REALTIME MUSIC PROGRAMMING USING SND-RT

Kjetil Matheussen
Norwegian network for Technology, Acoustics and Music. (NOTAM)
k.s.matheussen@notam02.no

ABSTRACT

A revolutionary system for programming sound and music in realtime is being presented. The system provides interfaces to extremely efficient high-level and garbage-producing programming languages, which process indivudal samples at a time, sample by sample, in realtime. This is possible by using the Rollendurchmesserzeitsammler garbage collector, the only known conservative garbage collector suitable for hard realtime audio DSP.

The name of the systen is Snd-Rt. Snd-Rt lives inside the Snd sound editor and currently consists of a realtime sound engine, a built-in Scheme-like programming language named "RT", an interface for using the Faust compiler, and an interface for using the Stalin Scheme compiler. Stalin and "RT" can use CLM for DSP operations, while Faust uses its own system.

An interactive Lisp interface is provided for all compilers, [1] and they all support very efficient sample by sample processing and strongly timed [2] coroutines. [3]

Snd-Rt has been used for custom DSP routine prototyping, [4] interactive sound installations, [5] live improvisation (including use of custom-built hardware), [6] exploratory music programming, [7] surround mixing, [8] and to make general sound applications. [9]

1. INTRODUCTION

CLM [1] is a sound synthesis package by Bill Schottstaedt which has been frequently used among composers of electroacoustic music. This paper is about Snd-Rt, one of the environments CLM runs in. Snd-Rt lives inside the Snd sound editor, [10] and together they provide a pragmatic environment for realtime music programming.

Snd-Rt provides interactive interfaces to three different types of compilers, one of which is especially written for Snd-Rt to provide a general programming language with support for sample by sample processing using a common imperative programming style, while Snd provides CLM and almost everything else. The other two compilers are Faust [2] and Stalin [3].

Snd-Rt also lives inside the Guile R5RS Scheme interpreter, which provides the underlying Lisp environment. Snd-Rt has familiar programming paradigms for CLM users, and it provides good performance. One fulfilled goal of the language was that only changing a few lines should be necessary for translating a CLM instrument into an Snd-Rt instrument.

2. THE ENGINE AND THE COMPILERS

2.1. The RT Engine

The RT Engine schedules, mixes and controls the signal processing jobs.

The main task of the RT Engine is to receive <*realtime*> objects created by the compilers, and to provide an interface to control exactly when to run those objects. This interface is hidden from the user, and is accessed by instead calling methods provided by the <*realtime*> class.

The RT engine is hard realtime safe, has backend drivers for both Jack and Pd, provides properly made frame-accurate timing and scheduling, a protection mechanism to avoid locking up the computer, and is controlled entirely from Scheme using Guile running inside a Lisp programming environment such as Emacs.

2.2. Using the "RT" compiler

The "RT" compiler is a compiler for a Scheme-like programming language to generate code for the RT Engine.

The "RT" compiler was specifically written for the Snd-Rt system and is a little bit better suited for live coding than the other two compilers because of shorter compilation time.

The language is coincidentally quite similar to the Lisp dialect PreScheme [4], but while PreScheme is a standalone non-interactive programming language, RT is an interactive domain-specific language specialized for music, sound and realtime.

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.

[1] A feature increasingly known as "on-the-fly".
[2] The expression "strongly timed" was introduced in ChucK and means that scheduling is both on time and frame accurate.
[3] Faust does not have coroutines by itself, but code created by Faust can run inside coroutines created by the other two languages
[4] http://www.notam02.no/~kjetism/sandysth/
[5] http://www.intravisiongroup.com/projects/borgund/galleri.htm
[6] http://www.stanford.edu/~yusukem/images/SuitableAudio.pdf
[7] http://www.notam02.no/~kjetism/notamkonsert2006/
[8] http://lac.linuxaudio.org/?page_id=26
[9] http://www.notam02.no/~kjetism/sandysth/
[10] Snd is made by Bill Schottstaedt as well

The RT language is mostly imperative, but it's not a strict imperative language since it has some support for higher order functions.

