A SONIFICATION INTERFACE UNIFYING REAL-TIME AND OFFLINE PROCESSING

Hanns Holger Rutz **Robert Höldrich**

University of Music and Performing Arts Graz
Institute of Electronic Music and Acoustics (IEM)
hanns.rutz | robert.hoeldrich @kug.ac.at

ABSTRACT

SysSon is a sonification platform originally developed in collaboration with climatologists. It contains a domain-specific language for the design of sonification templates, providing abstractions for matrix input data and accessing it in real-time sound synthesis. A shortcoming of the previous version had been the limited breadth of transformations applicable to this matrix data in real-time. We observed that the development of sonification objects often requires pre-processing stages outside the real-time domain, limiting the possibilities of fully integrating models directly into the platform. We designed a new layer for the sonification editor that provides another, semantically similar domain-specific language for offline rendering. Offline and real-time processing are unified through common interfaces and through a mechanism by which the latter can make use of the outputs of the former stage. Auxiliary data calculated in the offline stage is captured by a persisting caching mechanism, avoiding waiting time when running a sonification repeatedly.

1. INTRODUCTION

The *SysSon* sonification platform came out of an eponymous research project running 2012–2015 whose aim was the systematisation of sonification practices [1]. It was designed to specifically handle the type of data produced in climate research, namely multi-dimensional time series, and to be able to translate this data into real-time sound synthesis processes. Another aim was to enable scientists—climatologists in our case—to directly engage with the sonification by running *SysSon* as an application that allows them to import data files and to select, parametrise and listen to sonification models. Development was resumed in 2016 as a case study in knowledge transfer from the artistic into a scientific domain. Working with the same group of climatologists, but using an adapted methodology, we aim at making the platform more practical in typical usage scenarios.

As the design process of sonification requires quick iterations of programming, adjustment and listening, *SysSon* was conceived also as an integrated development environment (IDE) for the sonification design itself, and not just an interface for the deployment of finalised sonification models to scientists as "end users". Consequently, the platform integrates the two perspectives of programming and editing sonification models, on one hand, and the usage and adjustment of sonification models, on the other hand. The former is done through an integrated text editor and compiler—using embedded domain specific language (DSL) extensions for the Scala programming language—the latter through a graphical user interface (GUI).

Until recently, sound computations were restricted to mainly real-time synthesis, based on abstractions provided by a computer music framework [2] that utilises *SuperCollider* in the back-end. For example, there are objects that encapsulate functions that define a *SuperCollider* graph of signal processing blocks or unit generators (UGens). These objects can be copied and moved around in the application and posses a dictionary of parameters, such as scalar values, references to buses, sound files, etc. In the IDE, the UGen graph function is written inside a text editor, using Scala, while the associative dictionaries are generally editable through GUI elements and drag-and-drop.

We have found that the restriction to real-time synthesis can be limiting in many cases, and thus we set out to understand how dedicated pre-processing stages could be added to our system. What we are interested in is to learn how our specific embodied knowledge of the software platform can be analysed through its development process, and how it can be translated to yield a more practical application. The rest of the article is organised as follows: First we analyse how the advantages and limitations of the previous version of the platform led to particular ways of developing sound models. This is elaborated through the specific scenario of sonifying the quasi-biennial oscillation (QBO). We then show how we extended the system by a non-real-time component that in many ways mirrors the real-time API, making it easier to unify the two time domains.

2. DEVELOPMENT PROCESS

It is clear that any decision on design and representations enables particular ways of working with and thinking about the subject matter, as much as they inhibit others. For example, being able to create and program the sound objects from within the application allows them to be conceived as things that can be evolved, duplicated, moved and copied

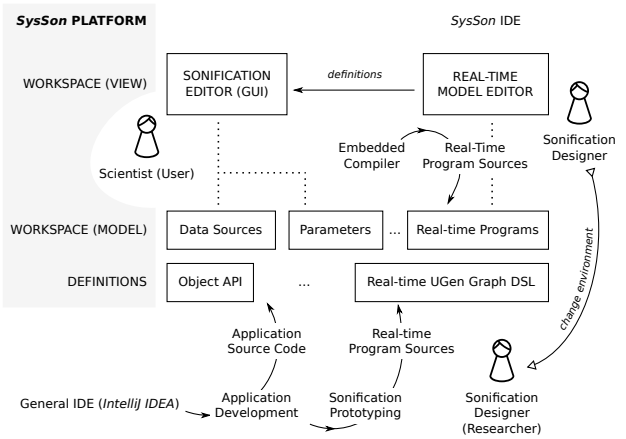


Figure 1. User and developer perspectives and dynamics. This is before the introduction of offline pre-processing.

around between workspaces. The sound objects *live* in workspaces, and in fact the platform can be used to program entire music compositions or sound installations.

