footils – Using the *foo* Sound Synthesis System as an Audio Scripting Language

Martin RUMORI
Klanglabor, Academy Of Media Arts
Peter-Welter-Platz 2
D-50676 Cologne
Germany
rumori@khm.de

Abstract

foo is a versatile non-realtime sound synthesis and composition system based on the Scheme programming language (Eckel and González-Arroyo, 1994; Rumori et al., 2004; Rumori, 2005). It is mainly used for sound synthesis and algorithmic composition in an interactive *type-render-listen*-loop (the musician's *read–eval–print*-loop) or in conjunction with an editor like the inferior mode of *emacs*. Unlike with other sound synthesis languages, *foo* programs are directly executable like a shell script by use of an *interpreter directive*. *foo* therefore allows for writing powerful sound processing utilities, so called *footils*.[1]

Keywords

foo, scheme, scripting, algorithmic composition, sound utilities

1 Introduction

Scripting has played a major role in the development of computer systems since the early days. Scripting often means being a *user* and a *programmer* at the same time by accessing functions of applications or operating systems in an automated, "coded" way rather than interactively.

A major design principle of the UNIX operating system is to create several simple applications or tools which are suitable for exactly one purpose and to combine them in a flexible way to implement more complex functionalities. This is possible through the well-known UNIX concepts of pipes and file redirections. A powerful command line interpreter, the *shell*, allows for accessing these concepts both in interactive mode as well as in so called *shell scripts*. It is quite easy to generalize an interactive shell command for using it in a script and vice versa. In fact, the UNIX shell programming language has started to blur the distinction between *user* and *programmer*.

[1]Use of the term *footils* by courtesy of Frank Barknecht, see http://www.footils.org

UNIX shell scripts are often used for recurring custom tasks closely related to the operating system itself, such as system administration, maintenance and file management. Apart from that, there are many scripting languages for special purposes, such as text processing (awk, Perl). Scripts written in one of these languages can be seamlessly integrated with UNIX shell scripting by means of the so called *interpreter directive* at the beginning of a script (as documented in `execve(2)`):

```
#!/usr/bin/perl
```

Those scripts appear and behave like any other UNIX program or shell script and thus get *scriptable* itself. This property of being "recursive" makes shell scripting so powerful.

2 Scripting and computer music

2.1 Standalone applications

In the field of computer music, composers often deal with graphical standalone applications, such as Ardour or Pd, or with dynamic languages in an interactive fashion, such as SuperCollider. While these tools are very powerful in terms of harddisk recording, sound synthesis or composition (like Perl for text processing), they do not integrate in the same way with the operating system's command line interface as textual scripts (unlike Perl for text processing). In most cases, however, this is not necessary or desirable.

Pd can be launched without showing its GUI. the patch to be executed can be given at the command line, including initial messages to be sent to the patch. SuperCollider's language client, *sclang*, may be fed with code through its standard input which then is interpreted. This level of shell automation is already sufficient for tasks such as starting a live session or an interactive sound installation.

2.2 Soundfile utilities

A major task when using a computer for any task is file maintenance. This is especially true for dealing with soundfiles. Common tasks include conversion between different soundfile types, changing the sample rate or the sample format, separating and merging multichannel files, concatenating soundfiles, removing DC offset or normalizing. As for sorting or archiving text files, this kind of tasks are often applied to many soundfiles at a time and therefore should be scriptable.

In order to accomplish those tasks, a lot of command line utilities are available which fully integrate which shell scripting in the abovementioned sense. Examples for such utilities are *sndinfo* and *denoi* included in Csound (Boulanger, 2005), the tools bundled with *libsndfile sndfile-info*, *sndfile-play* and *sndfile-convert*, or *sndfile-resample* from *libsamplerate* (de Castro Lopo, 2006). Another example is the well known *sox* program, which also allows for some effects processing (Bagwell, 2005).

Those tools can be called from a shell script in order to apply them comfortably to a large number of files. The following *bash* script might be used to remove mains power hum from a number of soundfiles specified on the command line:

```
#!/bin/bash

SOX=sox
FREQUENCY=50 # european origin
BANDWIDTH=6
SUFFIX="_br"

while getopts ":f:b:s:" OPTION; do
  case $OPTION in
    f ) FREQUENCY=$OPTARG ;;
    b ) BANDWIDTH=$OPTARG ;;
    s ) SUFFIX=$OPTARG ;;
  esac;
done

shift $(($OPTIND - 1))

for INFILE; do
  OUTFILE=‘echo $INFILE | sed -r -e \
      "s/^(.*)(\.[^\.]*)$/\1${SUFFIX}\2/g"‘
  $SOX $INFILE $OUTFILE \
    bandreject $FREQUENCY $BANDWIDTH;
done
```

While using command line soundfile tools this way might be quite elegant, they still can only run as they are. It is not possible to directly access and manipulate the audio data itself from inside such a script.

