The Shepard Tone and Higher-Order Multi-rate Synchronous Data-Flow Programming in SIG

Baltasar Trancón y Widemann
Ilmenau University of Technology, Ilmenau, DE
baltasar.trancon@tu-ilmenau.de

Markus Lepper
semantics GmbH, Berlin, DE
post@markuslepper.eu

Abstract

The total functional real-time data-flow programming language SIG features a core layer with elegant denotational semantics, in terms of Mealy stream transducers and coiterative causal stream functions, that is convenient for domain experts in the primary application domains, such as scientific modeling and digital music and event arts. The core suffices for the implementation of many basic signal processing components. For the expression of more sophisticated computations, a second layer of SIG provides additional features, namely higher-order functional programming and multi-rate synchronicity, reducible by transformational semantics to the core layer. Here we describe the design of the upper layer of SIG and demonstrate its usage with the Shepard Tone, a well-known sound synthesis problem and model of psycho-acoustically paradoxical perception of relative musical pitch.

Categories and Subject Descriptors D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages, Data-flow languages; H.5.5 [*INFORMATION INTERFACES AND PRESENTATION*]: Sound and Music Computing—Methodologies and techniques; J.5 [*ARTS AND HUMANITIES*]: Music

Keywords data flow; real time; sound synthesis; dynamic configuration

1. Introduction

The SIG language is a purely functional, total, clocked synchronous data-flow language. It describes programs for stream processing in *real-time* situations, where input and output may be available and required, respectively, at regular intervals in real time (*online*), and where the length of that interval may be reflected as a parameter of algorithms (*time-aware*). Possible applications range from reactive systems, in a context with input devices, sensors or communication channels, to processors and generators of time-series data, such as audio signals or dynamic simulations in numerous branches of science, engineering and digital arts. The SIG programming system is implemented in Java and currently compiles to the Java virtual machine platform, although alternative low-level backends are a matter of ongoing research.

We have recently demonstrated the effective use of the SIG programming infrastructure for sound synthesis, namely a simple but nontrivial polyphonic organ simulator, calculated from first principles [17]. In the present paper, we discuss extensions of the SIG language for more sophisticated stream processing applications. We describe the extended language design, and demonstrate the use of additional features by tackling yet another sound synthesis problem. In particular, the novel contributions are:

- Higher-order functions are added, a notoriously tricky problem in systems where all variables are time-dependent [19]. We tackle it by an unusual application of multi-stage programming (section 2.)
- The real-time behavior is extended by allowing multi-rate systems, where data flow can occur at different rates, chosen by the context orthogonally to the functional definition, subject to synchronization constraints. We enable this degree of freedom by static analysis and program slicing (section 3.)
- A historically influential sound synthesis problem of medium complexity is implemented as a SIG program, and the language extensions are discussed concretely in terms of their usage in the example code (section 4.)

The presentation of the example gives a preview on the language features we are currently research and implementing in the SIG system. The status of the implementation is summarized in section 5.1; see [15] for more details.

1.1 Some Related Work

Evidently, the design of SIG owes much to the "French school" of synchronous languages such as Lucid Synchrone [3], but takes different turns with regard to high-level features. On the other side, the SIG approach is less abstract and more machine-oriented than the principal contemporary competitors, functional reactive programming (FRP) languages [7]. We have discussed in [16] how the formal semantics of core SIG, derived from first principles, coincide with continuation-based FRP implementations [10]. While (Haskell-based) embedded solutions such as YAMPA [5] certainly have their merits, we aim for a self-contained solution with greater neutrality towards alternative execution models and means, as discussed elsewhere [15]. In particular, the behavioral semantics as well as the concrete implementation scheme of SIG aim at similarity, and ideally practical interoperability, with established systems such as CSOUND, with improved theory and reliability.

1.2 Core SIG in a Nutshell

In a real-time language that is designed to both support effective low-level near-hardware implementation, and expressive high-level symbolic computation, it is a good strategy to do so in a well-stratified way, by providing a hierarchy of language layers with

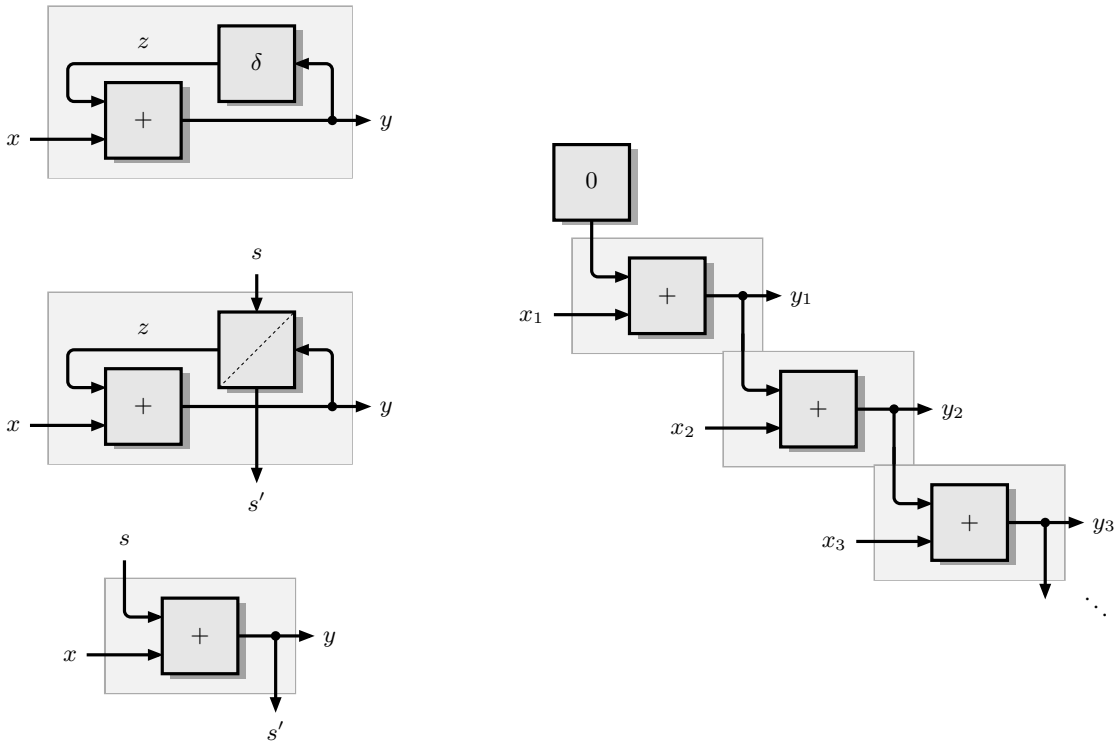

Figure 1. Cumulative sum component. *Top left:* mostly element-wise computation, before delay elimination (initial value abstracted); *center left:* during delay elimination (pair of identities); *bottom left:* Mealy step operation, after delay elimination (and copy propagation); *right:* coiterative stream semantics (initial value re-attached).

appropriate balance of power and requirements at each of various levels of abstraction, as has been demonstrated for HUME [6].

