Extending Snd with Eval-C and Snd-Rt

Kjetil Matheussen
Norwegian network for Technology, Acoustic and Music. (NOTAM).
Nedre Gate 5,
0551 OSLO,
Norway,
k.s.matheussen@notam02.no

Abstract

This paper presents two domain specific programming environments made for extending Snd: Eval-C and Snd-Rt. Eval-C and Snd-Rt add another set of pragmatic and practical solutions for working with music programming in Snd, from very low level to very high level, and both in realtime and non-realtime.

The software synthesizer program "San-Dysth" is also presented, since its programmed using Snd-Rt. And finally, Snd/Pd is presented, which makes Snd's and Snd-Rt's features available as a Pure Data (Pd) external.

Keywords

Music programming, Snd, Programming languages, Lisp, Pure Data

Software overview

Name	Description
Snd	A sound editor written by Bill Schottstaedt.
Snd-ls	A distribution of Snd.
CLM	A music synthesis and signal processing package (Common Lisp Music) written by Bill Schottstaedth.
Snd-Rt	A realtime music programming environment running inside Snd.
Eval-C	The programming language large parts of Snd-Rt is written in.
San-Dysth	A softsynth written in Snd-Rt.
Snd/Pd	Snd running as a Pd external.
Guile	An R5RS Scheme interpreter. A community effort based on the SCM Scheme implementation by Aubrey Jaffer.

Language features

	Guile	Snd-Rt	Eval-C
Interpreted	x		
Compiled		x	x
Closures	x		
Nested functions	x	x	
Functions as arguments	x	x	x
Lisp lowlevel macros	x	x	x
Dynamic typing	x		
Static typing		x	x
Type inference		x	
Heap allocation	x		x
Stack allocation		x	x
Garbage collection	x		
Runtime error checking	x	x[1]	
Can segfault			x
Speed	1	9	10
Hard realtime	0	10	5
Soft realtime	5	10	9
C library access	8	8	10
Vector read access	10	10	5
Vector write access	10	0	5
List read access	10	10	5
List write access	10	0	5
Vct read access	10	10	5
Vct write access	10	10	5
Signal bus access	0	10	2
CLM access	10	8	4

1 Snd, Eval-C and Snd-Rt

Eval-C and Snd-Rt are two domain specific programming environments made for extending Snd. Eval-C and Snd-Rt are implemented as lisp low level macros running inside the Guile Scheme implementation, and Guile again runs inside the Snd soundeditor. This means that the three languages, Scheme, Snd-Rt and Eval-C, can be written and evaluated in the same source to provide an integrated and interactive environment. By combining Guile, Snd-Rt and Eval-C, programmers get three levels of versatility and execution speed which can all be used in

[1] Can be turned off, with the consequence of making Snd-Rt segfaultable and somewhat faster.

the same source files. Snd/Pd, presented later, is programmed using all three languages.

2 Snd

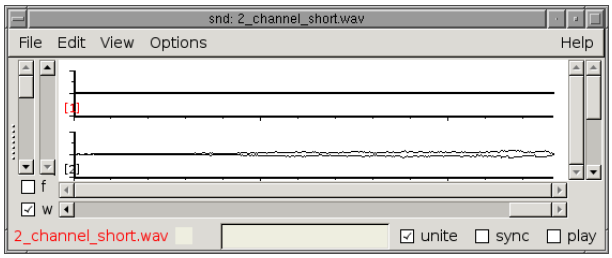

Figure 1: Snd 9.4, vanilla

Snd is an extendable sound editor made by Bill Schottstaedt. Here is a quote from the Snd manual:

> Snd is a highly customizable, extensible program. I've tried to bring out to the extension language nearly every portion of Snd, both the signal-processing functions and much of the user interface. You can, for example, add your own menu choices, editing operations, or graphing alternatives. Nearly everything in Snd can be set in an initialization file, loaded at any time from a text file of program code, or specified in a saved state file. It can also be set via inter-process communication or from stdin from any other program (CLM and Emacs in particular), embedded in a keyboard macro, or typed in the listener.