2.3 Scriptable audio applications

Apart from that, there are operations on soundfiles which are much closer related to the artistic work with sound itself, such as filtering or effects processing of any kind, mixing, arranging or simply "composing" based on algorithms and/or underlying sound material. While requesting scripting capabilities is evident for doing file related tasks mentioned above, the latter procedures are mostly done inside single standalone applications or sound synthesis systems.

Attempts have been made to open the processing scheme of the audio data to a script interface. One approach was realized in the *Computer Audio Research Laboratory (CARL) Software Distribution* at *CRCA (CME)* since 1980 (Moore and Apel, 2005). The *CARL* system consists of several independent small UNIX programs for reading and writing soundfiles, as well as sound synthesis, effects processing and analyzing audio data. They communicate with each other via ordinary UNIX pipes. This way it is possible to generate a kind of signal processing patches as in Pd, but by means of an arbitrary shell scripting language:

```
$ fromsf infile.ircam | \
    filter bandreject.fir | \
    tosf -if -os outfile.ircam
```

This approach is quite smart, as it allows for using the UNIX command line for audio processing in the same way as for text processing. It is even possible to set up parallel processing pipes with the `para` program. since the shell language is not powerful enough for expressing those parallel pipes, this program has to use its own syntax, which unfortunetaly causes some deformation to the aesthetical integrity of the *CARL* approach.

Another approach was implemented by Kai Vehmanen in his *ecasound* application (Vehmanen, 2005). *ecasound* allows for creating flexible so called signal processing *chains*. The parameters for these chains are specified via command line options or from files containing the chain rules. Therefore *ecasound* is fully scriptable from the commandline or from inside shell scripts.

```
$ ecasound -i:infile.aiff -o:outfile.aiff \
    -efr:50,6
```

Ecasound allows for building up parallel processing chains at the same grammatical level of the scripting language used for simpler tasks. *ecasound* does not only operate on soundfiles, but may also record, process and play audio in realtime, optionally using controller data such as MIDI.

Both the *CARL* tools and *ecasound* are examples for tools which allow for directly accessing the audio processing scheme. by calling them from inside a shell script, similar to the *sox* example above, one is able to create very sophisticated sound processing utilities.

3 Using *foo* for writing audio processing scripts

foo's approach for allowing scripting is different from the abovementioned ones. *foo* is neither a closed standalone application nor a utility especially designed for scripting. *foo* is a sound synthesis and composition environment for non-realtime use based on the dynamic, general purpose programming language Scheme.

3.1 History of *foo*

foo was developed by Gerhard Eckel and Ramón González-Arroyo in 1993 at ZKM, Karlsruhe, for the *NeXTStep* platform. The low-level *foo kernel* is written in Objective-C, while the higher level parts are written in Scheme.

Starting from 2002, *foo* was ported to the Linux platform by the author using the *GNUStep* framework (Fedor et al., 2005), a free *OpenStep* implementation. The project was registered at *SourceForge* in 2003. In 2004, *foo* was ported to Mac OS X, where it runs natively using the *Cocoa* (formerly *OpenStep*) framework.

Also in 2004, *foo* was partitioned into *libfoo*, which contains the signal processing primitives, and *elkfoo*, which consists of the interface to the Elk Scheme interpreter (Laumann and Hocevar, 2005). This should make a possible future transition to a different Scheme implementation easier. For easier packaging and cross-plattform-building, *foo* got an *autotools* build system in the same year.

3.2 Major concepts of *foo*

Them main purpose of *foo* is to provide a high quality, highly flexible sound synthesis and music composition system.

foo provides signal processing primitives written in Objective-C which can be accessed from inside the Scheme environment. Unlike CLM (Schottstaedt, 2005) or Csound, *foo* does not distinguish between *instruments* and *events (score)*. Nevertheless, it is easily possible to express a Csound-like semantics of orchestra and score with *foo*, or the concept of *behavioral abstractions* as in Nyquist.

By means of the *foo* primitives, static signal processing patches can be generated and executed. Temporal relationships are expressed in hierarchical time frames which are relative to the surrounding one.