The core layer of SIG is a simple functional programming language [16]. Functions are pure and total; hence observable side effects, spontaneous events such as messages or interrupts, partially defined operations and general recursion are conspicuously absent. All variables are time-dependent; streams arise not as data structures, but implicitly as the consecutive values of a variable at equidistant clock ticks. Computations are notated as if instantaneous, and indeed most operations are stateless and simply repeated independently at each clock tick. For example, consider the following expression defining a component that computes the element-wise average of two input streams, `x` and `y`, as its output stream:

```
avg = {x, y -> (x + y) / 2}
```

The curly bracket is SIG's lambda abstraction. The SIG type system, which plays only a marginal role in the present paper, is ignored for the moment for the sake of simplicity. Stream computations beyond the element-wise can be expressed by the special operator `(;)`[1], which delays a stream by exactly one clock tick, prepending a given initial constant[2] value. For example, consider the following expression defining a component that computes the backward difference transform of its input stream:

```
diff = {x -> x - (0 ; x)}
```

As the only exception to the general ban on recursion, delayed feedback loops are allowed. However, such loops are at odds with the anonymity of function results in the lambda-style notation. Therefore, SIG provides a second abstraction bracket, which names inputs and outputs symmetrically, and has a body that specifies the internal data-flow graph as a list of (static single) assignment statements. For example, consider the following expression defining a component that computes the cumulative sum of its input stream:

```
csum = [x -> y where y := (0 ; y) + x]
```

This latter "box abstraction" is privileged in core SIG, because it is more general and handles multiple results and non-tree-shaped data flow more gracefully, although more verbosely. Since many of the following examples have only a single assignment statement, we take the liberty to abbreviate to an intermediate form that expresses feedback with minimal syntactical noise:

```
csum = [x -> y := (0 ; y) + x]
```

Delay as a compositional add-on to the operator set increases the domain of definable stream functions from element-wise to *causal* functions: The output values for each clock tick may depend on past and present, but never on future, input values. Furthermore, delay operations can be translated to state variables of a Mealy[3] state transducer [9, 16]. Since access to past values is only via delay, no implicit storage of stream elements is required. As a result, all algorithms expressed in SIG can operate *online* on streams realized as real-time mutable variables.

Delay as a syntactic building block is eliminated by conversion to two-dimensional data flow; not only from input to output, but also from pre-state and to post-state. Each delay operator in the network takes the form of a pair of cross-over equations, namely equating its pre-state and output, and independently its input and post-state. Thus delayed feedback loops disappear; if any (undelayed) circular

[1] Variously written >> or `fby` in other synchronous languages.

[2] Or relatively constant; see sections 2.2 and 4.1 below.

[3] Moore machines would be more tolerant to feedback, but at odds with instantaneous element-wise computation (*map*) on streams.

data flow remains, the expression is invalid. See [16] for the full formal specification of the elimination procedure.

For example, consider the graphical depiction of the cumulative sum component in Figure 1, top left. The delayed expression `0 ; y` has been abstracted from its initial value and represented as $z := \delta(y)$. I/O data flow is from left to right. Introduction of state variables s, s' yields the intermediate form depicted below, center left. The pseudo-operator δ is replaced by a pair of identities: $z := s$ and $s' := y$, respectively. State transition data flow is from top to bottom. Simplification by the delay-derived identities (*copy propagation* in program transformation parlance) yields the final step form depicted below, bottom left. The global semantics of this element-wise step operation as a stream transformation is obtained by coiteration; it can be conceived as replication of the step operation, ω times along the state axis. The initial value of each delay, previously abstracted, is re-attached as the respective pre-state of the first step. The result is a causal function between synchronous infinite I/O streams; see Figure 1, right.

1.3 The Shepard Tone

The Shepard Tone has been described as a scale progression [13], and later as a continuous tone, that challenge simple psychoacoustic models of relative musical pitch perception. It has been cited as the acoustic equivalent of Penrose's and Escher's impossible circular graphical configurations, or the barber-pole illusion. It has been chosen here as an example, because the stratified nature of its definition evokes high-order programming as the natural means for its implementation. See the appendix for a description of the supplementary audio demonstration file.

The tone is a composite sustained sound, created by superposition of an ensemble of oscillators. The individual oscillators are spaced evenly in pitch, a logarithm of frequency, at a harmonic interval ivl, such as an octave. The pitch of all oscillators increases linearly over time, at a stepping rate t_2^{-1}, such that each traverses the interval ivl in a given period $t_1 \gg t_2$. The proportional resolution parameter $res = t_1/t_2$ is tunable to the available computational resources and desired smoothness, varying from a discrete scale for long t_2, to a smooth (Shepard–Risset) glissando for short t_2. [12]

With recurrence rate t_1^{-1}, the currently highest oscillator reaches the maximum pitch $p_{\max}$ and is switched off; likewise a new oscillator is introduced at minimum pitch $p_{\min}$, ivl below the currently lowest oscillator. To keep these two kinds of switching events synchronous, and the number of live oscillators constant, the total pitch range should be divisible by the spacing interval: $p_{\max} - p_{\min} = (2r + 1) \cdot ivl$, where r is a non-negative integer. See Figure 2, where $r = 1$ for simplicity; high-quality realizations typically have $r \geq 4$.

The ensemble of live oscillators can be thought of as a shift register of size $2r + 1$. The switching/shifting events occur at a rate on the order of one in several seconds, but the Shepard effect relies on them being imperceptible as discrete events. To this end, the amplitude envelope of oscillators attains the maximum in the central interval only, and tapers off gradually, that is synchronously with pitch at rate t_2^{-1}, towards the ends of the pitch range, such that switching occurs below the audible threshold. See Figure 3. The overall effect of the tone creates the conflicting perceptions of smoothly *increasing* pitch on short, $\mathcal{O}(t_2)$, time scales, but *stationary* pitch on long, $\mathcal{O}(t_1)$, time scales.