Virtually everything in Snd can be customized via extension languages. Currently, Snd supports the following implementations as being extension language: Guile (Scheme), Gauche (Scheme), Ruby (Ruby) and FTH (Forth).

Snd also provides features such as interaction with Common Music (CM) and bindings for CLM, Gtk, Motif, Xlib, OpenGL, etc. Snd runs in Linux, Freebsd, MacOSx, Solaris, Windows and probably many other Unix dialects.

2.1 Snd-ls

Snd-ls[2] is a distribution of Snd.

[2]Snd-ls stands for *Snd-Non Lisper*. I accidentally wrote *ls* instead of *nl* in the beginning of the development

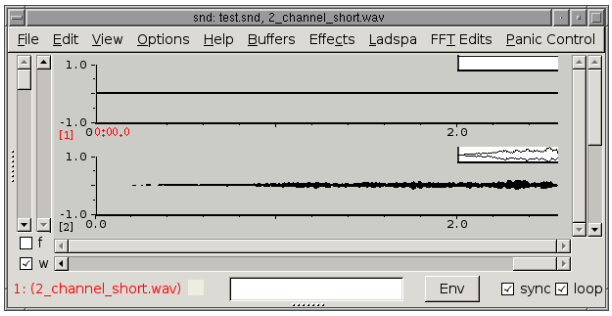

Figure 2: Snd-ls V0.9.8.3

Snd-ls has enabled various features which are not by default enabled in vanilla Snd. Snd-ls is also configured to run Snd-Rt and Eval-C code.

3 Eval-C

Eval-C is a language based on C, but using S-expressions as its syntax format. Using S-expressions makes it possible to mix Scheme and Eval-C in the same source files and easily add advanced syntactic features to the C language such as object oriented programming and evaluation on the fly to support interactive development of programs. Evaluating and re-evaluating code on the fly makes it very convenient to add low-level features to Snd and Snd-Rt since its not necessary to recompile and restart Snd while developing.

3.1 Some features provided by Eval-C

- Integration of C-code from within Lisp. Scheme and C-code is mixed in the same source, and the two languages can interact directly.
- Direct access to C libraries
- Lisp-macros.
- Generate, compile, link, run and redefine C-code on the fly in Emacs or other Lisp programming environments
- Same efficiency as C
- Same low-level features as C
- Extra macros to support features such as arrays, shared structures, classes, and automatic wrapping of C library functions.

3.2 Hello world!

```
(eval-c ""
        (run-now
         (printf (string "Hello world!\\n"))))
```

First argument to **eval-c** contains extra options sent to the C compiler.

The rest of the arguments are Eval-C code.

Strings containing pure C-code can be used anywhere in Eval-C code, which is very handy when writing macros and providing C features. For example, the above example can also be written like this:

```
(eval-c ""
        (run-now
         "printf(\"Hello World!\\n\")"))
```

To make an actual null-terminated string for C, the *string* special form must be used instead, as in the first version.

3.3 The Fibonacci function

The simplest Fibonacci function:

```
(define-c <int> (fib ((<int> n))
                     (if (< n 2)
                         (return n)
                         (return
                           (+ (fib (- n 1))
                              (fib (- n 2)))))))
(fib 30)
=> 832040
```

define-c is a macro transforming its arguments into a call to **eval-c**. The function *fib* above is transformed into:

```
(eval-c ""
  (public
   (<int> fib
          (lambda ((<int> n))
            (if (< n 2)
                (return n)
                (return (+ (fib (- n 1))
                           (fib (- n 2)))))))))
```

When evaluating the above block of code, the **eval-c** macro uses gcc or icc to compile and link the code.