The RT programming language includes features such as infix operators, automatic read access to all Guile Scheme variables, and write access to some types of variables created by Guile. All RT variables and signal buses can be read and modified from the surrounding Guile environment as well.

The RT programming language is statically typed with type inference, and it has full support for Lisp low-level macros and Lisp structures. Furthermore it has common music programming language features such as signal buses and sound buffers, quick GUI support, plug-in support and midi. OSC is available on request.

The RT compiler works by translating the Scheme-like input-code into Eval-C code [5]. Eval-C is a domain-specific language provided by Snd, and is basically just C using the S-expression syntax plus various syntactic sugar. [11]

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

2.3. Using the Faust compiler

Faust is a purely functional language written by Yann Orlarey which compiles into extremely efficient machine code.

Faust is a very different language than "RT" since the most common data type seems to the function and not the sample, where instead of building a program by putting together functions which handle samples, the programs are often built by putting together functions which handle functions .

Snd-Rt's Faust interface currently has support for GUI, signal buses and low-level Lisp macros.

2.4. Using the Stalin compiler

Stalin is an almost conforming R4RS Scheme compiler written by Jeffrey Mark Siskind which produces code often running faster than hand-written C.

Snd-Rt's Stalin interface currently has support for midi, signal buses, low-level Lisp macros, a convenient system for adding functions, and an interface for adding functions written in C using the S-expression syntax.

2.5. No control rate and sample by sample processing

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, hence supporting sample by sample processing. Pd, for example, generates 64 frames at each control rate tick, while the CLM generators only generates one frame per call.

[11] Eval-C was originally written to be a convenient language for writing Snd-Rt in, and a language for which RT code can be compiled into, but has later found other uses as well, such as automatically creating wrappers for C libraries and for gluing separate code parts.

This creates a simpler interface for the user, and removes the need to create generators in a different language, usually C or C++.

For example, the following block creates a 400Hz oscillator only by using the sine function (*sin*) and manually increasing the phase for each sample:

```
(let ((phase -0.062))
  (<rt-out> (sin (inc! phase 0.062))))
```

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 synthesis routine shown in the examples section in this paper.

2.6. Why using compilers?

PD, CSound, the SuperCollider server, and many other, probably most other, music programming languages are interpreters since they are dataflow languages where the advantage of low latency and simpler implementation far outweighs the advantage of an increased (and usually insignificant) code efficiency. This is because most of the CPU is spent inside prebuilt generators in those languages.

However, in systems without a control rate, most of the time may very well be spent in the language itself, and the difference in speed between interpreting and running compiled code is therefore quite significant. This is especially important when generating sound in realtime, since we don't want to hear interruptions in the sound.

2.7. Coroutines and dynamic control rate

Coroutines is a more than 45 year old concept[6] which is often used for simulating natural processes.[7] Most programming languages support, or can easely support, coroutines, and the concept recently appeared as "Shreds" in the ChucK music programming language.[8] Later, Graham David Wakefield's music programming system "Vessel" [9] supported coroutines as well.

Coroutines in the "RT" language are implemented by providing a dedicated stack for each coroutine, while coroutines in Stalin are implemented using continuations and the *call/cc* operator.

Coroutines are convenient for playing many voices simultaniously, and since arbitrarily placed breaks belongs to the coroutines concept, features such as dynamic control rate similar to ChucK, pausing, and natural sequential operations, are almost automatically supported.

However, implementing coroutines efficiently in languages supporting sample by sample processing requires some extra care, where the internal coroutine scheduler needs two separate binary heaps: One binary heap for coroutines currently doing sample by sample processing (ie. running inside the **block** operator), and one binary heap for those who currently don't.