The switching regime as described above is actually necessary for the effect to manifest properly; it is not sufficient to play a loop, say, of the highlighted rectangular clip in Figure 2: The relative phases of the oscillators, being integrals of frequency, a transcendental function (exponential) of pitch, get out of synch during one iteration of the loop necessarily. It is hence impossible to cut the clip in a way that maintains phase continuity across loop boundaries (along diagonal lines in Figure 2); any cut results in an audible click.

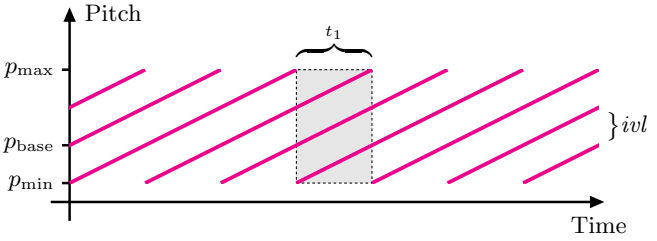

Figure 2. Shepard oscillator ensemble pitches as multi-valued function of time.

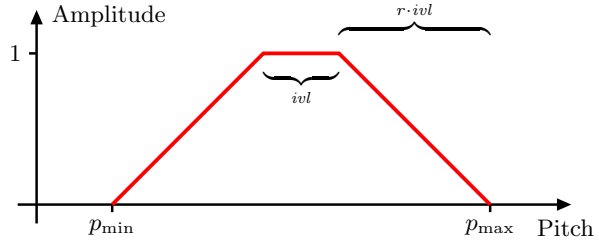

Figure 3. Envelope of individual Shepard oscillator as function of pitch.

Of course it is conceivable to evade actual higher-order programming by representing the oscillators indirectly by their state tuples; but we consider higher-order computation with oscillator components, with the respective phase encapsulated as private state, a more adequate representation from a software engineering viewpoint, with respect to information-hiding modularization, and hence a more expressive and scalable methodology.

1.4 Temporal Considerations

The Shepard Tone illustrates non-trivial real-time structures that are characteristic for advanced sound synthesis applications:

Digital audio signal-processing systems traditionally operate synchronously but heterogeneously at several distinct rates, mandated by the combination of aesthetical and technological requirements.

The highest rate is the *audio* rate, which should be well above the Nyquist rate of human frequency perception, in order to avoid aliasing artifacts. For instance, the standard CD consumer audio rate is 44.1 kHz, whereas digital sound studios operate at 48 kHz or 96 kHz. Waveform generator components operate at this rate, and must be kept extremely simple in all but the most powerful implementations, because of the considerably hard real-time bounds on the order of a few microseconds per sample.

Conceptually continuous behavior that is computationally more expensive than a handful of CPU cycles is implemented by components operating at the *control* rate, which should therefore be below the audio rate, but well above rates that human perception resolves as discrete events. For instance, control rate may be fixed at 1/128 of audio rate in low-level implementations with hardware binary counters, or at 1 kHz in software-intensive systems.

Additional, lower rates may be called for, either technically by different effort–smoothness trade-offs, or aesthetically by musical beat quantization and long-scale behavior. (For example, the switching events in the Shepard ensemble.) Finally, computations at *initialization* time of a complex system can be conceived as operating at *zero* rate, for the sake of a semantically clean, uniform and reliable approach to initialization, a traditional weakness of the relevant *practical* audio programming systems.

Components operating at different rates need to interact without breaking the synchronous paradigm. In classical, first-order programming approaches, the interaction is by data only, and can be conceived as *resampling* of a signal. Two ubiquitous patterns can be discerned: Slow components control parameters of faster ones (*modulation*), where the parameter streams are *upsampled*. (For example, the pitch and amplitude parameters of Shepard oscillators from control rate t_2^{-1} to audio rate.) Conversely, fast components report statistical feedback, such as energy balance or peak amplitude (*aggregation*), where the statistic streams are *downsampled*. (No example in the Shepard Tone.)

By contrast, in a higher-order programming approach, there is the additional possibility of slow components periodically incarnating faster ones (*configuration*), where a function-valued stream is *upsampled* to the application rate and its elements switched dynamically. (For example, the Shepard oscillator ensemble at rate t_1^{-1}.) We believe that the corresponding freedom of expression is valuable and natural for many real-world programming problems in the application domains of interest, and have made support for higher-order multi-rate programming a top priority in the development of SIG.

In the remainder of the present paper, we discuss the current state of design and implementation of the concerned language features, and illustrate their usage with an executable definition of the Shepard Tone.

2. Higher-Order Synchronous Data Flow

In SIG, by contrast to many FRP approaches, there is no type-level distinction between time-less (element) and time-dependent (stream) values. All variables are uniformly understood as referring to real-time streams operating at individual but fixed rates. Constant element values fit the general picture as the corner case of a zero rate stream, such that the second element is already deferred infinitely, and only the first element can ever be observed.

This all-streams policy is consistent with simple "clocked hardware" metaphors for SIG semantics, and hence beneficial for intuitive program understanding by domain experts. However, it comes at the price of interfering awkwardly with higher-order functional programming and in particular nested function expressions. Consider the following higher-order component, which simply exhibits Currying of a primitive operation:

```
add = {x -> {y -> x + y}}
```

For the sake of precision, consider also the equivalent but more verbose SIG box notation, which names all streams explicitly:

```
add = [x -> f := [y -> z := x + y]]
```

The intended semantics, according to the principle of least surprise, is that add maps a stream of numbers x_i to a stream of unary functions $f_i = (\_ + x_i)$; hence f_i maps a stream of numbers y_j to a stream of numbers $z_j = x_i + y_j$. But there is a catch:

Evidently, streams `x` and `f` are synchronized, and so are `y` and `z`. But if the dependency of `z` on `x` (via supposedly synchronous `+`) is not to break the paradigm, then the rates of all four must actually be identical. This has the paradoxical result that, by transitivity, `f` and `y` must be synchronized, that is, functions are produced and used exactly as quickly as the elements of their input streams, which rather defies the purpose of stream-processing functions (one function, potentially infinitely many inputs and outputs).