3.4 Structures

The macro **define-ec-struct** creates structures which are shared between Eval-C, C and Guile[3]:

```
(define-ec-struct <struct_name>
  <int> one
  <float-*> twos
  <char-*> three
  <SCM> scm)
```

By evaluating the above code, a structure called *struct_name* will be available from guile. It can be used like this:

```
(define test (<struct_name> :one 1
                            :twos '(2)
                            :three "three"))
(-> test one)    => 1

(-> test one 90)
(-> test one)    => 90

(-> test twos)   => (2.0)

(-> test twos '(4 5 6))
(-> test twos)   => (4.0 5.0 6.0)

(-> test three)  => "three"

(-> test three "four")
(-> test three)  => "four"

(-> test scm)    => #f

(-> test scm (lambda (x) x))
(-> test scm)    => #<procedure #f ((x) x)>

(-> test get-size) => 16

(-> test get-c-object)
 => ("A_POINTER" 147502592)

(-> test destructor) ;;free it
```

Modifiers and accessors are available for all slots. Known slot-types are *SCM*[4], *char*, *int*, *float*, *double* and *char **, and arrays of those. C modifiers such as *unsigned* are treated appropriately. There are also functions to add new types which can be mapped to the known types. Slots with unknown types keep their names, but are treated as pointers by Eval-C.

Ec-structs are extremely convenient when communicating with C programs since they contain actual C structures.

The C object must unfortunately be freed manually[5]. This was in the example done by evaluating *(-> test destructor)*.

[3] And hopefully Snd-Rt sometime into to future as well

[4] *SCM* is the Guile datatype, which can hold any type of Scheme object. However, care must in some situations be taken using *SCM* types in ec-structres, since Guile's garbage collector doesn't scan ec-structure objects. The objects are also kept in the Scheme object though (ie. the variable *test* in the example), so usually this should not be a problem.

[5] Either using a general garbage collector for C, such as the one made by Hans Boehm, or use a Guile SMOB around the structures, would solve this problem. However, freeing ec-structures manually hasn't been a big issue yet.

4 Snd-Rt

Snd-Rt, or the RT extension for Snd, consists of two parts:

The RT Engine - An engine for doing realtime signal processing.

The RT Compiler - A compiler for a Scheme-like programming language to generate realtime-safe code understood by the RT Engine.

The entire realtime engine and large parts of the compiler for Snd-Rt is written in Eval-C. Nothing in the realtime engine or the RT language is written directly in C. Wrapper functions for libjack and liblrdf are created automatically using Eval-C as well.

4.1 The Fibonacci function

```
(define-rt2 (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(fib 30)
=> 832040.0
```

4.2 A sound

```
(let ((osc (make-oscil)))
  (<rt-play> 0 3
             (lambda ()
               (out (* 0.8
                       (oscil osc))))))
```

To run the example above, paste the 5 lines into the terminal Snd-ls was started. After that, a 440Hz pure tone should be played for exactly three seconds.

4.3 The RT Engine

The main purpose of the RT Engine is to receive <*realtime*> objects created by the RT Compiler, and provide ways to control exactly when to run those objects.

The API for doing this is hidden from the user, and is accessed by instead calling methods provided by the <*realtime*> class.

4.3.1 RT Engine features

- Hard Realtime safe
- Jack and Pd driver
- Realtime scheduler with a properly made priority queue to ensure a stable CPU load when adding and removing realtime objects to and from the engine.
- <*realtime*> objects are started and stopped in realtime without sound glitches.
- Buses can be rerouted in realtime without sound glitches.
- Frame-accurate timing and scheduling
- Protection mechanism to avoid locking up the computer if using too much CPU.
- Controlled entirely from Scheme using Guile, and, to a certain degree, from inside the realtime objects themselves

4.4 The RT Compiler

The RT compiler works by first translating the Scheme-like input-code into Eval-C code, and then calling **eval-c** to create machine code.

The RT compiler translates in several stages, which includes macro expansion, lambda lifting, name mangling, type inference, syntax checking, etc.

