CompScheme: A Language for Composition and Stochastic Synthesis

Luc Döbereiner
Institute of Sonology, The Hague, The Netherlands

***Abstract*—In this paper, I present a programming language for algorithmic composition and stochastic sound synthesis called *CompScheme*. The primary value generating mechanism in the program are *streams*, which allow the user to concisely describe networks of dynamic data. Secondly, I present *CompScheme*'s event model, which provides a framework for building abstract structural musical units, exemplified by showing *CompScheme*'s functionalities to control the *SuperCollider* server in real-time. Thirdly, I discuss *CompScheme*'s stochastic synthesis functionality, an attempt to generalize from I. Xenakis's *dynamic stochastic synthesis* and G.M. Koenig's *SSP*.**

I. Introduction

CompScheme is a program for algorithmic music composition and stochastic sound synthesis written in Objective Caml (*OCaml*) [6]. *CompScheme* can be used in two ways, as a library for developing applications in *OCaml*, or by accessing its functionality interactively through an interface language. All the code examples in this text are written in the interface language. The primary value generating mechanism in the program are streams, which are a concept from functional programming, which allow the user to concisely describe networks of dynamic data. Streams themselves are rules for generating values. Since streams can be combined and even used to generate new streams, the rules themselves become the object of composition. "Composing with rules" is thus not only interpreted as the mere application of a rule but the actual composition of rules, an idea that is very prominent in functional programming if one sees rules as functions.

The interface language of *CompScheme* is an implementation of the functional programming language *Scheme*[1] [1]. Instead of having a graphical environment or a fixed work flow, where the user can either visually or by filling out forms or questionnaires construct networks and derive musical data, *CompScheme* requires the user to have a degree of programming proficiency. By using an elegant, popular, small, and powerful general-purpose language such as *Scheme*, the user has all of its expressiveness and means of abstraction at his or her disposal. The user can develop full-range programs, or make small experiments by plugging together built-in streams and output functions.

Internally, *CompScheme* consists of several modules, which contain functions for specific fields of application. Data generated in *CompScheme* can be written out in several ways, such as in Wav audio files, Midi files, and binary OSC files for *SuperCollider*. It is also possible to control the *SuperCollider* server in real-time, plot and draw data. Furthermore, *CompScheme* has an event type system, which features built-in event types, and the possibility to create custom event types by bundling named parameters, setting default values, creating transformation and output functions.

CompScheme runs on *Mac OS X* and *Linux*. The top-level interpreter can run as a command line program, in a *Scheme*-mode in an editor such as *Emacs*, or on *Mac OS X* in a specially developed *Cocoa*-application, which follows the usual editor-and-listener design. A beta version of *CompScheme* can downloaded from http://sourceforge.net/projects/compscheme/.

In this text, I discuss the concept of *streams*, as well as some issues and design ideas of *CompScheme*, which relate to *streams*. Subsequently, I present *CompScheme*'s event model, which provides a framework for building abstract structural musical units. The discussion of the event model is followed by a concrete example, the modeling of the first structure of Gottfried Michael Koenig's piano piece *Übung für Klavier*. Subsequently, the expressiveness of multi-layer event streams is exemplified by presenting *CompScheme*'s facilities for controlling the *SuperCollider* server (SC Server) in real-time. Finally, the last section deals with *CompScheme*'s functions for stochastic synthesis. Starting from finding a generalization of G.M.Koenig's *SSP* and I.Xenakis's *Dynamic Stochastic Synthesis*, I have tried to develop a framework, which facilities experimentation in this field. We will see that, here, the event model is applied to a lower level, the digital sample itself.

II. Streams

As H. Abelson and J. Sussman state in their book *Structure and Interpretation of Computer Programs*, "programs must be written for people to read, and only incidentally for machines to execute."[1] Programming languages are primarily tools to express ideas. The formal nature of programming languages stipulates abstraction and generalization. Thus, through programming the structure of an idea may be revealed. Our means of expression shape what we can express. As Ludwig Wittgenstein famously formulated, "die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt." ("The limits of my language mean the limits of my world.")[10] Music composition programming languages, therefore, influence our ideas

[1]The implementation of the interface language is based on *Schoca*: http://home.arcor.de/chr_bauer/schoca.html

of music. It may, thus, be argued, that the choice of a programming language also has musical consequences.

A musical performance or playback is a continuous stream of sound. In any computer representation this continuum, however, is broken up into a discrete sequence of values. The common digital representation of sound in form of a sampled waveform, common practice musical notation, as well as event-based higher-level musical abstractions follow this rule. *CompScheme* is built around the data type *streams*, which are an elegant and simple way of dealing with sequences of values, that is widely known, used, and "one of the most celebrated features of functional programming."[9] Whereas in most imperative and object-oriented systems these sequences are usually created by some iteration that collects the values or a mechanism that involves change of state, streams are persistent.

> Stream processing lets us model systems that have state without ever using assignment or mutable data. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment.[1]

This persistence also has advantages in musical applications. The main one, of course, is that no values are lost and everything that has been produced, and therefore everything that *will* be produced, can be referred to, which provides the user with the possibility to look into the "future" of a process and make decisions depending on what is going to happen.

Streams are flexible, they can be combined, can contain values of any kind, such as other streams or functions, and are collections as well as generative mechanisms. In more imperatively oriented systems, values are usually generated with an iterative process and collected in lists. In order to combine several processes, one has to generate values, collect them, operate on the collection, collect again, and so forth. Streams operate differently in that they generate values on demand. If several streams are combined they are piped into each other, one stream generates as much as the next one demands. Hence, infinite processes can easily be expressed. Streams, thus, allow the user to concisely describe networks of dynamic data by plugging simple parts together.