This example shows that free variables in nested function expressions in the time-less lambda calculus do not generalize straightforwardly to free variables in nested stream-processing component expressions in time-dependent data-flow programs. Note that the most elegant existing proposal, given by Uustalu and Vene in terms of the rather arcane categorial language of symmetric semi-monoidal comonads [19], does not help: Their approach, "the present value of a function application is the present value of the function applied to the history of the argument," is unsuitable for recursion-free online and real-time calculation; when it comes to the present, alas, the history is past and cannot be replayed.

Fortunately, there is a well-established, sufficiently time-aware functional programming technique that we can employ.

2.1 Staged Meta-Programming

The *staged* approach to meta-programming [14] is halfway between two classical language technologies, *hygienic macros* and *closures*. Compared to macro systems, stages emphasize dynamical use and orthogonal integration with the type system and program reductions. Compared to closure conversion of nested functions, stages give greater control over evaluation strategies. See [4] for a detailed comparison. Their typical applications are the definition of custom control constructs, as well as in-the-loop programming with partial evaluation, such as in adaptive algorithms, or (Futamura-style) compilers [14]. More recent applications, closer to our goals, include the dynamic configuration of high-performance computations [8].

In general, staging adds up to four operators to the language. We discuss the general idea using the syntax of METAML [14] for easy reference, and present the adaptation to SIG in the next subsection.

The *suspend* operation $\langle\,\rangle$ defers the evaluation of its argument expression until the *next stage*. If e has type t, then $\langle e\rangle$ has type $\langle t\rangle$. Free variables of e are fixed to the creation-time environment of the suspension (*cross-stage persistence*). The similar *lift* operation does evaluate its argument expression at the current stage, and just suspends the resulting value as a constant. All values of type $\langle t\rangle$ are suspensions of some computations of result type t.

Conversely, the *run* operation enters the next stage, thus evaluating its argument expression to a suspension, and then the content of that suspension as well. If e has type $\langle t\rangle$, then $\mathrm{run}\, e$ has type t. The similar *splice* operation ˜ escapes from a stage; it is evaluated at creation time of the surrounding suspension (as opposed to *run* which obeys the normal evaluation strategy). The resulting value, which is also a suspension, is spliced into the parent suspension in place of the escaped operation. Free variables in $\tilde{}e$ must not be used in a way that violates stage causality; they may not be evaluated at an earlier stage than they are bound.

2.2 Staged SIG

Staging can be used to solve the problem of unintentional higher-order synchronicity, by stipulating the rather plausible axiom that *suspension stops time*.

In METAML, suspension is an orthogonal program construct. For instance, the inner application in $\lambda xy.\,(\lambda z.\,z)z$ can be forced by rewriting the expression to run $\langle\lambda xy.\,\tilde{}((\lambda z.\,z)\langle x\rangle)\rangle$ [4]. By contrast, we deliberately couple suspension and nested function abstraction in SIG, and hence staged and higher-order programming. In the examples discussed here, all nested functions are suspended[4], and hence their free variables are cross-stage persistent. The above axiom requires that cross-stage variable references are not understood as the underlying streams, but as *constant* snapshots of the current values at creation time of the suspension.

SIG uses the prefix operators `#`, `$` and `%`, for *suspend*, *splice* and *run*, respectively. Since suspension is only applied to function abstractions, which are not normally evaluated, *lift* is redundant. Thus the curried addition example becomes:

```
add = [x -> f := #[y -> z := x + y]]
```

Here the inner reference to `x` is cross-stage persistent and retains the constant value that the outer stream `x` had at creation time of the corresponding suspended function value of `f`. This gives the

[4] Future research on synchronous nested functions notwithstanding.

desired semantics, and removes any spurious necessity to have the function elements of `f` synchronize their I/O streams with stream `x`. The corresponding user-level expression `add(a)`, for some input stream `a`, is a first-class citizen of the language, and can be applied using either *run* or *splice*, depending on the desired evaluation strategy: A splice operation is evaluated in the context and at the rate of suspension *creation*, and the denoted function is applied persistently to its argument streams in the context of suspension *evaluation*. By contrast, a run operation is evaluated in the context of suspension evaluation, and creates freshly initialized functions at that rate. Operational differences aside, for the example the two are denotationally equivalent, because `add(a)` is stateless.

Each application site instantiates the element components freshly with non-shared private state, such that referential transparency is ensured [15].

Staging is reflected in the SIG type system. The preceding definition, annotated as would be required for a fully explicit SIG program[5], could be given the following types:

```
add = [x : real -> f : #[real -> real]
       where f := #[y : real -> z : real
                    where z := x + y]]
```

3. Multi-Rate Components

The semantics of SIG programs is particularly simple if all streams are operated at a single, known clock rate. The whole program can be conceived as a monolithic Mealy machine. However, for many realistic applications, this simple account must be refined. Currently we consider two directions of generalization to multi-rate computation, allowing for both distinct and fixed but unknown rates.[6]

The first direction of generalization is analogous to the concept of *polymorphic* functions: A component definition may be instantiated multiply, and the instances operated at different clock rates. We propose to call such components *polydromic*. Since the SIG execution model understands components as passive routines, to be invoked periodically at the desired rate by a scheduler, there is no fundamental technological obstacle to polydromic implementations. In fact, the rate of components is orthogonal to the type system, and inferred from the global real-time usage context, which we refer to as the *driver*. However, the function of a component may depend on its clock rate. For example, consider the following variant on the cumulative sum, namely a properly time-scaled discretized integrator:

```
dint = [ x -> y := (0 ; y) + x * dt ]
```

The value of the context-dependent constant **dt** is the inverse of the rate of its evaluation.[7] This value needs to be identified, either at component runtime by information provided by the scheduler, of preferrably at component creation time by information provided by a static rate analysis. Confer the *dictionary* approach to Haskell type class polymorphism, where bindings of context-sensitive operations are passed as arguments, and partially evaluated where possible.

The second direction of generalization has no counterpart in time-less functional programming. It is concerned with components whose distinct variables (input, output, state and local) may be operated at different rates. In the synchronous paradigm, that degree of freedom is subject to strict constraints.

As a first approximation, all primitive operators, including delay, are considered fully synchronous. Hence all variables connected by data flow must be in the same rate equivalence class. But if a component splits into data-independent subsystems, then these may well be operated at different rates at the discretion of the local invocation context.