Higher level concepts, such as envelopes or musical processes, are entirely implemented in Scheme in the *control* library by Ramón González-Arroyo, which is part of *foo*.

This openness allows for using *foo* for very different tasks: apart from algorithmic composition based on highly abstracted Scheme constructs as found in the *control* library, it is also possible to use *foo* on the *kernel* level for simple tasks like converting soundfiles, extracting or merging channels, or effects processing.

Like CLM, *foo* is usually used interactively by entering and evaluating Scheme expressions, which construct signal processing patches or render them into soundfiles. It is also common to use *foo* in conjunction with an editor, such as the *inferior*-mode of *emacs*. Since Scheme is an interpreted language (at least in the implementation used so far), *foo* programs can also made directly executable from the shell command line prompt.

This allows for writing versatile "shell" scripts which are not bound to the capabilities of a specific application like *sox*, but rather can benefit from the full power of a generic programming *and* sound synthesis language. Writing foo scripts ("footils") also differs from approaches such as *ecasound* in that there is no distinction anymore between the calling language (shell) and the audio processing language (*ecasound* chain rules).

3.3 Making *foo* scripting comfortable

Several issues had to be solved in order to make *foo* programs directly executable as scripts while not affecting the interactive use of *foo*. In the following, some of these issues are described.

3.3.1 Understanding the *interpreter directive*

According to the manpage of `execve(2)`, a script can be made executable by adding an *interpreter directive*:

> execve() executes the program pointed to by filename. filename

must be either a binary executable, or a script starting with a line of the form `#! interpreter [arg]`. In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as `interpreter [arg] filename`.

Since the original *elk* executable did not accept a scheme file as a direct argument, a different startup executable for *foo* has been written. This was already done in the first version of *foo*. A *foo* script can be build by adding a line like

```
#!/usr/local/bin/foo
```

at its first line.

3.3.2 Load stages, packages and script arguments

The startup procedure of the *foo* sound synthesis program follows a multi-stage approach:

- start the interpreter executable and evaluate command line options directed to the interpreter itself (heap size, etc.)
- hand over control to the scheme interpreter by loading the toplevel scheme file, load *foo* primitives into the interpreter, evaluate scheme stage command line options, load *foo* packages
- if a script file was specified on the command line, build the command line left over for the script and execute it, else enter the interactive read-eval-print-loop

This load procedure indicates a problem which arises when specifying options to the *foo* executable:

```
$ foo --unload control
Usage: foo [options] [arguments]
...

$ foo -- --unload control
```

The first invocation of *foo* fails, because the option `--unload` is not understood by the scheme interpreter's command line parser. In order to make sure it “reaches” the scheme initialization stage, it has to be “quoted” with `--` to bypass the *elk* interpreter.

Invoking *foo* with the option `--unload control` will prevent the control library from being loaded into *foo* at startup. This might be suitable for scripts which do not need this package in order to speed up the initialization process. Therefore the interpreter directive for such a script should read:

```
#!/usr/local/bin/foo -- --unload control
```

Another problem occurs at this point: `execve(2)` apparently does not tokenize multiple initial arguments given in an interpreter directive into several arguments but passes them as a whole as one argument to the interpreter.

In order to be able to parse multiple options given in the interpreter directive, the *foo* executable contains a hack which tries to tokenize `argv[1]` into several arguments according to a certain heuristic, constructs a corrected argument vector and re-executes itself.

3.3.3 Command line parsing

foo scripts have to be able to access the part of the invoking command line following the scriptfile argument in order to understand script options. Command line parsing therefore is a common task in *foo* scripts.

In order to make command line parsing easier for script authors, a scheme library `cmdline` is included with *foo*. It features alternative options such as long- and shortopts, options with or without parameters, different possibilities of specifying multiple parameters to options, and automatic help message generation:

```
#!/usr/local/bin/foo -- --unload control

(require 'cmdline)

(let*
  ;; equiv-opts-list | mandatory? | \
                  with-params? | help-string
  ((option-list
    '((("--help" "-h") #f #f "this help screen")
      (("--outfile" "-o") #t #t "output file")
      (("--type" "-t") #f #t "file type")
      (("--sformat" "-s") #f #t "sample format")
      (("--srate" "-r") #f #t "sample rate")
      (("--channels" "-c") #f #t "channels")))
   ;; show help message
   (help
    (lambda ()
      (format #t "~a: script foo~%"
        (car (foo:script-args)))
      (format #t "usage:~%")
      (format #t "~a~%"
        (cmdline:help-message option-list))
      (exit))))

  ;; help requested?
  (if (cmdline:option-given?
        (foo:script-args) option-list "--help")
      (help))

  ;; commandline valid?
  (if (not (cmdline:cmdline-valid?
         (foo:script-args) option-list #t))
      (help)))
```