A. Finite and Infinite Streams

Most stream constructing functions in *CompScheme* return infinite streams, and it is important for the user to keep the distinction between finite and infinite streams in mind. Some operations on streams, such as plotting a stream, searching for the minimum or maximum element, accessing the last element, or appending streams require the input streams to be finite. Figure 1 shows the definition of an infinite stream and the construction of a stream which contains the first three elements of that stream appended onto the stream itself in its infinite form. The function `st-first n stream` limits the `stream` to the first `n` values, it thus converts an infinite stream into a finite one.

```
> (define my-stream1 (st-sum 1 0))
> (for-example (st-append
>                (st-first 3 my-stream1)
>                my-stream1))
(0 1 2 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
```

Fig. 1. Appending a finite and an infinite stream

B. "Inherited Ending"

When building networks of streams, in which streams act as supplies for parameter values of other streams, it is not necessary to limit the topmost stream explicitly, if a stream in the network is already finite. In other words, if any stream in a network of streams is finite, the whole network is finite and contains as many elements as the shortest stream in the network does. The stream of random values constructed in figure 2 and displayed in figure 3 ends after 10000 elements because the parameter for the lower boundary is a linear shape from 1 to 20, which ends after 10000 elements.

```
> (simple-plot
>   (st-random-value (st-line 10000 1 20) 20))
```

Fig. 2. Plotting the implicitly finite stream

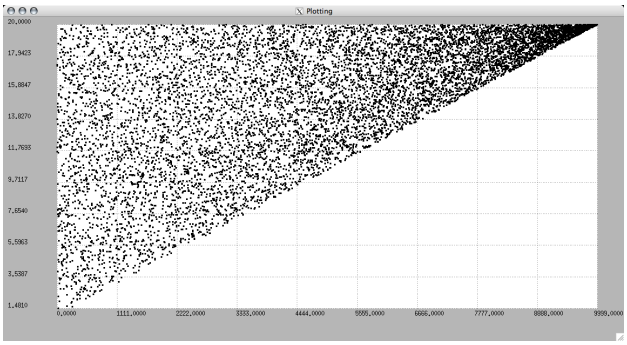

Fig. 3. The implicitly finite stream

C. Higher-Order Functions and Streams

"The most powerful techniques of functional programming are those that treat functions as data."[9, page 171] *Higher-order* functions or *functionals* are functions, which operate on other functions. In functional programming it is common to abstract by defining functions, which take other functions as arguments. This can help revealing the general structure of a method. In a stream based approach many transformations, filterings, and techniques of creating variation can be expressed in terms of higher-order functions.

Figure 4 shows the filtering of a stream of random values. The returned stream only contains those elements of the original stream, which fulfill the predicate $x \ mod$

$12 = 0$. If the numbers were to be interpreted as midi note numbers, the returned stream would only contain the pitch C in different registers. The function `st-filter function stream` returns a stream, whose elements are those of `stream` for which `function` is true.

```
> (for-example
>   (st-filter (lambda (x) (= 0 (modulo x 12)))
>              (st-rv 0 127)))
(48 120 72 96 72 120 60 48 60 72 72 108 24 24 12
24 36 72 36 0)
```

Fig. 4. Filtering a stream

Figure 5 demonstrates the function `st-apply function` $stream_1 \ldots stream_n$ by creating a triangular random distribution. The function `st-apply` constructs a stream, whose elements are the results of successively applying the function to the streams given. A triangular random distribution can be created by taking the average of two uniformly distributed random values in the same range.

```
(st-apply
  (lambda (x y) (/ (+ x y) 2))
  (st-rv 0.0 1.0)
  (st-rv 0.0 1.0)))
```

Fig. 5. Using `st-apply` to create a triangular random distribution

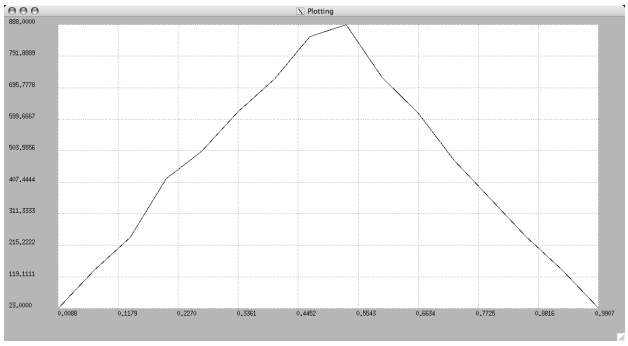

Fig. 6. A histogram of the first 10000 elements of the stream from figure 5

D. Defining Streams

Besides using the built-in streams, the user may also define his or her own streams. One way to do this is to define a function for a specific stream network. In this way, we can define a function which returns a stream of triangularly distributed random numbers by naming the stream described above, illustrated in figure 7.

```
(define (st-trirnd minium maximum)
  (st-apply (lambda (x y) (/ (+ x y) 2))
    (st-rv minimum maximum)
    (st-rv minimum maximum)))
```

Fig. 7. A simple stream definition

If the desired stream can not be built by combining already available streams, the function `st-cons value continuation` can be used. This function builds a stream, which contains `value` as its first element and `continuation` will be a delayed expression that constitutes the rest of the stream. Figure 8 shows the definition of a stream using `st-cons`. The first value is `start` and the continuation will be built by recursively calling `st-product` with the start value multiplied by `factor`, thus creating an exponentially increasing sequence.

```
(define (st-product start factor)
  (st-cons start
    (st-product (* start factor) factor)))
```

Fig. 8. A stream definition using `st-cons`

However, the definition shown in figure 8 has one limitation, being that the factor is constant and cannot be controlled using a stream. Figure 9 shows an improved version in which the factor can be dynamically controlled using a stream. In order to conform to the "inherited ending" principle, one has to check first if the factor stream is empty. For each iteration the 'current' value of the factor stream is accessed using the function `this` and updated for the next call using the `next` function, which returns the stream without the first (i.e. 'current') element. For values which are not streams, the functions `this` and `next` act as identity functions.

```
(define (st-product start factor)
  (if (empty-stream? factor)
    empty-stream
    (st-cons start
      (st-product
        (* start (this factor))
        (next factor)))))
```