2.8. Dynamic memory allocation

Dynamic memory allocation on the heap is provided by using "The Audio Rollendurchmesserzeitsammler" conservative garbage collector.[10] Rollendurchmesserzeitsammler has been specifically written for Snd-Rt and is the only known conservative garbage collector suitable for hard realtime audio DSP.

Currently, Rollendurchmesserzeitsammler is only used in the "RT" language when creating CLM generators and Faust instances, but there are plans to support creating lists and closures as well.

Rollendurchmesserzeitsammler is also used instead of Hans Boehm's garbage collector[11] when Stalin is used as the backend compiler.

Faust does not use the Rolendurchmesserzeitsammler garbage collector, since Faust doesn't allocate any memory during audio processing.

3. EXAMPLES

3.1. Playing A 3 seconds long sound

```
(<rt-out> :dur 0 3:-s (oscil))
```

3.2. Coroutines

```
(<rt-run>
 (define osc (spawn
               (debug "Making a sound.")
               (block (out (oscil)))))
 (wait 5:-s)
 (debug "Exactly 5 seconds later.")
 (stop osc)
 (debug "Now there is silence."))
```

3.3. Sine-wave grain cloud

```
(<rt-stalin>
 (while #t
   (wait (random 30):-ms)
   (define osc (make-oscil :freq (between 50 2000)))
   (define dur (between 400 2000):-ms)
   (define e (make-env '((0 0)(.5 .05)(1 0)) :dur dur))
   (spawn (block :dur dur
                 (out (* (env e) (oscil osc)))))))
```

3.4. Making an oscillator GUI using Faust [12]

```
(<rt-faust>
 (= vol  (hslider "volume" .5 .0 1 0.01))
 (= freq (hslider "freq" 1000 0 4000 0.1))
 (out (vgroup "Osc" (* (osci freq) vol))))
```

3.5. Making an oscillator GUI using "RT"

```
(<rt-out>
 (* (<slider> "Amp" 0 0.5 1)
    (oscil* (<slider> "Freq" 20 200 4000 :log #t))))
```

3.6. Sequential operation

The following block of code shows how to play a pure tone for 5 seconds, followed by a 6 seconds long square sound, only using features provided by the language and not having to schedule more than one *<realtime>* instance:

```
(<rt-run> (block :dur 5:-s (out (oscil)))
          (block :dur 6:-s (out (square-wave))))
```

Due to the regularity of the above block of code, it's tempting to write this general *seq* low-level Lisp macro:

```
(define-rt-macro (seq . rest)
  `(begin ,@(let loop ((rest rest))
              (if (null? rest) '()
                  (cons `(block :dur ,(car rest)
                                (out ,(cadr rest)))
                        (loop (cddr rest)))))))
```

And by using this macro, the operation can be shortened into:
(<**rt-run**> (*seq* (*oscil*) *5:-s* (*square-wave*) *6:-s*)) [13]

3.7. Parallel operation

By using the *seq* macro from the sequential example, it's just spawning a new coroutine for each sound to run the 5 second pure tone and the 6 second square sound in parallel, still without having to schedule more than one *<realtime>* instance:

```
(<rt-run> (spawn (seq (oscil) 5:-s))
          (spawn (seq (square-wave 6:-s))))
```

Like in the example with the sequential operation, it's possible to write a small Lisp *par* macro to make parallel operations less verbose.

3.8. Midi synthesizer

This example shows how to implement a very simple polyphonic midi soft synth:

```
(<rt-stalin>
 (while #t
   (wait-midi :command note-on
     (define osc
       (make-oscil :freq (midi-to-freq (midi-note))))
     (spawn
       (define player
         (spawn (block (out (* (midi-vol) (oscil osc))))))
       (wait-midi :command note-off :note (midi-note)
         (stop player))))))
```

3.9. San Dysth

This example shows a larger example doing sample by sample processing.