On the other hand, our IDE is very rudimentary compared to a professional IDE, when it comes to code navigation, code completion, error checking, break-point debugging and so forth. Also, a monolithic real-time UGen graph function as the embodiment of a sonification process is a strong constraint. Both the delegation of tasks to the graphical interface and the restriction to a single graph function had been perceived as restrictions by some participants of a workshop on *SysSon* we held in 2014. In our own practice of developing sonification models, we found that we could experiment better with some ideas by falling back to a professional IDE (*IntelliJ IDEA*) that was used to develop the application itself, temporarily removing some constraints. This was possible due to the existence of an overall application programming interface (API) extending beyond what is exposed in the GUI. This type of development process, where the sonification designer moves back from the in-application coding to an outside IDE, the same IDE that is used to write the entire application, is illustrated in Figure 1.

2.1 Case Study: Quasi-Biennial Oscillation

To understand this process better, and how it can connect with the in-application work, we present as a case study the creation of a particular sonification model. The scenario in climatology that we were targeting was the quasi-biennial oscillation (QBO) [3]. This phenomenon occurs in the lower stratosphere and equatorial latitudes, and includes temperature anomalies that rhythmically alternate approximately every two years. A temperature anomaly is a deviation of the temperature from its average value at a given time and location. In the QBO, these anomalies oscillate with the temporal phases shifting from higher to lower altitudes, something that can be clearly seen in a plot (Figure 2).

Because of its technical simplicity, many sonification models are built on what has been called *parameter mapping* [4], techniques of more or less directly translating

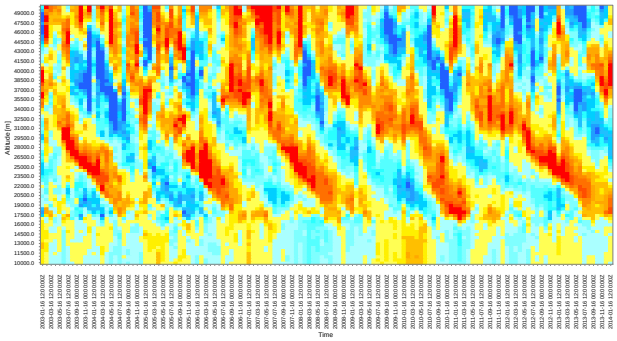


Figure 2. Temperature anomalies ($-5°C$ blue, $+5°C$ red), equatorial in $10km$ to $50km$ altitude, Jan 2003 to Jan 2014.

(mapping) a stream of input data to a stream of data that controls one or several sound parameters. Parameter mapping is also suggested in the design of *SysSon*, as the prominent abstraction added to the set of UGens is `Var`, a reference to matrices and sub-matrices that may then be unrolled as a temporal and/or multi-channel real-time signal that can be treated like any other UGen. While we can create hundreds or even thousands of oscillators or modulators this way without technical problems, obtaining a suitable sonic representation of a multi-dimensional matrix is all but easy. We have tried to address this in the GUI by allowing the user to quickly create sub-selections from matrices and to assign different dimensions to different logical inputs of the sound model.

Nevertheless, it became more and more clear that what was missing from the framework was some general notion of what Grond and Berger call the *data preparation* stage. The data sets in our scenario are stored in *NetCDF* format, and of course it is possible to prepare these sets with various other tools, with numerical computing software, and so forth. The climatologists are used to these tools and perform typical operations such as reducing the resolution of a data set, averaging over a dimension, or calculating the anomalies by subtracting out the monthly averages of a variable. However, we would like to build sonification templates that imply a certain pre-processing without requiring the user to carry it out by hand. This is especially true if we want to explore more unconventional data preparation.