Fig. 9. A stream definition using `st-cons` and a stream as argument

III. The Event Model

CompScheme's event model provides a framework for building abstract structural units. Musical events are most commonly represented by bundling values for the description of parameters together. In this way, a musical event can be seen as a list of name-value pairs. A name may, for example, be "freq" and its associated value the number 440. *CompScheme*'s event model provides means of building name-value pairs, by defining *event types*. Event types are name-value pairs which have a name and default values. *CompScheme* has a number of general functions with which values and names can be accessed and events transformed. Events, however, do not need to be understood only as lower-level musical events, such as notes, messages to a synthesis processes etc., but may as well be representations of higher-level structural units, such as sections, passages, phrases, blocks, or entire pieces. In that sense, *CompScheme* offers a simple, yet

powerful event model, which enables the free construction and aggregation of possibly inter-dependent parametric control structures on multiple temporal and structural levels.

A. Simple Event Types

Figure 10 shows the definition of a simple event type called `myevent1` with three parameters, `start`, `dur`, and `freq`, and default values associated.

```
(defevent myevent1
  (start 0.0)
  (dur 1.0)
  (freq 440.0))
```

Fig. 10. Defining a simple event type

Figure 11 demonstrates how a stream of events can be created using the function `event-stream`. In this example, the starting values are a series of numbers starting with 0.0 and increasing by 1. The frequency parameter is controlled by an exponentially distributed sequence of random numbers. The duration parameter is not controlled and will thus be the default value from the event type definition, i.e. 1.0. It is also possible for the user to provide more parameters than given in the event type definition, the event will then extend automatically and hold the additionally given values too.

```
(define eventstream1
  (event-stream 'myevent1
    (start (st-sum 1 0.0))
    (freq (st-exprand 100.0 1000.0))))
```

Fig. 11. Constructing an event stream

In general, it is the user's responsibility to define in which way an event is to be interpreted. The function shown in figure 12 can be used to output an event of the defined type in a *Csound* score file syntax with the fixed instrument number 1.

```
(define (print-myevent1 ev)
  (write "1 ")
  (map (lambda (x) (write x) (write " "))
    (list
      (event-get 'start ev)
      (event-get 'dur ev)
      (event-get 'freq ev)))
  (newline))
```

Fig. 12. Defining a printing function

B. Higher-Level Events

CompScheme also has a number of built-in event types, such as different midi and *SuperCollider* events. The functions, which output midi or *SuperCollider* events, require streams which contain events that contain at least all of the parameters, which the respective built-in types have. They may, however, contain additional name-value pairs. An event in *CompScheme* is not only to be created in the last instance, as a bundling of values before the data is written out, but may also be a representation of a higher-level structural element, which requires further interpretation. Thus, a hierarchy of events may be created and top-level or intermediate-level events and their interpretation can be created and changed independently.

The event type defined in figure 13 stands for a higher-level construct, a section. In this simple example, a section has five properties: a time offset (`offset`), a duration (`dur`), a minimum frequency (`freqmin`), a maximum frequency (`freqmax`), and starting frequency (`freqstart`).

```
(defevent mysection1
  (offset 0.0)
  (dur 10.0)
  (freqstart 100.0)
  (freqmin 100.0)
  (freqmax 10000.0))
```

Fig. 13. Defining a simple higher-level event type

However, an event is only given meaning through interpretation. Figure 14 shows the definition of a function, which constructs a stream of events of the above defined type `myevent1` with the parameters from a given event of type `mysection1`. The function `st-until` ends the "section" when the start value of the lower-level event stream is greater than the duration specified in the higher-level event.

```
(define (interpret-mysection1 section)
  (let ((offset (event-get 'offset section)))
    (st-until
      (lambda (event)
        (> (- (event-get 'start event) offset)
           (event-get 'dur section)))
      (event-stream 'myevent1
        (start (st-sum 0.1 offset))
        (freq
          (st-walk
            (event-get 'freqstart section)
            (st-rv -200.0 200.0)
            (event-get 'freqmin section)
            (event-get 'freqmax section)))))))
```

Fig. 14. Interpreting the defined event

Figure 15 shows the creation of twenty sections by creating the higher-level event stream and mapping the interpretation function, defined in figure 14, over it. Figure 16 shows the frequencies of the twenty created sections.

This simple example demonstrates the elegance and ease with which a number of sections can be produced from a higher-level description. The possibility to define abstract, higher-level structural units enables the composer to establish long-term relationships among sections, phrases, units, or blocks. It also facilitates the algorithmic organization of form concerning decisions.

```
(st-apply
   interpret-mysection1
   (st-first 20
      (event-stream 'mysection1
      (offset (st-sum 10.0 0.0))
      (frestart (st-rv 1000.0 5000.0))
      (freqmin (st-rv 100.0 8000.0))
      (freqmax (st-rv 100.0 8000.0)))))
```