4.4.1 RT Compiler features

- Compilation of simple Lisp functions into machine code.
- Generation of hard realtime safe code.[6]
- The compiled code should ideally be close to C in efficiency.

4.5 The RT programming language

The Snd-Rt programming language (RT) is coincidentally quite similar to the Lisp dialect PreScheme [1]. But while PreScheme is a general programming language standing on its own feet, RT is a domain specific language specialized for music, sound and realtime use running inside the Guile Scheme interpreter.

The RT language is designed to be a pragmatic language. It is not always as fast as C, and it is not as practical and feature rich as Scheme or Common Lisp, but it fits nicely in between and can provide almost-as-fast-as-C of usually-high-level-enough code.

The RT language is mostly imperative, but since it also has fairly good support for higher level functions, the programmer is often not restricted to one programming paradigm, but has the choice between coding semi-functionally or imperatively.

[6] Unless the RT code contains non-realtime safe functions such as *printf*.

4.5.1 RT programming language features

- Ladspa
- Alsa midi
- Lisp Macros
- Automatic read access to Guile numbers, lists, vectors and numbers. (seamless integration)
- Automatic read- and write access to VCTs[7]
- Guile has read- and write access to all of RT's variables
- Most of CLM [2] is provided as well as many functions provided by Snd, Sndlib and Guile.
- Static typing and type inference
- The types for numeric variables can be specified for optimization
- Runtime error checking and crash insurance
- Buses
- Realtime-safe ring-buffers between Guile and RT
- Lisp structures. Accessible from, and using the same syntax as, both Guile and RT.[8]
- Access to external libraries can easily be added on the fly by using Eval-C

4.6 No control rate

A liberating feature of CLM and Snd-Rt is that there is no control rate, and that every sound frame is automatically exposed to the programmer. Pd, for example, generates 64 frames at each control rate tick, while CLM generates only one.

This creates a simpler interface for the user, and removes the need to create generators in a different language, usually C or C++. The downside is that its harder to make the signal processing algorithms run fast this way, especially for interpreters.

Exposing every sound frame to the programmer is also more powerful. For example, a new type of filter can perhaps be made in Pd by using the z~ external or similar mechanisms, but chances are that it is easier to write an external in C instead. However, in CLM, doing this kind of operation is both an easy and a straight forward task for a programmer. An example of such a generator is the San Dysth softsynth described in the next section.

[7]Vct is Snd's data type for holding arrays of sound data. It is similar in functionality to Scheme's vector type, but vct's can only contain sounds, and are optimized to do so.

[8]Not the same kind of structure as ec-structures

4.7 Why a compiler?

Most other music programming languages, like PD, CSound and the SuperCollider server, are interpreters since the speed of the language controlling the MusicV graph doesn't matter that much.

In those languages, the generators process blocks of audio samples, and most of the CPU is therefore spent inside those generators and not in the language controlling the graph.

However, in CLM, most of the time is often spent in the language controlling the graph, and therefore the difference in speed between interpreting and running compiled code is quite significant.[9] This is especially important when generating sound in realtime, since we don't want to hear interruptions in the sound.

4.8 Dynamic memory allocation

Some important features are missing from the RT programming language, such as functions to dynamically allocate memory on the heap for creating lists and closures. In practice, these features will probably not very often be used since its more natural to create lists in Guile instead of Snd-Rt, ie. preallocating the required lists. And so far, I have not had any use for closures.

However, since it hasn't been possible to create lists or closures in Snd-Rt so far, I can not be entirely sure its not very useful either.

5 San Dysth

San Dysth is a standalone realtime soft-synth written in Snd using the Snd-Rt language.

San Dysth has controls to generate various kinds of sounds in between white noise and pure tones. It also provides controllers to disturb the generated sound by using a "period counter" to extend the variety of the generated output.

Common usage for San Dysth is organ-like sound, organic-like sound, alien-like sounds, water-like sounds, and various kinds of noise. Noise artists could find this softsynth most useful.