One of the ideas we developed for creating a sonification of the QBO phenomenon is to use an algorithm from the visual domain, a blob detection. Blob detection is an approach from computer vision to determine coherent shapes in a static or moving digital image. The motivation came from the clarity with which we identify the regular diagonal shapes in Figure 2. If we transformed the time-altitude grid of temperature data into a low number of expressive features, we could potentially build a sonification that paralleled the visual clarity in the aural domain.

2.2 Obtaining and Representing Blob Data

To prototype this idea, we looked for a simple open source library that could be used from within the Java Virtual Machine (JVM), since our application runs on the JVM. We

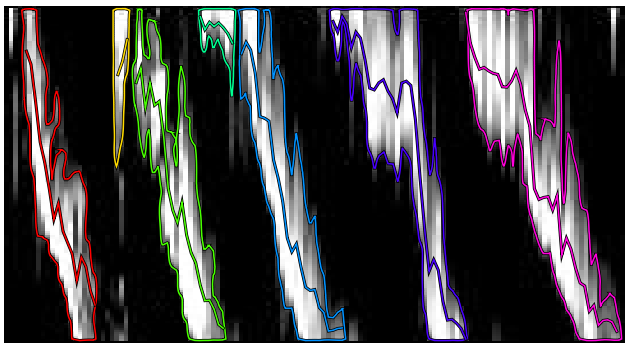


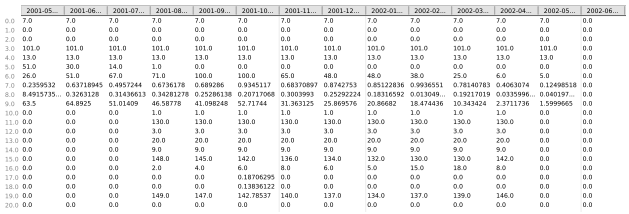
Figure 3. Output of blob detection for positive temperature anomalies (detection threshold $+0.26°C$), equatorial in $21km$ to $36km$ altitude, Jan 2003 to Jan 2014.

found a library written by Julien Gachadoat [1] that produced reasonable results while being implemented in only a few hundred lines of Java code. Within two days, we were able to take a data set of temperature anomalies, convert each latitude/longitude slice into a greyscale image with the time series on the horizontal axis (180 months) and altitudes on the vertical axis (600 levels), run the blob detection, and output a new matrix file containing the evolution of the blob shapes for a given maximum number of concurrent voices. Figure 3 shows the input matrix with blob contours. The polyline inside the blobs is one of the features extracted, the centroid or weighted arithmetic mean, which gives a good representation of the "glissandi" of the anomalies.

How can we make the results of the blob analysis available to the sonification? The solution we picked is to render each blob with one sound generator within a polyphonic UGen graph. The blob data is again rasterised along the time axis, and blobs are pre-sorted into voices. Since there is no support for dynamic growth of voices within a UGen graph, we define a maximum number of concurrent voices at the analysis stage. Theoretically, two voices would suffice to capture the overlapping diagonal shapes, but since the shapes are not fully regular and there occur disturbances to the pattern—visible in the figure by the very short yellow blob, for example—we use four voices *in the prepared data*, and twice as many voices *in the UGen graph*, so that we may employ envelope release phases that extend beyond the strict boundary of one time slice.

If the rasterised and organised blob-data is written out as another matrix file, we thus have a preparation stage that consists of a transformation (input matrices → structured blob-set → equidistant-sampled blob-data → voice distribution → output matrices). The written *NetCDF* matrix file can then be used inside the *SysSon* application like any other sonification data source. A sample section from such a file is shown in Figure 4. The time dimension is preserved and the altitude dimension is replaced by a new dimension that contains the blob "records", each of which occupies ten rows. Without going into the details, each blob record starts with a unique identifier (rows 0, 10, and 20 in the figure) that the sound synthesis will use to trace

[1] `http://www.v3ga.net/processing/BlobDetection/index-page-home.html` (accessed 18-May-2017)