Fig. 15. Controlling the higher-level event stream

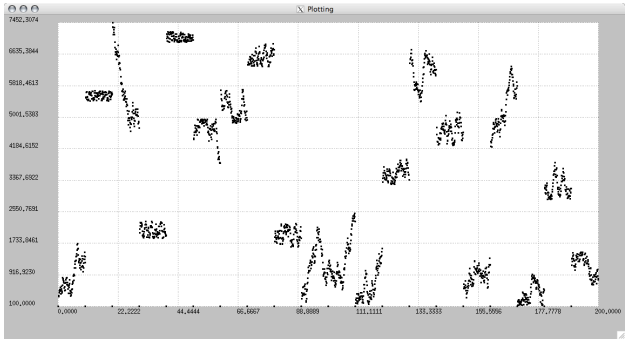

Fig. 16. The event's frequencies over their starting points

IV. Modeling Structure 1 of Koenig's *Übung für Klavier*

Gottfried Michael Koenig's piece *Übung für Klavier*, composed in 1969, is the first piece he realized with his program *Project 2*. *Project 2* was not designed for the realization of a single piece, but as a general composition program. As Koenig writes in the preface of the score, it is therefore, "necessary to generalize individual composing habits; an attempt must be made to formulate a theory – however limited – of composition."[3] *Übung für Klavier* (Study for Piano) is thus a first test of this theory. The word *Übung*, meaning 'study', but also 'practice', does not primarily refer to the player or the instrument, but rather to the composition of the piece; it is a study in writing a piece with the program *Project 2*. The title thus identifies the work as a test object and reveals the critical reflection of the work itself on the model and material from which it is derived. The piece's first section, or *structure*, will here again serve as a test object for testing the capabilities and design of *CompScheme*.

The piece consists of 12 structures and 3 variants of each of these structures, of which the pianist chooses one variant for each structure to be played. Hence, there are $3^{12} = 531441$ possible combinations of variants, of which one is performed.

Project 2 is based on a number of basic notions: compositional rules, musical quantities (*data*), characteristics of musical sound (*parameters*), combinations of rules and data (*structure formula*), and "combinatorial possibilities"[5] resulting from a structure formula (*variants*). As Koenig states, "The purpose of PROJECT 2 (PR-2) is to 'calculate musical structure variants'."[5] Since aleatoric decisions are employed in different phases of the program, it is not necessary to enter additional data for the creation of *variants* from a *structure formula*. The rules and the set of data are fixed for a specific structure and the computer constructs variants.

By following a questionnaire of over 60 questions, the *composer* describes a certain *model* of which *variations* are created. In *Übung für Klavier*, the 12 structures are *structure formulas*, variants are created by the use of aleatoric procedures. There are eight parameters that describe a *structure formula*: instrument, harmony, register, entry delay, duration, rest, dynamics, and mode of performance. Since the piece is only for one instrument, the instrument parameter is ignored.

The basic principle for the construction of musical data in *Project 2* is a three-layered process of entering, grouping, and selecting elements, the so-called *List-Table-Ensemble* principle. The construction of data for almost all parameters follows this principle. In the first instance, the composer enters a list of "allowed" elements; a basic reservoir of the smallest components. In the second layer, the user forms groups of these elements in a table, a list of selections from the list of "allowed" elements. Consequently, an *ensemble* is formed by selecting groups from the table. Thus, there are three layers of selection, i.e. choices of elements from a given supply. The first two selections are done by the composer. The composer chooses the basic elements and determines their grouping in the table. These selections and groupings remain the same for all variants of a structure. The third selection, however, is done with the help of the selection programs *alea*, *series*, and *sequence*, which chose the number and indices of the table-groups to be inserted into the *ensemble*. The third level differs from the first two levels of selection in that the selection can be changed for each variant and not single elements, but whole groups of elements are selected.

The *List-Table-Ensemble* principle is an extension of the series as a basic building block, which can be permutated in order to derive relationships. Aside from the input of the initial reservoir of "allowed" values, the other levels operate on indices. The concrete values are substituted by pointers to concrete values. The operation on pointers is an abstraction, which constitutes an intermediate meta-level, through which aspects of the musical reality become controllable. In doing so, numbers operated on never refer to themselves, i.e. calculations and the construction of numerical structures are not done for their own sake, but always for the purpose of referring to concrete values. Thus, numerical values always serve the description of musical situations. The translation of concrete values into indices creates a level on which processing is possible. The *List-Table-Ensemble* principle clearly discriminates between the material and its order. The initial input of "allowed" values is an unordered set of possible elements and the operations on indices establish orders and groupings, thereby breaking the series up into material and sequence. In *CompScheme*, I will not directly model this work flow, I will rather divide the construction into three different steps: the definition of user supplied data specific to structure 1 of *Übung für Klavier*, the definition

of functions, which model the workings of *Project 2*, especially with regard to entry delay production, and thirdly plugging together the necessary stream functions and the user supplied data to define a function, which returns the structure.

In this section, I will show how the first structure of this piece can be modeled in *CompScheme*. Due to the partly incomplete description of the structure it is not possible to regenerate the exact structure, it is, however, possible to come very close to the original, using the available documentation.

The most characteristic aspect about the first structure of *Übung für Klavier* is the use of masks for the entry delays and dynamic values. Starting very dense, the entry delays gradually get larger towards the end of the structure. As an example, I will show how the entry delays are dealt with in the *CompScheme* model of this structure, the other a parameters are handled similarly. Figure 17 shows first the definition of the entry delay list and secondly the definition of two functions, which return the indices of the lower and upper boundaries of the entry delay selecting tendency mask. The parameter `x` is a position within the structure in percent of the total duration. The function `linear-shape x list` linearly interpolates the `list`, which defines line segments in the following format: $n_1\ start_1\ end_1\ \ldots n_n\ start_n\ end_n$ and returns the interpolated value at position `x` of the specified linear shape.

```
;; list of basic entry delay values
(define *entry-delays*
  '(0.1 0.12 0.15 0.19 0.24 0.30 0.37 0.46 0.58
      0.72 0.89 1.11 1.38 1.72))

;; functions, which return the indices of the
;; lower and upper boundaries of
;; the entry delay selecting tendency mask.
;; the parameter 'x' is in percent of the whole
    structure.
(define (entry-lower x)
  (round (linear-shape x '(20 0 0 27 0 1 20 0 6
      33 6 8))))

(define (entry-upper x)
  (round (linear-shape x '(20 0 6 27 4 6 20 7 10
      33 10 13))))
```

