- s4m-schedule-delay creates a clock, with a callback that will receive the handle
- when the clock callback runs, it uses the FFI to call the scheme procedure s4m-execute-callback, which also takes the symbolic handle as an argument.
- s4m-execute-callback, defined in Scheme, receives the handle, retrieves the procedure from the hash-table (deleting it in the process), and runs the procedure.

Cancelling a delay function consists of merely replacing the callback registered in the hash-table with the value #false, in effect letting the clock fire harmlessly.

The s7 Scheme code for this is show 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.

```
; internal registry of callbacks
(define s4m-callback-registry (hash-table))

; procedure 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 runs
(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 ms (int or float)
; returns the gensym callback key, which can be used to cancel it
(define (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 time cb-handle)
    cb-handle))
```

The implementation in the C code for the S4M external is more involvedto 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 data structure (t_s4m_clock_callback) is dynamically allocated and used to store a reference to the s4m object and to the handle
- a Max clock is created, passing in a void pointer to this structure and a generic clock callback, and the clock's timer is started
- the clock_callback struct is also stored in a C hashtable (Max's hashtab implementation), keyed by the handle, so that cancellation or object deletion can find all clocks
- when the generic clock callback function runs, it uses the pointer argument get the the delay handle and the instatiated s4m object, through which it can get the s7 interpreter pointer
- the interpreter and handle are then used to call the Scheme function s4m-execute-callback, running our Scheme procedure
- the callback then removes the clock reference from the C hashtable, deletes the clock, and frees the memory allocated for the callback info struct.

```
// 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
    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, registered in object setup to be run from Scheme as s4m-schedule-
static s7_pointer s7_schedule_delay(s7_scheme *s7, s7_pointer args){
    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(size
    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 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 cancel
    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
// gets access to the handle and s4m obj through the clock_callback struct
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 delay
    // 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 l
    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);
}
```

```
// the s4m object's free method, called by Max internals on object deletion
// it must cancel and free any outstanding allocated clocks
void s4m_free(t_s4m *x){
    //  ... various other cleanups trimmed ...

    // clocks must all be stopped and deleted.
    // iterate through the hashtab to stop them
    hashtab_funall(x->clocks, (method) s4m_cancel_clock_entry, x);
    // this will free all the clocks
    object_free(x->clocks);
}

// iterator used above to cancel a clock
void s4m_cancel_clock_entry(t_hashtab_entry *e, void *arg){
    if (e->key && e->value) {
        clock_unset(e->value);
    }
}
```

6 CONCLUSION

Well that was cool, I wonder how I should conclude such a thing???

A RESEARCH METHODS

A.1 Part One

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

A.2 Part Two

Etiam commodo feugiat nisl pulvinar pellentesque.

B ONLINE RESOURCES

Nam id fermentum dui.