	2001-05...	2001-06...	2001-07...	2001-08...	2001-09...	2001-10...	2001-11...	2001-12...	2002-01...	2002-02...	2002-03...	2002-04...	2002-05...	2002-06...
0.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3.0	101.0	101.0	101.0	101.0	101.0	101.0	101.0	101.0	101.0	101.0	101.0	101.0	101.0	0.0
4.0	13.0	13.0	13.0	13.0	13.0	13.0	13.0	13.0	13.0	13.0	13.0	13.0	13.0	0.0
5.0	51.0	30.0	14.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
6.0	26.0	51.0	67.0	71.0	100.0	100.0	65.0	48.0	48.0	38.0	25.0	6.0	5.0	0.0
7.0	0.2359932	0.63718945	0.4957244	0.6736178	0.689286	0.9345117	0.68370897	0.8742753	0.85122836	0.9936551	0.78140783	0.4063074	0.12498518	0.0
8.0	8.4915735...	0.3263128	0.31436613	0.34281278	0.25286138	0.20717068	0.3003993	0.25292224	0.18316592	0.013049...	0.19217019	0.0335996...	0.040197...	0.0
9.0	63.5	64.8925	51.01409	46.58778	41.098248	52.71744	31.363125	25.869576	20.86682	18.474436	10.343424	2.3711736	1.5999665	0.0
10.0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0
11.0	0.0	0.0	0.0	130.0	130.0	130.0	130.0	130.0	130.0	130.0	130.0	130.0	0.0	0.0
12.0	0.0	0.0	0.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	0.0	0.0
13.0	0.0	0.0	0.0	20.0	20.0	20.0	20.0	20.0	20.0	20.0	20.0	20.0	0.0	0.0
14.0	0.0	0.0	0.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	0.0	0.0
15.0	0.0	0.0	0.0	148.0	145.0	142.0	136.0	134.0	132.0	130.0	130.0	142.0	0.0	0.0
16.0	0.0	0.0	0.0	2.0	4.0	6.0	8.0	6.0	5.0	15.0	18.0	8.0	0.0	0.0
17.0	0.0	0.0	0.0	0.0	0.0	0.18706295	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
18.0	0.0	0.0	0.0	0.0	0.0	0.13836122	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
19.0	0.0	0.0	0.0	149.0	147.0	142.78537	140.0	137.0	134.0	137.0	139.0	146.0	0.0	0.0
20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 4. Section from a rasterised blob matrix

a coherent object. The identifier is zero whenever the particular voice is not allocated. In the shown table, the top most voice begins at the first time index with blob id 7, a second voice with blob id 1 is allocated in the forth column. Next in the record are the blob dimensions (first altitude level, start time index, number of altitude levels, number of time slices), followed by the features of each time slice, including mean, standard deviation and centroid of the temperature anomalies included (visible in the table by their floating point nature).

3. INTEGRATED OFFLINE UGEN GRAPHS

The problem with the creation of the blob matrix is, of course, that it is not yet in any way integrated into the application. While this is acceptable for research, we do not want the climatologists to send us their data sets in order to run them through the transformation and send them back to be used in the sonification. A simple solution would be to add an action to the workspace GUI to allow the users to pre-process their matrices and then re-import the resulting blob matrix into the workspace. In fact, we have taken this route before with an action to pre-calculate an anomalies matrix from a regular (e.g. temperature) matrix. But this way, the underlying problem is not addressed: If one can think up and implement a real-time sonification model from scratch with the built-in code editor, why is there no counterpart for the pre-processing stage?

The idea that we have now implemented and that we are now testing is this: **Can we conceive a domain-specific language for offline processing that is relatively easy to understand if one has already learnt how to write the real-time code?** Is the UGen graph model applicable for the pre-processing stage? What are its advantages, disadvantages, impedance matches and mismatches?