[9]If we assume that compiled code runs faster than interpreted code, which has been a common observation.

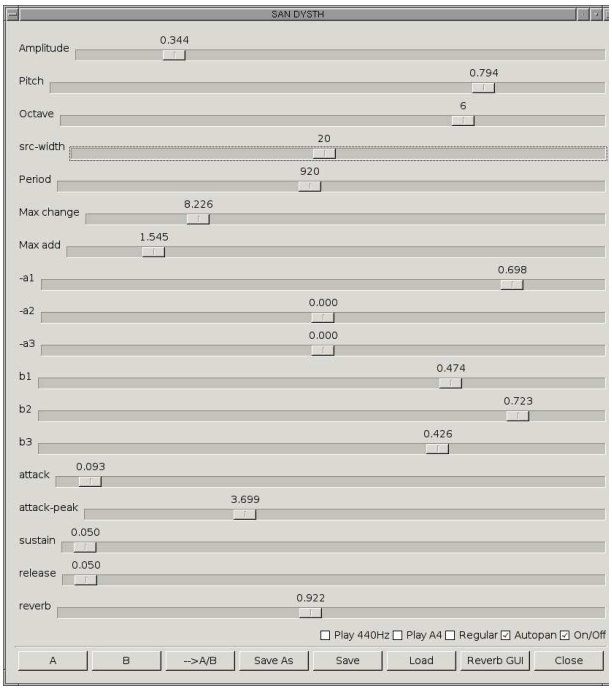

Figure 3: San Dysth

San Dysth's synthesis technique works by using a set of rules which change state for each sample:

```
;; This function provides the main synthesis
;; routine for San-dysth. The function
;; returns one sample per call.
;;
(define-rt (san-dysth-dsp direction)
  (cond ((<= val -1)
         (set! inc-addval #t))
        ((>= val 1)
         (set! inc-addval #f))
        ((> addval max-add-val)
         (set! periodcounter period)
         (set! inc-addval #f))
        ((< addval (- max-add-val))
         (set! periodcounter period)
         (set! inc-addval #t))
        ((= 0 (inc! periodcounter -1))
         (set! periodcounter period)
         (set! inc-addval (not inc-addval))))
  (define drunk-change
    (random max-drunk-change))
  (set! addval
        (filter das-filter
                (if inc-addval
                    drunk-change
                    (- drunk-change))))
  (inc! val addval)))))))))
```

```
;; The following block creates and plays a
;; realtime object. The variables max-add-val,
;; max-drunk-change, period, das-filter, vol and
;; rate are parameters which later can be adjusted
;; by the user. Notice the use of dynamic scoping
;; when Snd-Rt use variables created by Guile.
;;
(let* ((val 0)
       (addval 0)
       (max-add-val 0.01)
       (max-drunk-change 0.00005)
       (period 400)
       (periodcounter period)
       (inc-addval #f)
       (sr (make-src :srate 0 :width 20))
       (rate 10)
       (vol 20)
       (das-filter (make-filter
                    4
                    (vct 1 -0.474 -0.723 -0.426)
                    (vct 1 0.698 0 0))))
  (<rt-play>
    (lambda ()
      (out (* vol
              (src sr rate san-dysth-dsp))))))
```

6 Snd/Pd

Snd can be configured to work as a Pd external.