Disciplined support for a limited form of asynchronous, inter-rate data flow can be added behind the scenes without breaking the paradigm. Special primitive operations **upsample** and **downsample** are added to the higher layer of SIG. Instead of rate equalities, they introduce either *inequations* if used qualitatively (for instance, `y := upsample(x)` implies that the rate of y is greater than or equal to the rate of x), or *proportions* if used quantitatively (for instance, `y := upsample(x, 5)` implies that the rate of y is five times the rate of x). All resampling operations can be thought of as implemented by communication buffers. It is the responsibility of the scheduler to resolve read–write conflicts in a well-defined way. This is appears viable for *directed* resampling operations such as the above, but not for completely unspecified rate conversions; therefore the latter are forbidden in SIG.

Ultimately, a whole program must meet the requirements of a real-time context that drives the rates of global I/O streams. It can fail to do so in two distinct ways: Firstly, the actual rates of global I/O can be inconsistent with the internal constraints incurred by directed resampling. For instance, if an audio system employs upsampling from formal control rate to formal audio rate, then it is illegal to drive it with an actual control rate that is higher than the actual audio rate. This mode of failure depends on the program and context only, but not on the cost model of the execution platform. Secondly, a real-time program can of course also fail because the required processing rates cannot be met with the given computational resources. Established methods of worst-case execution time analysis should be employed to check for this second mode of failure. We expect that an approach similar to the one presented in [1] in the context of STREAMIT, but much leaner due to the simplicity of core SIG, should be viable.

3.1 Multi-Rate SIG

In the SIG programming system, multi-rate components are supported through the following transformational procedure:

In the first step, a static analysis of rates is performed. It spans a rate constraint system over a finite set of program *locations*. Locations are defined inductively: There is a location for each program variable. For a location of (suspended) function type, there is a location for each I/O variable (parameter), of the respective type. SIG has preliminary support for array data types, which is used also in the Shepard example. For a location of array type $[t]$, there is a location of the element type t, the static abstraction of array elements at arbitrary index. The static approximation via locations implies a restriction on acceptable programs. Namely it is assumed that all function values attained by a function-typed variable over time, and all function elements of an array, operate their respective I/O streams at identical rates. Examples so far suggest that this is a reasonable restriction. Rate constraints are assigned to all operations in the program, either equations for synchronous operations or inequations or proportionalities for resampling operations, respectively, as described above.

In the second step, a factorization of locations into rate equivalence classes is performed. Locations connected by synchronization equations are lumped into a single class. Apart from direct synchronization, spurious rate equivalences can arise from circular inequations, such as mutual upsampling. These are likely programmer mistakes and hence raise a warning. For the whole program, the rates of all variables should be determined uniquely, via equations and proportionalities, by the global I/O variables. Internal degrees of rate freedom leave the behavior of the program in real-time context underspecified, and hence raise an error. Streams of *void* type (*clock*

[5] Future research on type inference notwithstanding.

[6] We leave non-constant rates for possible future research.

[7] Put differently, it is specified that `csum(dt)` yields component life time.

signals) can be passed explicitly to drive internal rates. Furthermore it is recorded which rates need to be identified at runtime, via the **dt** operation.

In the third and final step, the component is sliced into a family of subcomponents according to the equivalence classes of variables, deliberately breaking the resampling data flow [18]: Each resampling operation is split into a pair of read-only and write-only variables, respectively, which are connected behind the scenes as aliases for a shared asynchronous communication buffer. For example, consider the following amplitude-modulated oscillator:

```
amo = #[ amp -> out where
         out := upsample(amp) * sin(phase)
         phase := 0 ; phase + velo * dt ]
```

where the angular velocity `velo` is a cross-stage persistent parameter. The static rate analysis obtains the following constraint system:

$$\mathcal{R}(amp) \leq \mathcal{R}(out) = \mathcal{R}(phase)$$

The finest factorization, which avoids all spurious synchronization, yields two classes, $C_1 = \{amp\}$ and $C_2 = \{out, phase\}$. Slicing results in a pair of subcomponents

```
amo₁ = #[ amp -> amp⁻ where
          amp⁻ := amp ]
```

and

```
amo₂ = #[ amp⁺ -> out where
          out := amp⁺ * sin(phase)
          phase := 0 ; phase + velo * dt ]
```

that communicate the resampled variable amp via a shared buffer, apparent as a pair of eponymous variables $amp^{\pm}$. The implied "anionic" data flow from amp^- to amp^+ through the buffer, is performed behind the scenes.

The scheduling policy prescribed by SIG for the resolution of read–write conflicts on resampling buffers states that for upsampling, writes occur before reads, and for downsampling vice versa. That is, if the ticks of two clocks with different rates coincide, and the slower one writes to an upsampling buffer, then the change (modulation) is propagated to the faster reader instantaneously. This is also necessary for the base case of zero rate streams to function as constants. Conversely, a slow reader observes not the instantaneous value (aggregation) produced by a faster writer, but the previous one. The incurred delay is negligible at the slower rate.

On the whole, this resampling rule allows for a robust non-circular scheduling strategy to be devised based on the system rates alone, regardless of actual inter-rate data-flow paths.

There is currently no automatic code generation support for the scheduler that embeds SIG components into their real-time execution context. Hence the automatic checking of rate constraint satisfaction is not yet an issue. However, components do perform inference of internal rates from the dynamic global rate identification data provided by the hand-coded scheduler, and detect inconsistencies at initialization time.

4. Implementation of the Shepard Tone

Figure 4 shows a SIG program for the Shepard Tone. The following section is a walkthrough that relates the code fragments to the preceding descriptions and arguments. Some aspects of the program could be formulated differently, using language features in alternative ways. We have preferred illustration of many different useful programming patterns over uniformity. Discussion points are located in the source code by giving line numbers in parentheses. All variables are named explicitly and uniquely for easy reference.

4.1 Code Structure

On the whole, the program defines a sound generator in a two-stage process (1, 5, 21–26). The second-stage result `shepard_2` is a component with a clock signal input `clk_1` that drives the period t_1, and an audio signal output `s_out` (5, 21). The various parameters of the Shepard construction are given as first-stage inputs (2–4). The intended rate of the first stage is zero; it serves only to configure and initialize the sound generator properly. The other nested definitions are auxiliary subcomponents (7–20). We shall discuss them in conceptual bottom-up, textual top-down order.