The UGen graph concept is well established in sound synthesis and could be seen as a form of dataflow or stream processing model. A network of unit processors is run on a scheduler, each consuming and producing a stream of values. For example, in *SuperCollider*, the UGens form a directed acyclic graph and process floating point values either as chunks at audio sampling-rate, or as individual values at control rate, a fraction of the audio sampling-rate that makes computations less expensive. In the beginning of 2016, we were working precisely on the translation of the UGen graph model to offline processing in the reformulation of the *FScape* audio signal processing software. A few months later, the implementation was stable enough to use it to render not just sounds, but also bitmap images

```
// real-time
val gen = LFSaw.ar(SampleRate.ir/800, 0.5)
val up  = (gen + 1) * 2
val a   = up.min(1)
val b   = (up - 3).max(0)
a - b

// offline
val gen = LFSaw(1.0/800, 0.75)
val up  = (gen + 1) * 2
val a   = up.min(1)
val b   = (up - 3).max(0)
a - b
```

Figure 5. Comparison of real-time and offline graphs, using the Scala DSLs in *SysSon*.

and video sequences, suggesting that this framework could be also used as the basis for offline processing of matrices in *SysSon*. It is thus useful to summarise the experience with this framework so far:

3.1 Applying the UGen Model to Offline Processing

At first glance, offline and real-time processing should be quite similar, only distinguished by the kind of scheduler that runs the nodes of the graph, in the former case being busy incessantly, and in the latter case paying attention to deadlines. Figure 5 shows the example of an envelope generator in real-time (*SuperCollider*) and in offline (*FScape*) code. The envelope goes from one to zero in 200 frames, then back to one in 200 frames, and remains there for another 400 frames. Three subtle difference can be seen already:

- The phase argument to the sawtooth generator differs. While in *SuperCollider* the ranges of some parameters have historically been chosen to minimise mathematical operations on the server side, in *FScape* we favour consistency across various UGens over additional arithmetical operations required.

- In *SuperCollider* we must specify whether the UGen uses audio or control rate, while in *FScape* these categories do not exist. Instead, the block sizes of signals depend on the particular UGen and may even vary across the inputs and outputs of a UGen, although a nominal block size can be specified that determines the default buffer size for dynamic signals.

- The frequency is given in Hertz in *SuperCollider*. In *FScape* it is given as a normalised value, as there is no notion of a global sampling rate. We were contemplating the introduction of a `SampleRate` UGen for convenience, but since the system was primarily designed to transform audio files, one would mostly want to work directly with the respective sampling rates of the input files. In the cases where images or videos are processed, the notion of audio sampling rate would also be meaningless. In fact, many UGens are now specified to take period parameters in number-of-frames rather than their reciprocal, a normalised frequency, as we also support integer numbers next to floating point numbers, avoiding time-base drift over long runs, something that is known to cause issues in *SuperCollider*.

Property	***SuperCollider***	***FScape***
Language	C, C++	Scala
Scheduler	custom written	*Akka Streams*
Threading	single threaded	optional asynchronous, thread pool
Data rates	dual (audio and control)	multi-rate
Buffering	assigned fixed size “wire-buffers”	variable size, reference passing
Data type	float	unrestricted
Life cycle	all UGens start and stop together	UGens can shut down at different times

Table 1. Comparison of back-end architecture

Both systems use a weak type `GE` (graph element) for the values that can be used as parameters to UGens (constants are graph elements, as are UGens themselves), but while on the *SuperCollider* server all streams are restricted to 32-bit floating point resolution, in *FScape* we adopted multiple numeric types, consisting currently of 32-bit and 64-bit integer numbers and 64-bit floating point numbers. Some UGens are generic in that they can natively work with different numeric types, preserving particular precision and semantics, while others coerce a `GE` into a particular type. For example, `ArithmSeq(0.0, 4)`—similar to `Dseries` in *SuperCollider*—produces the arithmetic series $0.0, 4.0, 8.0, \ldots$ with floating point numbers, while `ArithmSeq(0L, 4)` produces the series $0L, 4L, 8L, \ldots$ with 64-bit integer numbers. This approach is still under evaluation.

The way UGens are constructed from the user-facing side is very similar, including lazy expansion of elements that allows the preservation of pseudo-UGens (e.g. in serialisation) and the late binding of resources needed from the environment (e.g. automatic adaptation of the expanded UGen graph to variable types and dimensions of control inputs), multi-channel expansion, and the composition with unary and binary operators. But when it comes to the implementation of the UGens, the systems use very different architecture, which is summarised in Table 1.