Fig. 17. The user defined masks and basic values for the entry delays

Figure 18 shows the definition of the event generating function. As seen above in figure 17, the tendency masks are relative, i.e. they do not have a fixed number of elements, but maintain their shape for different specified durations of the structure. The goal is thus to define a function, which takes the durations, i.e. the sum of all entry delays, as an argument, and returns the entry delays, while maintaining the shape of specified tendency masks. In order to do that, the entry delay selecting function needs to know its own output (the 'current' time). In *CompScheme*, this can be done by using `st-iterates fun arg`, which returns a streams with the following elements: arg, $(fun\ arg)$, $(fun\ (fun\ arg))$, ...

```
(define (entry-delays duration)
  (st-until (lambda (time) (> time duration))
    (st-iterates
      (lambda (time)
        (let ((percent
                (* 100 (/ time duration))))
          (+ time
             (nth *entry-delays*
               (alea (entry-lower percent)
                     (entry-upper percent))))))
    0.0)))
```

Fig. 18. The entry delay generating function

The duration and the velocity parameter are constructed similarly, but since their values depend on the position in time, i.e. on the entry delays, they do not need `st-iterates`. The generalized selection function for masks in percent is shown in figure 19.

```
(define (selecting-mask material shape-upper
    shape-lower start-times duration)
  (st-apply
    (lambda (idx) (nth material idx))
    (st-rv
      (st-apply (lambda (time)
        (shape-lower (* 100 (/ time duration))))
                  start-times)
      (st-apply (lambda (time)
        (shape-upper (* 100 (/ time duration))))
                  start-times))))
```

Fig. 19. Generalized selection with masks in percent

Figure 20 shows the definition of the final structure generating function. As stated above, one of the powerful consequences of using relative tendency masks is that the final structure can be stretched and compressed in time. Since the function accepts a duration parameter we can produce structures of any length while maintaining the same development, the same musical gesture. Many other parameters, such as duration, velocity, and register depend on the entry delays, this is why we first define a local variable `start`, which contains the entry delays. As a consequence of the above described persistence of streams, no copying of values is necessary and all streams refer to the same sequence of entry delays, despite the indeterminacy in the process of generation. The function `interval-matrix` and `st-interval-matrix` are built-in *CompScheme* functions and deal with the *Project 2* interval transition matrix pitch model, which is not to be discussed here, but described in [5] and [3].

V. Real-Time Control of the *SuperCollider* Server

The sound synthesis and music composition programming language *SuperCollider* underwent a major change in its internal design from version 2 to 3. The so-called SC Server, a powerful synthesis engine, and the *SuperCollider* language have been separated into two separate programs and now communicate with the Open Sound Control (OSC) protocol. *SuperCollider*'s system designer James McCartney writes:

```
(define (structure1 duration)
  (let ((start (entry-delays duration)))
    (st-midi-note
      (start start)
      (duration (selecting-mask *durations*
                  durations-upper
                  durations-lower
                  start duration))
      (velocity (selecting-mask *dynamics*
                  dynamics-upper
                  dynamics-lower
                  start duration))
      (note
        (st-apply
          (lambda (pc mini maxi)
            (pc-alea pc mini maxi))
            (st-interval-matrix
              (interval-matrix
                '(0 4 5 8 9 11))
               (alea 0 11) (alea 1 11))
            (st-apply
              (lambda (time)
                (nth *registers-lower*
                  (registers-lower
                    (* 100 (/ time duration)))
                      ))
                start)
            (st-apply
              (lambda (time)
                (nth *registers-upper*
                  (registers-upper
                    (* 100 (/ time duration)))
                      ))
                start))))))
```

Fig. 20. The structure generating function

> One goal of separating the synthesis engine and the language in SC Server is to make it possible to explore implementing in other languages the concepts expressed in the SuperCollider language and class library. Some other languages that I think may have interesting potential in the future for computer music are OCaml, Dylan, GOO, and also possibly Ruby[...].[8]

CompScheme's control possibilities for the SC Server are not designed to be a replacement for the *SuperCollider* language. Synth definitions (SynthDefs), recording, and routing, for example, must still be done in the *SuperCollider* language, but control and instantiation of synths can be done through *CompScheme*. *CompScheme*'s event model for controlling the SC Server is similar to the Pattern classes and the Pbind synth control (See the *SuperCollider* help files for more information on these classes) in the *SuperCollider* language, in that it allows synths to be scheduled with certain parameters. However, it differs from the Pattern classes in several ways, allowing arbitrarily deep nesting of control streams as the parameters of a synth can be updated within one event, event streams may be directly written into an OSC binary file for non-realtime rendering, and durations and entry delays are always controlled independently. Moreover, the *SuperCollider* event type (SC event) is a regular *CompScheme* event type and can be interpreted, transformed, and written out in numerous ways.