The subcomponent `osci` (7–13) is a two-stage definition of an amplitude- and frequency-modulated oscillator. The first stage fixes the initial phase `init_phase` (7). The resulting second stage `osci_2` takes two inputs streams, `amp` and `freq`, to linearly modulate the phase change (10) and wave amplitude (11), respectively. Note that, by indepent upsampling, the two parameters could vary at distinct rates if desired. Note also that the initialization of the private `phase` state by an input rather than a static constant (10) is only causally legal because they occur at different stages. The component produces an audio signal `o_out` (11), by applying a spliced incarnation of the cross-stage persistent component value `wave`, understood as a periodic normalized function of period 1, such as $\sin(2\pi \cdot \_)$.

The next-level subcomponent `voice` (14–20) is a two-stage definition of the intended control over an oscillator throughout its linear course across the pitch range. The first stage fixes the initial pitch and rate of change, `ascent` (14). The resulting second stage `voice_2` takes a clock input `clk_2` that drives the pitch-stepping period t_2 (15). Its private `pitch` state increases according to the specified `ascent` (17), annotated using the operator `@` to be synchronous with the (value-less) clock signal. The audio output is provided by a spliced-in oscillator obtained by `osci`, which is escaped twice in correspondence with its two-stage definition, fixing the initial phase to zero in the process (18). Note that splicing, rather than running, ensures that the oscillator maintains its internal phase state as long as the enclosing component lives. The modulating amplitude is calculated using the pitch-to-amplitude global parameter function `envelope` (18), which should give a shape such as the one depicted in Figure 3. The pitch (logarithmic relative frequency) is converted to a modulating absolute frequency, offset by the global parameter `base_freq` (18).

The ensemble of ocillators is managed by the second stage of the top level component, `shepard_2` (21–26). It takes a clock input `t_1` that drives the life-cycle period t_1 (21). Its output is a polyphonic audio signal `s_out`, obtained by mixing (summing) the outputs of the `ensemble`, a private state variable holding an array of live oscillators (21, 25). The use of the array as a function is implicitly vectorized, that is, each component in the array simultaneously receives the given input, namely the clock signal `clk_1` upsampled by factor `res`, and the outputs are collected as an array, to be supplied to the **sum** operation (25).

The `ensemble` is annotated to be updated at the clock rate of `clk_1`. Its initial value is an array obtained by mapping the auxiliary function `make` (implicitly vectorized) over an array of the integers $-r$ to $+r$, converting the relative pitches, measured in units of global parameter `ivl`, to oscillators (24). The update at each clock tick shifts the array one element to the right, thus discarding the topmost oscillator, and adding a fresh one at position $-r$ (24). The auxiliary function `make` converts the integral pitch position `k` to an autonomous oscillator obtained by `voice`, scaling the pitch by `ivl` and setting the rate of ascent such that the unit interval takes precisely t_1 to traverse (23). Note that `make` is run rather than spliced in: fresh incarnations are produced at the rate of `clk_1`. This idiom is to emphasize that `make` is stateless, and produces identical values for all elements of the constant stream `-r`.

```
1  shepard = [
2      wave, envelope : #[real -> real],
3      base_freq, ivl, res : real,
4      r : int
5    -> shepard_2 : #[void -> real]
6    where
7      osci := #[ init_phase : real ->
8        osci_2 := #[ amp, freq : real -> o_out : real
9          where
10           phase := init_phase ; phase + upsample(freq) * dt
11           o_out := upsample(amp) * $wave(phase)
12       ]
13     ]
14     voice := #[ init_pitch, ascent : real ->
15       voice_2 := #[ clk_2 : void -> v_out : real
16         where
17           pitch @ clk_2 := init_pitch ; pitch + ascent * dt
18           v_out         := $($osci(0))($envelope(pitch), base_freq * exp(pitch))
19       ]
20     }
21     shepard_2 := #[ clk_1 : void -> s_out : real
22       where
23         make             := #[ k : int -> m_out := $voice(k * ivl, ivl / dt(clk_1)) ]
24         ensemble @ clk_1 := %make(seq(-r, +r)) ; shiftr(%make(-r), ensemble)
25         s_out            := sum(ensemble(upsample(clk_1, res)))
26     ]
27 ]
```

Figure 4. SIG implementation of the Shepard Tone.

$wave$, $wave.i_1$, $wave.o_1$, $envelope$, $envelope.i_1$, $envelope.o_1$, $base\_freq$, ivl, res, r, $shepard_2$, $shepard_2.i_1$, $shepard_2.o_1$, $osci$, $osci.i_1$, $osci.o_1$, $osci.o_1.i_1$, $osci.o_1.i_2$, $osci.o_1.o_1$, $init\_phase$, $osci_2$, $osci_2.i_1$, $osci_2.i_2$, $osci_2.o_1$, amp, $freq$, o_out, $phase$, $voice$, $voice.i_1$, $voice.i_2$, $voice.o_1$, $voice.o_1.i_1$, $voice.o_1.o_2$, $init\_pitch$, $ascent$, $voice_2$, $voice_2.i_1$, $voice_2.o_1$, clk_2, v_out, $pitch$, clk_1, s_out, $make$, $make.i_1$, $make.o_1$, $make.o_1.i_1$, $make.o_1.o_1$, k, m_out, $m\_out.i_1$, $m\_out.o_1$, $ensemble$, $ensemble[]$, $ensemble[].i_1$, $ensemble[].o_1$, s_out

Figure 5. Locations for static rate analysis.

$$\mathcal{R}_1 : \{clk_1, ensemble, \underline{ensemble[]}, \dots\}$$
$$\mathcal{R}_2 : \{clk_2, amp, freq, pitch, \underline{ensemble[].i_1}, envelope.i_1, envelope.o_1, \dots\}$$
$$\mathcal{R}_3 : \{phase, o\_out, v\_out, s\_out, \underline{ensemble[].o_1}, wave.i_1, wave.o_1, \dots\}$$
$$\mathcal{R}_1 \le \mathcal{R}_2 \le \mathcal{R}_3 \qquad \mathcal{R}_1 \cdot res = \mathcal{R}_2$$

Figure 6. Results of static rate analysis.

4.2 Rate Analysis

Locations arise inductively as discussed above; see Figure 5 for a list in order of textual appearance. Basic locations are named after the variables they represent. Locations induced by a function-typed location ℓ are named $\ell.i_k$ or $\ell.o_k$ for the k-th input or output, respectively. Locations induced for the elements of an array-typed location ℓ are named $\ell[]$.