6.1 FM synthesis example

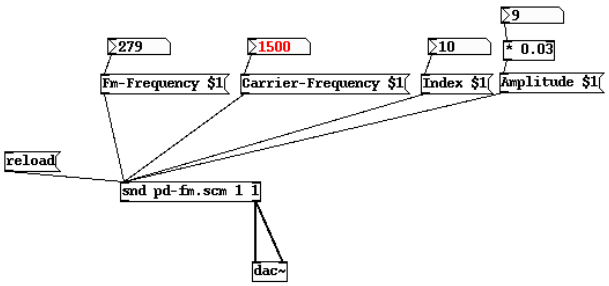

Figure 4: Snd/Pd doing FM synthesis

The file pd-fm.scm used in figure 4 looks like this:

```
(let ((amp 0.6)
      (mc-ratio 2)
      (index 4)
      (freq 300))
  (define fm
    (make-oscil (* freq mc-ratio)
                :initial-phase (/ 3.14159 2.0)))
  (define carrier (make-oscil freq))
  (define fm_index (* (hz->radians freq)
                      mc-ratio
```

```
                    index))
(define instrument
  (<rt-play>
   (lambda ()
     (out 0 (* amp
               (oscil carrier
                      (* fm_index
                         (oscil fm)))))))))
(pd-inlet 0 'mc-ratio
          (lambda (val)
            (set! mc-ratio val)))
(pd-inlet 0 'Fm-Frequency
          (lambda (val)
            (set! (mus-frequency fm)
                  (* mc-ratio val))))
(pd-inlet 0 'Carrier-Frequency
          (lambda (val)
            (set! (mus-frequency carrier)
                  val)))
(pd-inlet 0 'Index
          (lambda (val)
            (set! (-> instrument fm_index)
                  (* (hz->radians freq)
                     mc-ratio
                     val))))
(pd-inlet 0 'Amplitude
          (lambda (val)
            (set! (-> instrument amp)
                  val))))
```

6.2 Threading

When Snd runs as a Pd external, it also runs in its own dedicated thread. This ensures that Guile's garbage collector doesn't interrupt sound processing. Communication between Pd and Snd happens by sending messages back and forth on two ringbuffer. Snd-Rt's signal processing thread, on the other hand, runs directly in Pd's main thread, like all other signal processing code in Pd.

6.3 Interactive development

The Snd external also uses Snd's code to read Scheme code from stdin. This makes it possible to do interactive development inside a Lisp environment like Emacs by starting Pd as a Scheme sub-process. The external also uses Snd's code to properly catch errors and display relevant error messages, making debugging very convenient.

6.4 API for data processing

Below is a brief description of all functions provided for the Snd/Pd interface. A more complete description is available in the Snd manual.

(*pd-inlet inlet-num type func*)
: *func* is a function which is called when the object receives a message to inlet number *inlet-num* of type *type*. Common values for *type* are 'float, 'list and 'bang.

(*pd-outlet outlet-num arg0 arg1 ...*)
: Sends one or more values to outlet *outlet-num*. The arguments can be of any type.

(*pd-bind symbol func*)
: Pd messages sent to *symbol* arrive at the *func* function.

(*pd-unbind symbol*)
: Stop receiving messages sent to *symbol*.

(*pd-send symbol arg0 arg1 ...*)
: Sends one or more values to receivers for *symbol*. *symbol* can either be a scheme symbol or a pd symbol.

(*pd-get-symbol symbol*)
: Returns the pd symbol for the scheme symbol *symbol*. pd-send works faster when a pd symbol is used instead of a scheme symbol.

(*pd-set-destroy-func thunk*)
: *thunk* is called before the object is destroyed or reloaded.

6.5 API for signal processing

The API for doing signal processing is simple: (*in* 0) is by default mapped to inlet 1, (*in* 1) is mapped to inlet 2, and so on. (*out* 0) is by default mapped to outlet 1, (*out* 1) is by mapped to outlet 2, and so on.

7 Future work / Soft realtime using Guile

There are situations where its more convenient to start new RT instances based on input either from a graphical user interface, midi messages or OSC messages, rather than to start many RT instances in advance and then let those instances handle input.

Too much resources may also be used when forced to start many new RT instances in advance, instead of only starting them when needed.