As figure 21 shows, *CompScheme*'s *SuperCollider* synth creating event type has three default parameters: the name of the SynthDef, a starting value, which is an entry delay relative to the previous event's starting time, and the duration. It is important, that the synth will free itself, at latest after the time of the duration has passed, because *CompScheme* manages the synth's IDs, in order to be able to update the synth during an event.

```
(st-sc-event
  (synth "sine1")
  (start 1.0)
  (dur 0.5)
  (<parameter4> <value4>)
  (<parameter5> <value5>)
  ...)
```

Fig. 21. The `sc_event` stream and its default values

A. Simple SC Events

Figure 22 shows a simple synth definition (synthdef), taken from [2], which is to be evaluated in the *SuperCollider* language. The parameters, which can be controlled with *CompScheme*, are the arguments of the synthdef: freq, amp, dur, attack, decay.

```
(
SynthDef("sine1",{ arg freq = 440, amp = 0.2,
  dur = 2.0, attack = 0.25, decay = 0.25;

  var   ssTime = dur * (1 - attack - decay);
  var    attackTime = dur * attack;
  var    decayTime = dur * decay;
  Out.ar(0,
          SinOsc.ar(freq, 0,amp)
                  * EnvGen.kr(
                    Env.linen(attackTime,
                              ssTime,
                              decayTime,
                              1),
                    doneAction: 2)
  )
}).store;
)
```

Fig. 22. A simple synth definition (taken from [2])

Figure 23 demonstrates how to play a SC event stream in real-time. This example also demonstrates the advantage of persistent streams and the power of higher-order functions. In contrast to midi event streams, SC event streams work with relative entry delays, not absolute starting times. This decision has been made to ensure sensible time values for real-time output. In the example shown in figure 23 the starting times are made by a random choice from a list of four values. The value 0.0 stands for simultaneous events (chords). The events last until the next event starts, which is made by using the entry delays of the start parameter and dropping the first value. There is, however, one problem. Due to the chords, events which are followed by simultaneous events will have a duration of 0.0 seconds. In order to ensure that all events last at least 0.1 seconds, a clipping function is applied to the duration stream.

```
(sc-play
  (let ((entry-delays
          (st-random-choice
            '(0.0 0.1 0.15 1.7))))
    (st-sc-event
      (synth "sine1")
      (start entry-delays)
      (dur (st-apply (lambda (x) (max x 0.1))
             (st-drop 1 entry-delays)))
      (freq (st-exprand 100.0 4000.0)))))
```

Fig. 23. Playing a SC event stream

B. Sub-Events

As stated above, one of the powers of *CompScheme*'s SC event type system is that events, which instantiate synths, can update the synth during an event. This means that events can not only represent note-like sound events, but also control updates, within such a sound event. In general, this mechanism works by not supplying a static value or a stream of numbers, but by supplying a *stream of streams*. Every stream in this stream of streams is then seen as a development the parameter has during the respective event. However, the streams inside must be of a certain type, namely `sc_nset`. Figure 24 shows the definition of two auxiliary functions for the creation of a sub-event stream. The first function returns a stream of `st-sum` streams. The second function returns the stream of `nset`-streams we will use in the final output. The `nset` event type holds two values, `start`, which is a starting value relative to the starting point and duration of the parent event, where 0.0 denotes the starting point and 1.0 the ending of the parent event, and `value` which is the respective value used for the update of the synth's parameter. The defined function `stream-nsets` takes three arguments, which will be streams, the number of elements for each sub-event stream, the starting points and the values themselves, which are assumed to be streams of streams.

```
 ;; a stream of streams
(define (sum-streams add start)
  (st-apply st-sum add start))

;; a stream of nset streams
(define (stream-nsets st-n st-start st-value)
  (st-apply
    (lambda (nstr strt vls)
      (st-first nstr
        (st-sc-nset (start strt) (value vls))))
    st-n st-start st-value))
```

Fig. 24. Defining auxiliary functions for sub-events

Figure 25 finally shows how the an `nset`-stream can be embedded. In the example, we create a simple SC event stream, but use the above defined function for the creating a stream of `nset`-streams to control the frequency parameter. The duration of the update streams will be randomly selected between 2 and 5, the starting points of the updates are generated by streams of streams, which all start at 0.0 and increment by a constant addition of a randomly generated value for each event between 0.05 and 0.2. The frequency of each event will thus always start at 800 Hz. The defined function takes the frequency increment per sub-event as an argument, here we call the function with a constant of 100 Hz.

```
(define (sc-nset-stream1 freqadd)
  (st-sc-event
    (start 2.0)
    (dur 2.0)
    (freq (stream-nsets
            (st-rv 2 5)
            (sum-streams (st-rv 0.05 0.2) 0.0)
            (sum-streams freqadd 800.0)))))

(sc-play (sc-nset-stream1 100.0))
```

Fig. 25. Defining and playing a stream with an embedded nset stream

C. Scheduling Event Streams

It is not only possible to extend the event model to lower levels, as described in the previous section, but also to extend it to higher levels. SC event streams themselves can also be scheduled. There is another type of event called `sc_stream_event`, which contains SC event streams and starting times as relative entry delays. In the example in figure 26 a stream of SC event streams is build by mapping the SC event stream returning function `sc-nset-stream1` defined in figure 25 over a stream of random values, which will be interpreted as frequency increments for the sub-event (see previous section). The function `st-sc-stream` schedules the stream of SC event streams, the entry delays are given by the `start` argument. The function `st-sc-stream` can also take further `st-sc-stream`'s. Therefore, there is no built-in limit and scheduled event streams can be scheduled again.

```
(sc-play-stream
  (st-sc-stream
    (start (st-rv 0.0 2.0))
    (sc-stream
      (st-apply sc-nset-stream1
                (st-rv 10 100)))))
```

Fig. 26. Playing a stream of event streams

VI. Stochastic Synthesis

The idea to synthesize sound directly by using musical procedures has been employed by composers of electronic music at least since the early 1950s. Extending the compositional control down to the micro-level, and thus being able to actually compose the sound itself, has not only been part of the basic postulate of the Köln electronic music school, but has also been a general thought in many approaches to computer generated sound until today.