This script will produce the following output when invoked with no arguments:

```
$ ./script.foo
(cmdline:validate) \
  mandatory option missing: --outfile
./script.foo: script foo
usage:
   --help, -h             this help screen
   --outfile, -o <args>   output file
   --type, -t <args>      file type
   --sformat, -s <args>   sample format
   --srate, -r <args>     sample rate
   --channels, -c <args>  channels
multiple <args>: --opt <arg1> --opt <arg2> \
              or --opt <arg1,arg2,...>
```

Other functions of the `commandline` scheme library not shown in this example include reading the parameter lists of specific options or getting the remaining arguments of the command line, e. g. file arguments.

3.3.4 Interacting with the outer world: `stdin` and `stdout`

Reading from standard input and writing to standard output from *foo* scripts is important if executed inside a pipe. Imagine calling a *foo* script like this:

```
$ find . -name '*.wav' | script.foo
```

This will mean reading a file list from standard input, which can be accomplished with standard scheme functions. The following code reads the files from standard input into the list `files` and the number of files into `num-files`:

```
(let*
  ((files
    (do ((file-list '()
                    (cons last-read file-list))
         (last-read))
        ((begin
           (set! last-read (read-string))
           (eof-object? last-read))
         (reverse file-list))))
   (num-files (length files)))

  ...)
```

Writing to standard output is done similarly through standard scheme functions.

3.4 *footils*

footils is a collection of *foo* scripts written so far by Gerhard Eckel and the author. The aim of *footils* is to provide a set of powerful and robust scripts for the musician's everyday use. *footils* currently consists of the following scripts:

fsconvert convert soundfiles

fssrconv do a samplerate conversion on soundfiles

fsextract extract channels from multichannel files

fsfold fold a soundfile over itself and normalize

fskilldc remove DC from soundfiles

fsmono2stereo create stereo file from mono file

fsquadro2stereo create stereo file from quadro file

fsnorm normalize soundfiles

fsregion extract a temporal region from soundfiles

fsreverse reverse soundfiles in time

fstranspose transpose soundfiles

fscat concatenate soundfiles

These scripts are currently refactored and integrated with the *foo* distribution.

4 Conclusions

Several issues of scripting in the field of computer music have been considered. It turned out that with most current software it is not possible to write scripts which are seamlessly accessible from the UNIX command line and at the same may benefit from the power of a fully featured programming and sound synthesis language.

It has been shown that writing scripts with *foo* might be able to close this existing gap.

5 Acknowledgements

I would like to thank Gerhard Eckel and Ramón González-Arroyo for developing *foo* and for starting the development of scripts for the *foo* system.

Many thanks to Frank Barknecht for his kind permission to use the term *footils* for the *foo* script collection.

References

Chris Bagwell. 2005. Homepage of sox. *http://sox.sourceforge.net*.

Richard Boulanger. 2005. Official homepage of csound. *http://www.csounds.com*.

Erik de Castro Lopo. 2006. Homepage of libsndfile, libsamplerate etc. *http://www.mega-nerd.com*.

Gerhard Eckel and Ramón González-Arroyo. 1994. Musically salient control abstractions for sound sound synthesis. *Proceedings of the 1994 International Computer Music Conference*.

Adam Fedor et al. 2005. The official gnustep website. *http://ww.gnustep.org*.

Oliver Laumann and Sam Hocevar. 2005. Homepage of Elk Scheme. *http://sam.zoy.org/projects/elk/*.

F. Richard Moore and Ted Apel. 2005. Homepage of CARL software. *http://www.crca.ucsd.edu/cmusic/*.

Martin Rumori, Gerhard Eckel, and Ramón González-Arroyo. 2004. Once again text and parentheses – sound synthesis with foo. *Proceedings of the 2004 International Linux Audio Conference*.

Martin Rumori. 2005. Homepage of foo sound synthesis. *http://foo.sf.net*.

Bill Schottstaedt. 2005. The clm home page. *http://ccrma.stanford.edu/software/clm*.

Kai Vehmanen. 2005. Homepage of ecasound. *http://www.eca.cx/ecasound/*.