San Dysth [5] is a standalone realtime midi soft synthesizer written in Snd using the Snd-Rt language. San Dysth's synthesis technique works by using a set of rules which change state for each sample.

The function *san-dysth-dsp* provides the main synthesis routine for San-dysth. *san-dysth-dsp* returns one sample per call:

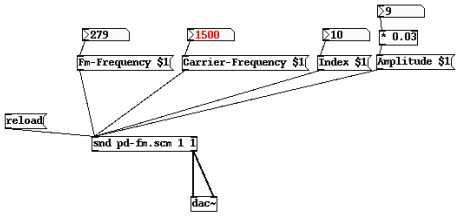
[12] Faust only uses the S-expression syntax in Snd-Rt

[13] This *seq* operator can not be implemented with functions.

```
(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))
```

For a complete implementation of this routine, please refer to [5] or San Dysth's homepage.

3.10. Running Snd as a Pd external



This figure shows Pd using Snd-Rt for doing FM synthesis. Pd and Snd communicate by sending messages back and forth on two ringbuffer to ensure Guile's garbage collector doesn't interrupt sound processing. However, Snd-Rt's signal processing thread runs directly in Pd's main thread. Pd can also be started as an Emacs sub-process to allow interactive development with Pd, Snd and Snd-Rt.

4. FUTURE WORK

The S-expression syntax and the conservative garbage collector could make it very easy to add support for Clean, Common Lisp, Haskell or other high level languages doing interactive realtime sample by sample processing.

5. CONCLUSION

Snd-Rt together with Snd provide a general and very large environment for realtime music programming. It also provides a base environment where operations such as sequential processing and parallel processing can both easily and efficiently be implemented in the language itself.

Snd-Rt also provides convenient access to three very different, high-level, interactive, and high performing programming languages, which all eliminates the use of implementing custom DSP routines in C or C++.

Snd-Rt's home page contains more detailed information about Snd-Rt's scheduler, the RT compiler, and using Stalin or Faust together with Snd:
`http://www.notam02.no/arkiv/doc/snd-rt/`.

6. ACKNOWLEDGEMENTS

The initial work was funded by the Art Council Norway and NOTAM. Intravision Group later funded the Windows port.

Thanks to Herman Ruge Jervell, my advisor at the Department of Informatics at University of Oslo for inspiration and liberty in topics to work on; Bjarne Kvinnesland and Siren Tjøtta for initiating the work on Snd-Rt back in 2004; Jøran Rudi, the director at NOTAM, always supporting my work; Yann Orlarey and Jeffrey Mark Siskind for helping me using their languages; and finally Bill Schottstaedt for various help and all his work on CLM and the fantastic Snd sound editor.

7. WEB LINKS

CLM	`http://ccrma.stanford.edu/software/clm/`
Faust	`http://faust.grame.fr`
Guile	`http://www.gnu.org/software/guile/guile.html`
Jack	`http://www.jackaudio.org/`
Pd	`http://crca.ucsd.edu/~msp/software.html`
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/`
Stalin	`http://cobweb.ecn.purdue.edu/~qobi/software.html`

8. REFERENCES

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

[2] E. Gaudrain and Y. Orlarey. A faust tutorial, `http://faust.grame.fr/pubs.php`, 2003.

[3] Jeffrey Mark Siskind. Stalin - a STAtic Language ImplementatioN, `http://cobweb.ecn.purdue.edu/~qobi/software.html`.

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

[5] K. Matheussen. Extending snd with eval-c and snd-rt, *linux audio conference*, 2008.

[6] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[7] O.-J. Dahl and K. Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[8] Ge Wang and Perry Cook. Chuck: a concurrent and on-the-fly audio programming language, *proceedings of the icmc*, 2003.

[9] Graham David Wakefield. Vessel: A platform for computer music composition, *master thesis*, 2007.

[10] The audio rollendurchmesserzeitsammler. `http://www.notam02.no/~kjetism/rollendurchmesserzeitsammler/`.

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