To schedule RT instances instantly, which in this case means that scheduling an event takes less time than *Just Noticable Difference* (JND), Guile needs to support soft realtime. By interpreting studies by Mäki-Patola and Hämäläinen on continuous sound instruments without tactile feedback. [3], and Adelstein, Begault, An-

derson and Wenze [4] on haptic-audio asynchronity, two studies which measure two very different kinds of JNDs, it seems probable that a JND of less than 20-30ms might be sufficient for almost all situations. But its not unlikely that more than 20-30ms could be tolerant for most kind of use as well.

Soft realtime is also needed to do proper interactive data handling when running Snd as a Pd external.

A large problem for Guile to support soft realtime is Guile's garbage collector. Guile's garbage collector seems to stall scheduling of RT-objects for as much as 100-500ms[10] now and then.

Preliminary tests using Han-Wen Nienhuys and Ludovic Courtès' experimental patches for Guile to let Guile use Hans Boehms garbage collector (HBGC) [5], and further let Guile receive midi to schedule appropriate RT objects in realtime, has shown promising results.[11] But this needs more testing.

So far I have unfortunately not succeeded running Guile with the HBGC in incremental mode. Running the HBGC in incremental mode could make Guile able to guarantee soft realtime performance.[12] I have not yet investegated reasons for the failure though, just observed it crashing.

In addition, patches to at least let Guile do realtime safe memory allocations might be necessary for Guile to run using SCHED_FIFO or SCHED_RR [6] realtime priority without risking priority inversion.

Simply turning off the garbage collector is not an option, especially since memory usage in Guile tends to surpass the size of physical memory surprisingly quickly.

Using a different Scheme implementation other than Guile is also an alternative. RScheme [7] is currently the only known larger scheme implementation who has, or at least earlier claimed to have, realtime support. It might be worth investigating the possibility of using RScheme as an extension language for Snd.

But for now, the safe option is to receive external input, such as MIDI, in Snd-Rt, and not in Guile, although both options are available.

[10] I have not measured the garbage collector latency, but sometime it feels like around 500ms on my machine.

[11] http://lists.gnu.org/archive/html/guile-devel/2007-06/msg00002.html

[12] Hans Boehm has indicated on a mailing list posting that a garbage collection time of 20-50ms could be achievable in incremental mode. http://gcc.gnu.org/ml/java/2006-04/msg00072.html

8 Acknowledgements

This work has partly been funded by the Art Council Norway, Intravision Group[13] and Notam.

Thanks to the following persons for support and help on this work: Chris Chaffe, Alexander Refsum Jensenius, Bjarne Kvinnesland, Jose Rio-Pareja, Jøran Rudi, Bill Shottstaedt, Siren Tjøtta, Anders Vinjar and Sook Young Won.

Web links

CLM	http://ccrma.stanford.edu/software/clm/
Common Music	http://commonmusic.sourceforge.net/doc/cm.html
Pd	http://crca.ucsd.edu/~msp/software.html
Guile	http://www.gnu.org/software/guile/guile.html
HBGC	http://www.hpl.hp.com/personal/Hans_Boehm/gc/
San-Dysth	http://www.notam02.no/~kjetism/sandysth/
Snd	http://ccrma.stanford.edu/software/snd/
Snd-Rt	http://www.notam02.no/arkiv/doc/snd-rt/

References

[1] R. Kelsey. Pre-scheme: A scheme dialect for systems programming, 1997.

[2] William Shottstaedt. Machine tongues xvii: Clm: Music v meets common lisp. *Computer Music Journal*, 18(2):30–37, 1994.

[3] T Mäki-Patola and P Hämäläinen. Latency tolerance for gesture controlled continuous sound instrument without tactile feedback, 2004.

[4] Bernard D. Adelstein, Durand R. Begault, Mark R. Anderson, and Elizabeth M. Wenzel. Sensitivity to haptic-audio asynchrony, 2003.

[5] H.-J. Boehm. A garbage collector for c and c++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[6] M. Harbour. Real-time posix: an overview, 1993.

[7] Donovan Kolbly. Rscheme, 1997.

[13] For the Windows port