In the 1970s the composers Gottfried Michael Koenig, Iannis Xenakis, Herbert Brün, and others developed systems that abandoned existing acoustic models, and tried

to derive sound synthesis methods directly from compositional activities. Rather than trying to compose with sounds created on the basis of given analytical models, the sound is supposed to be the result of the compositional process itself. In 1970 G.M. Koenig described his program SSP, which was not yet implemented at that time:

> As opposed to programmes based on stationary spectra or familiar types of sounds, the composer will be able to construct the waveform from amplitude and time-values. The sound will thus be the result of a compositional process, as is otherwise the structure made up of sounds. [4]

With SSP, Koenig extended the principles used in his earlier programs Project 1 and specifically Project 2 from the level of the note down to the level of the digital sample. As basic elements amplitude and time values were specified and grouped in segments, in which they were linearly interpolated. For the selection of the basic elements, aleatoric and serial principles were used. SSP may be seen as an attempt to overcome traditional ways of representation that stem from instrumental music, and substitute them with more general descriptions, such as similarity, transition, and variation that are to be applied to the macro-structure of the form as well as to the micro-structure of the sound in one process. This is derived from the axiomatic assumption, that "musical sounds may be described as a function of amplitude over time."[4]

Iannis Xenakis's idea of dynamic stochastic synthesis differs from Koenig's SSP in its initial intentions. The notion of an evolutionary process is central to Xenakis's idea of dynamic stochastic synthesis. In dynamic stochastic synthesis, breakpoints are grouped – here in cycles of a waveform – and linearly interpolated to form an integration of macro- and micro-levels of musical time. Both approaches to stochastic sound synthesis are primarily rooted in music composition, derived from compositional activities and not in the analysis of sound.

> Any theory or solution given on one level can be assigned to the solution of problems of another level. Thus the solutions in macro-composition (programmed stochastic mechanisms) can engender simpler and more powerful new perspectives in the shaping of micro-sounds. [11]

Xenakis, Koenig, and Herbert Brün were motivated by finding ways of producing sound that are idiomatic to the means of production, the computer. Instead of emulating an instrumental or electronic paradigm, the idea of the sample as the basic musical element is inherently digital. Xenakis, Koenig, and Brn used the sample as the basic musical element in a search for "sounds that had never before existed"[11]. Instead of the novelty of sound, the strength of this non-standard approach to sound synthesis lies in its unification of the sound production and compositional processes. It is therefore really one of representation.

In the following, I present a program that is not aimed at reimplementing, but rather an attempt to generalize from Xenakis's and Koenig's systems for stochastic sound synthesis and thus providing the possibilities for extensions. I try to show, that the flexibility and expressiveness of streams lends itself well not only to the description of higher-level compositional processes, but as well to the lower-level sound production. Stochastic sound synthesis is an area of application in which a basic motivation of electronic music, namely *composing sound*, demands a unified representation. This unification of the sound production and the composition process requires a previous relationship between sound and control data. However, most current sound synthesis systems and computer music languages establish a strict separation of synthesis and control data. There are, therefore, hardly any platforms today, that enable experimentation in this area.

In *CompScheme*, rather than considering sound synthesis and composition as two different domains, the same mechanisms are used to describe sound as well as higher-level control. There is no separating wall between sound and control built into the system and no limit to the level of abstraction.

A. A Generalization of Stochastic Synthesis

Both SSP as well as Xenakis's systems group amplitude and time points together, form sequences of these groups, and linearly interpolate the breakpoints. In the case of SSP, these groups – called segments – contain elements selected from initial amplitude and time lists by using Koenig's *selection principles*. In Xenakis's systems, these groups are cycles of one waveform, whose elements are a deviation from the previous cycle's elements, using stochastic processes.

In *CompScheme*, the basic sound synthesis element is the sample, which contains both a time and an amplitude value. A sample is considered an event, just like any other musical event, and can be built and transformed with the same mechanism. Figure 28 shows the function `st-sample` which uses the event type syntax shown in figure 27.

```
(make-event <name>
  (<parameter1> <value>)
  (<parameter2> <value>)
  etc...)
```

Fig. 27. Event stream creation

The example in figure 28 creates a stream of sample events from two streams, one determining the positions of the breakpoints and one that determines their amplitude. The positions in this example are taken from a list of four integers and the amplitudes are chosen randomly between -1.0 and 1.0.

```
(st-sample
  (pos (st-of-list '(0 1 2 5)))
  (value (st-random-value -1.0 1.0)))
```

Fig. 28. A sample stream

Based on SSP, we may call the sample stream of figure 28 a *segment*. A sample's time value denotes its position within the segment to which it belongs. Segments are then collected in a stream – a stream of sample streams – which can be interpolated with an interpolation function and written out into an audio file. A segment can thus be seen as cycle in a process of dynamic stochastic synthesis, or as segment in a collection from which we can select, using a *selection principle*.

B. Example 1: Dynamic Stochastic Synthesis

For a concrete example, we turn towards implementing a simple process close to Xenakis's GENDY. Generally speaking, in GENDY several breakpoints are defined and interpolated in what could be called one cycle of a waveform. The next cycle is a deviation of the previous one. Each breakpoint and time distance follows a random walk and the total length of each cycle is also controlled.

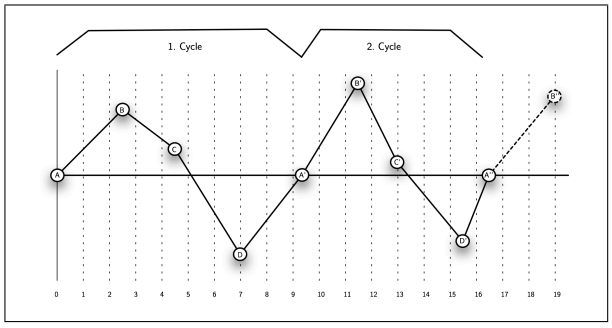

Fig. 29. Two cycles of a dynamic stochastic synthesis process

This process can easily be described in *CompScheme* using the function `segments-with-length`, which takes three arguments: the length of the cycle and two lists of numbers, the first one containing a value for each position and the latter one for each amplitude value. The time values are then scaled to fit inside of the specified length. It then returns a stream of samples; here to be considered one cycle.

```
(define (gendy1)
  (st-apply segments-with-length
    (st-walk 80.0 (st-rv -10.0 10.0) 50.0 250.0)
    (st-apply list
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0)
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0)
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0)
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0))
    (st-apply list
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0)
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0)
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0)
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0))))
```