3.2 Interfacing with an Offline Graph

It remains to define an input/output interface for integrating offline processing graphs with other objects inside *SysSon*. All objects share as a common interface a dictionary—called attribute map—that can be used to freely associate attributes with it, using conventional string keys. In the real-time case, for example, when one creates a named control `"freq".kr`, the control input is looked up in the attribute map of a containing `Proc` object (the object wrapping the UGen graph) at key `"freq"`. This could be a scalar value, a break-point function, or any other object that could be

```
val m  = Matrix("var")
val v0 = m.valueSeq
val v  = Gate(v0, !v0.isNaN)
val mn = RunningMin(v).last
val mx = RunningMax(v).last
MkDouble("min", mn)
MkDouble("max", mx)
```

Figure 6. An offline program that determines the range of values of a matrix.

translated into a control signal. The real-time signal output of a `Proc` is represented by a `ScanOut` UGen, for which a dedicated output object must be created inside a special `outputs` dictionary of the `Proc`. This object can then be placed in the attribute map of another object that wants to use the corresponding signal as input. These links thus function as "patch cords" between objects.

In the offline case, we use a similar approach. Here `"freq".attr` would again inject a value or value stream from the containing object's attribute map. Other UGens exists for other types of input: To get access to a matrix, one uses `Matrix("key")` where the matrix is then sought in the attribute map or in the list of variables of the containing `Sonification` object using `"key"`. Likewise, different UGens exists for the different types of outputs produced. Figure 6 gives the example of calculating the minimum and maximum elements of an input matrix. The `valueSeq` method on the matrix produces a one-dimensional "flat" (interleaved) stream of all the elements in the matrix. The `Gate` is used to remove missing values (NaN is for not-a-number), which commonly occur in the climatology data files. The `RunningMin` and `RunningMax` UGens are similar to their *SuperCollider* equivalents, but the `last` operation is unique: In the offline processes, UGens may terminate at different times, once their inputs or outputs are exhausted. The `valueSeq` stream terminates when all elements have been transported, and subsequently `Gate` and `RunningMin/Max` will terminate. The `last` operation produces a UGen that outputs only the last element from its input stream. There are also operations such as `drop` and `take` that act like their counterparts in standard Scala collections.

The last two lines define the outputs of the program. The string parameter defines the key in the output dictionary of the object containing the offline program. Corresponding output objects must be created in that dictionary and may then be linked to the attribute map of a `Proc`, where they are then used in the definition of the real-time process. Figure 7 shows a screenshot to clarify how offline and real-time programs are wired together and integrated within a sonification object. The editor of the offline program is in the top-left, containing the code of Figure 6, along with the two output objects named *min* and *max*. These two objects have been linked through drag-and-drop with the attribute map, shown in the bottom-right, of the real-time object, whose code in turn is shown in the editor window in the top-right.

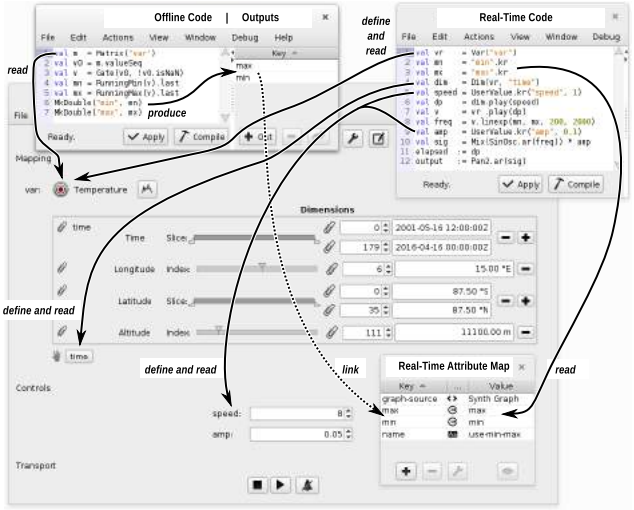


Figure 7. Interlinkage of offline and real-time programs within the application's user interface

The large window in the background shows the sonification editor GUI as it is presented to the user, e.g. the climatologist. The interface elements have been programmatically generated by the real-time program (this is still considered the main entry to the sonification model, while the offline programs function as auxiliary components): `Var` refers to an input variable like `Matrix`, but additionally defines a corresponding user interface element in the sonification editor. `Dim` UGens allow the user to associate dimensions of the sonification model with dimensions of the matrices sonified. The `UserValue` UGen, used for speed and amplitude here, is a control like `"key".kr`, but also requests the presentation of a user interface element. In a future version, we envision a set of dedicated user interface objects that can be linked to standard controls, disentangling the real-time program from decisions on the user interface.