Constraints follow from data flow and special operations as discussed above. For instance, line (11) implies that

$$\mathcal{R}(amp) \le \mathcal{R}(o\_out) = \mathcal{R}(wave.o_1)$$
$$\mathcal{R}(wave.i_1) = \mathcal{R}(phase)$$

Line(15) implies that

$$\mathcal{R}(voice.o_1) = \mathcal{R}(voice_2)$$
$$\mathcal{R}(voice.o_1.i_1) = \mathcal{R}(voice_2.i_1) = \mathcal{R}(clk_2)$$
$$\mathcal{R}(voice.o_1.o_1) = \mathcal{R}(voice_2.o_1) = \mathcal{R}(v\_out)$$

Line (25) implies that

$$\mathcal{R}(s\_out) = \mathcal{R}(ensemble[].o_1)$$
$$\mathcal{R}(ensemble[].i_1) = \mathcal{R}(clk_1) \cdot res$$

The results of the (straightforward but manually laborious) analysis are summarized in Figure 6. For simplicity, locations that reduce to others trivially by data flow, such as $shepard_2.i_1$ to clk_1 by line (21), are omitted from equivalence classes. Locations that operate at rate zero, if all inputs and output of the global component `shepard` are driven at rate zero as discussed, are omitted. Finally, it is assumed tacitly that the inputs and outputs of the auxiliary components `wave` and `envelope`, which occur as top-level parameters only, are synchronized. As a consequence, only three equivalence classes remain, corresponding to rates t_1^{-1}, t_2^{-1} and audio rate, respectively. They are displayed together with their relationships in Figure 6.

Due to rate consistency checking, the second stage will fail to run if $res < 1$. Note that this is not an exception to the totality of SIG, which applies within a stage only. If the rate constraints are met and the second stage can be run, then $\mathcal{R}_1$ and $\mathcal{R}_3$ are fixed by the context to t_1^{-1} and audio rate, respectively, and $\mathcal{R}_2$ is fixed by proportionality. No loosely rated internal streams remain.

Of particular note is the occurrence of the location $ensemble[]$ and its children $ensemble[].i_1$ and $ensemble[].o_3$, in the three different rate classes; see underlining in Figure 6. These are the

```
#[ clk_2 : void -> amp⁻, freq⁻ : real
  where
    pitch @ clk_2
          := init_pitch ; pitch + ascent * dt
    amp⁻  := $envelope(pitch)
    freq⁻ := base_freq * exp(pitch)
]
```

Figure 7. Effective ensemble member, slice for rate $\mathcal{R}_2$ (control).

```
#[ amp⁺, freq⁺ : real -> v_out : real
  where
    phase := init_phase ; phase + freq⁺ * dt
    v_out := amp⁺ * $wave(phase)
]
```

Figure 8. Effective ensemble member, slice for rate $\mathcal{R}_3$ (audio).

witnesses for the rates of configuration, modulation and audio signal generation, respectively, of the Shepard Tone.

For the effect of per-rate program slicing, consider the elements of the `ensemble` array, represented jointly by the location $ensemble[]$. The components themselves are shifted through at rate $\mathcal{R}_1 = t_1^{-1}$. Each of them is sliced into a pair of subcomponents with rates $\mathcal{R}_2$ and $\mathcal{R}_3$, and dealing with input and output, respectively. The slices, with the effect of the splicing-in of `osci` emulated by source-level inlining, are depicted in Figures 7 and 8, respectively.

5. Conclusion

We have sketched the design, syntax and semantics of SIG, discussed in detail in earlier publications [15–18]. We have also described the Shepard Tone as an audio synthesis problem from half a century ago [13]. It showcases realistic problems in advanced signal processing system structure that remain challenging for contemporary programming approaches.

Current work on the design and implementation of SIG is concerned with features required for the natural expression of these kinds of structures, namely dynamic program-controlled reconfiguration of the data flow network by higher-order programming, and synchronous computation at multiple rates.[8]

Semantic interferences of the two features have been identified, and solved with a novel idiom of staged meta-programming. This couples staging and higher-order functions; hence it is less expressive than stage-anywhere languages such as METAML, but generalizes safely to multi-rate stream computations.

The meaning of multi-rate components is specified in terms of rules for robust scheduling and the static analysis of operation rates. Analysis results are used to detect inconsistencies with the real-time environment, and for a slicing transformation that reduces multi-rate code to core SIG, up to implicit inter-rate communication buffers.

The semantics of the new features is given in transformational form, by reduction to the core language. Denotational semantics would obviously be helpful to study the theoretical properties of the solutions. Because of the lack of precedents in the concise formalization of multi-rate systems, this is a major task that we have to leave for future work.

5.1 Implementation

We have demonstrated all of the new features at work on a SIG implementation of the Shepard Tone. Their realization in the SIG programming system is currently under construction: Compiler support for rate analysis and slicing is incomplete, and requires some additional hand-coding for fully executable programs. In particular, the top-level scheduler is not generated, and the results of rate analysis need to be operationalized manually; we plan the completion of code generation as a next step, and expect no major obstacles. By contrast, the runtime system and higher-order component model are functionally complete, such that current semi-manually coded multi-rate component implementations have full binary compatibility with core SIG components generated by the compiler. Together with hand-coded Java GUI controls, SIG programs can run interactively in (soft) real time.

As mentioned in section 1, we envision a hierarchy of layers with increasing expressivity and implementation requirements for SIG, such that an adequate problem-specific balance can be chosen for each application program. The core layer features first-order functions and finite types, and hence has a static memory footprint and can be translated effectively to hardware; in fact, we are planning an FPGA backend. The higher-order features demonstrated here belong to a higher layer that requires more implementation effort.

The current Java implementation has closure-based support [15]. As such, it makes use of dynamic memory management that could cause problems with real-time constraints. Note that in the Shepard example, the allocation rate is *one* closure at $\mathcal{R}_1$, on the order of seconds. We expect that even this naïve approach on a standard JVM platform will scale well enough for many applications. A more rigorous approach would be to eliminate higher-order functions by defunctionalization, as proposed in [16]. Because SIG is non-recursive, this would allow for static memory immediately, and the result could be executed on real-time JVMs or translated to more low-level target platforms.