Fig. 30. The definition of `gendy1`

Figure 30 shows the definition of a function called `gendy1`, that describes a GENDY-like process, which is kept simple for the sake of brevity. The function `segments-with-length` is successively applied to the elements of the three argument streams. The first argument controls the lengths of the cycles, by means of a random walk (`st-walk`) starting with 80, successively adding the elements of the inner stream of random values between -10 and 10 onto its current value, and limited in borders ranging from 50 to 250. The second and third arguments determine the breakpoints's positions within the cycle and are also controlled by random walks. For the sake of brevity, only four breakpoints are made. Before `segments-with-length` is applied, the time points, as well as the amplitude values, are collected in lists. It is to be mentioned, that the returned stream of waveform cycles is infinite. Since the output is a stream, we have not left the high-level description and can easily transform and reuse the created cycles.

Figure 31 shows how to write out the first 10000 cycles of a sample stream into an audio file, using a sample rate of 44100 samples per second.

```
(write-sample-stream "gendy1.wav" 44100
  (st-first 10000 (gendy1)))
```

Fig. 31. Writing out an interpolation into an audio file

One possible extension of GENDY that composer Sergio Luque proposed [7] is the concatenation of several independent GENDYs. Similar to SSP's *permutation* function in which segments are concatenated by using *selection principles*, Luque concatenates waveform cycles from several independent GENDYs. We could easily concatenate several `gendy1`s with the function `st-interleave`, which interleaves the output of any number of streams and forms a new stream as shown in figure 32.

```
(st-interleave (gendy1) (gendy1) (gendy1))
```

Fig. 32. Concatenating three independent cycle streams

Usually stochastic synthesis is implemented in a form, that makes the positions of breakpoints depending on the sample grid. That means, that a breakpoint can only be set at a sample point. A consequence of restricting the positioning of the breakpoints to sample points is that one can only express cycles of durations, which are an integer multiple of the duration of one sample in the chosen sample rate. This limitation imposes a strong frequency grid, which is especially audible with high frequencies. A restriction like this would be considered intolerable in the case of standard oscillators, but it has been often neglected in the discussion of dynamic stochastic synthesis, in favor of reimplementing truthful adaptations of its historic original, including all of its idiosyncrasies. As can be seen in figure 30, the breakpoints's locations are expressed in floats. That means, they can be located in between samples, the resulting wave is then 'sampled' again during the interpolation process.

C. Example 2: A Variation on SSP

The following example demonstrates the use of higher-order functions to create variations of streams. In SSP, one defines segments and then selects an order of the defined segments with a function called *permutation*. In *CompScheme* there is a function called `segment` that takes three arguments: the length of the segment, a stream of relative time distances, and a stream of amplitude values and returns a stream of samples. Figure 33 shows the creation of a stream of variations of segments. The described function `segment` is mapped over three other streams, the first one producing the lengths, the second one a stream of streams produced by varying a stream, and the last argument is a stream of amplitude value streams. This last stream of streams is again produced by a mapping of a function, namely `st-repeat`, which takes two streams one containing the number of repetitions and the other containing the values to be repeated. Thus the variable `segments` contains an infinite stream of segments. Whereas in SSP every segment has to be created 'by hand', here we can easily employ SSP's principles on a higher level and create possibly infinite streams of segments.

```
(define segments
  (st-apply segment
    (st-random-value 5 50)
    (st-apply st-random-value 1 15)
    (st-apply st-repeat
      (st-apply st-random-value 1 10)
      (st-apply st-random-value -1.0 1.0))))
```

Fig. 33. The definition of the segment function

In order to create a *permutation*, we can select segments from the above defined stream by using another stream. Figure 34 shows a possible permutation. Three thousand segments are selected from the above defined `segments` with a tendency mask going from between 0 and 0 to 40 and 60 and an indexing function. Since the deviation among the first elements is smaller then that among the later ones, the output develops from a rather pitched sound to something more noisy.

```
(st-nth segments (st-tendency 3000 0 0 40 60))
```

Fig. 34. Constructing a permutation

ACKNOWLEDGMENT

I would like to express my gratitude to Paul Berg of the Institute of Sonology for his comments, help, and supervision, Gerhard Eckel of the Institute of Electronic Music and Acoustics in Graz for his ideas and help, and Graham Flett for his corrections.

REFERENCES

[1] H. Abelson, G. Sussman and J. Sussman, "Structure and Interpretation of Computer Programs," *MIT Press*, 1996.

[2] P. Berg, "Using the AC Toolbox", *Institute of Sonology*, 2007

[3] G.M. Koenig, "Übung für Klavier", *TONOS Musikverlags GmbH*, 1969

[4] G.M. Koenig, "The use of computer programmes in creating music," *La Revue Musicale*, 1970.

[5] G.M. Koenig, "Project 2, A Programme for Muscial Composition", *Electronic Music Reports*, *Institute of Sonology*, Utrecht, 1970

[6] X. Leroy, *The Objective Caml System: Documentation and user's manual*, *INRIA*, 2007

[7] S. Luque, *Stochastic Synthesis: Origins and Extensions*, Master's thesis, *Institute of Sonology*, 2006

[8] J. McCartney, "Rethinking the computer music language: SuperCollider", *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002

[9] L.C. Paulson, "ML for the Working Programmer," *Cambridge University Press* , 1996.

[10] L. Wittgenstein, "Tractatus logico-philosophicus", *Suhrkamp*, 1960

[11] I.Xenakis, "Formalized Music," *Pendragon Press* , 1992.