5.2 More Related Work

In the design of SIG, we have deliberately started from scratch in order to avoid some of the complexities of existing synchronous data-flow approaches. As a result, its relationship to established programming languages and systems is somewhat oblique, and theoretical groundwork for comparison emerges only gradually. Nevertheless, we shall give a few further comments that might be helpful to the reader, even if they point in directions not directly relevant to the present discussion.

The multi-rate extension of SIG bears a superficial similarity to the *clock calculus* [2] of French synchronous languages such as LUSTRE or LUCID SYNCHRONE. However, there are crucial differences: "clocks" in their sense are (abstractions of) boolean streams that filter other streams at "rates" that are quantized over the base rate, but otherwise arbitrarily irregular and under internal program control.[9] This makes their clock analysis a very different business than our rate analysis: Clocks in SIG exist in the execution model only, but they are everyday clocks ticking at regular intervals.

Our approach to extensive support for multi-rate systems is particularly called for by the audio application domain, but also for simulations of dynamical systems [11]. By contrast, classical numerical tools such as SIMULINK and continuous-time variants of FRP have a notion of *adaptive* rate. This stems from a numerical account of calculus problems, where discretizations at different rates form a family of approximations to the "true" behavior. SIG allows patterns of discrete data-flow computation, in particular involving delay by a fixed number of steps as opposed to a fixed interval, that are generally incompatible with the assumption of a continuous limit. However, certain ubiquitous situations are exceptions to the rule; see the `dint` example. We point to the identification of the "continuizable" sub-calculus of data-flow computations as an interesting open problem.

[8] Should this be called *oligochronous*?

[9] We consider this a blatant misnomer: clocks that tick sometimes are broken.

Acknowledgments

Anonymous referees have made a substantial number of useful suggestions for the improvement of this paper, and the next few.

References

[1] T. Bartenstein and Y. D. Liu. Rate types for stream programs. In A. P. Black and T. D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 213–232. ACM, 2014.

[2] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[3] P. Caspi and M. Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000.

[4] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In B. C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 74–85. ACM, 2001.

[5] G. Giorgidze and H. Nilsson. Switched-on Yampa. In *Practical Aspects of Declarative Languages (PADL 2008)*, pages 282–298, 2008.

[6] K. Hammond and G. Michaelson. The design of Hume: A high-level language for the real-time embedded systems domain. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 127–142. Springer-Verlag, 2003.

[7] P. Hudak. Principles of functional reactive programming. *ACM SIGSOFT Software Engineering Notes*, 25(1):59, 2000.

[8] O. Kiselyov, C.-C. Shan, and Y. Kameyama. Bridging the theory of staged programming languages and the practice of high-performance computing. Technical Report 2012–4, National Institute of Informatics, Japan, 2012.

[9] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *J. Funct. Program.*, pages 467–496, 2011.

[10] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Haskell Workshop*, pages 51–64. ACM, 2002.

[11] J. Pearce, R. Crosbie, J. Zenor, R. Bednar, D. Word, and N. Hingorani. Developments and applications of multi-rate simulation. In *Computer Modelling and Simulation, 2009. UKSIM '09. 11th International Conference on*, pages 129–133, March 2009.

[12] J. Risset. Pitch and rhythm paradoxes: Comments on "auditory paradox based on fractal waveform" [j. acoust. soc. am. 79, 186–189 (1986)]. *The Journal of the Acoustical Society of America*, 80(3):961–962, 1986.

[13] R. N. Shepard. Circularity in judgements of relative pitch. *Journal of the Acoustical Society of America*, 36(12):2346–2353, 1964.

[14] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[15] B. Trancón y Widemann and M. Lepper. On-line synchronous total purely functional data-flow programming on the java virtual machine with sig. In *Proc. International Conference Principles and Practice of Programming in Java (PPPJ 2015)*. ACM, 2015. To appear.

[16] B. Trancón y Widemann and M. Lepper. Foundations of total functional data-flow programming. In *Mathematically Structured Functional Programming (MSFP 2014)*, volume 153 of *Electronic Proceedings in Theoretical Computer Science*, pages 143–167, 2014.

[17] B. Trancón y Widemann and M. Lepper. Sound and soundness – practical total functional data-flow programming. In *Functional Art, Music, Modeling and Design (FARM 2014)*, pages 35–36. ACM Digital Library, 2014. Demo abstract.

[18] B. Trancón y Widemann and M. Lepper. Laminar data flow: On the role of slicing in functional data-flow programming. In *Trends in Functional Programming (TFP 2015)*, 2015. Draft proceedings.

[19] T. Uustalu and V. Vene. The essence of dataflow programming. In K. Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.

A. Auxiliary Material: Audio File

The supplementary audio file[10] is a 60 s rendering of the Shepard Tone. It has been produced by the SIG implementation as depicted in Figure 4, and some auxiliary tools.

The program has been "compiled" to Java, the target language of the SIG compiler. Java coding currently requires some manual extrapolation from existing code generation schemes and novel language features, due to compiler incompleteness. But the partially hand-coded components instantiate the Java runtime environment for SIG, and are hence fully binary compatible to other, properly compiled SIG components.

The Java application has been run in offline mode, not driven by a real-time audio output buffer, but by a non-interactive main loop, writing raw audio data to a file, which has been converted via lossless WAV to MP3 format using free tools.[11]

The first stage of the `shepard` component has been configured as follows:

$$\mathit{ivl} = 1/2\,\mathrm{octave} \qquad \mathit{res} = 1200$$
$$\mathit{base\_freq} = 440\,\mathrm{Hz} \qquad r = 8$$

We have used a normalized sine for the `wave` function and a logarithmic fading of 10 dB per `ivl` for the `envelope` function. The resulting second stage has been driven with the following virtual real-time parameters:

$$\mathcal{R}_1 = 1/12\,\mathrm{Hz} \implies \mathcal{R}_2 = 100\,\mathrm{Hz}$$
$$\mathcal{R}_3 = 44.1\,\mathrm{kHz}$$

Audio signals have been represented as Java `double` internally, and quantized to 16 bit PCM during format conversion.

The Java application, including WAV file generation, has been executed on a stock Oracle JRE 1.8.0_45 on an Intel Core i5-3317U CPU at 1.70 GHz. The audio stream has been generated in measured effective 11% of real time, that is, at nine times the required audio rate.

[10] Also available from:

`http://bandm.eu/music/downloads/shepard.mp3`

[11] `javax.sound.sampled` to WAV, then `sox` to MP3.