3.3 Data Preparation Stage

We now return to the blob example. In order to reformulate the data preparation, we need to create new UGens that encapsulate the blob detection and analysis. Generally, the best approach is to minimise the functionality of a UGen, perhaps splitting an algorithm across a number of UGens, so that partial functions can be reused in other contexts. An obvious choice would be one UGen `Blobs2D` for the blob detection stage, yielding the contours of the blobs within each matrix slice, and a second UGen `BlobVoices` that transforms this set of geometric objects into the rasterised matrix of voices containing the required analysis of the blob contents, such as the sliding extent and centroid. The program complete with input UGen `Matrix` and output UGen `MkMatrix` is shown in Figure 8.

Without going into much detail, we want to highlight three aspects of this reformulated algorithm:

Program storage: We store a program as a tree of (unexpanded) UGens and not the source code or the compiled

```
val voices  = 4    // maximum polyphony
val taLo    = 0.0  // lowest  temp anomaly considered
val taHi    = 3.5  // highest temp anomaly considered
val thresh  = 0.26 // blob detection threshold

val mIn     = Matrix("anom")
val d1      = Dim(mIn, "time")
val d2      = Dim(mIn, "altitude")
val width   = d2.size
val height  = d1.size
val blobSz  = voices * 10 // blob record has 10 elems

// transform the matrix spec (shape)
val specIn  = mIn.spec
val s1      = specIn.moveLast(d1)
val s2      = s1    .drop    (d2)
val d3      = Dim.Def("blobs", values =
     ArithmSeq(length = blobSz))
val specOut = s2    .append  (d3)

val win0    = mIn.valueWindow(d1, d2)
val win1    = Gate(win0, !win0.isNaN)
val win     = win1.max(taLo).min(taHi) / taHi

val blobs   = Blobs2D(in = win, width = width,
     height = height, thresh = thresh, pad = 1)

val winB    = BufferDisk(win)
val mOut    = BlobVoices(in = winB, width = width,
     height = height, numBlobs = blobs.numBlobs,
     bounds = blobs.bounds, numVertices =
     blobs.numVertices, vertices = blobs.vertices,
     minWidth = 10, minHeight = 4, voices = voices)

MkMatrix("out", specOut, mOut)
```

Figure 8. UGen-based formulation of the blob matrix preparation

bytecode. This has two reasons: First, Scala is a statically typed and compiled language, and while we embed an on-the-fly compiler for read-eval-print-loop (REPL) functionality, the compilation is slow and resource hungry. In the real-time case, compiling on-the-fly is prohibitive, and regular language control structures such as `if-then-else` blocks cannot be represented on the sound synthesis server, so there is little benefit to store Scala programs. For the offline case, storing regular (compiled) Scala programs is still an interesting option, and something that is done already elsewhere in the application for `Action` objects. The problem here is compatibility—on the one hand, Scala has adopted a policy of allowing binary incompatibility between major release versions in order to facilitate language evolution. *SysSon* has recently been migrated from Scala 2.11 to 2.12, so if we had stored binary code programs, we would face the problem of automatically detecting incompatible versions when opening an old workspace and recompiling them. An interesting new possibility will arrive in the future when Scala has migrated to a new serialisation format called *TASTY* [5] that promises to address this. But even then, a danger remains that closures may unintentionally capture variables in the system (cf. [6]). The second reason is that we would have to give much stronger guarantees about the stability of the entire API that is exposed if we stored the binary program. Storing the UGen graph has been proven a good solution, because it still very expressive as it basically corresponds to the abstract syntax tree (AST) of the DSL, and we can change implementation details without rendering the programs incompatible.

The isolation of the DSL has brought another tremendous advantage, as we can compare two programs for structural equality. This is crucial to correctly **cache the results** of the offline program. Because analyses like the blob detection can take some time, being able to cache the results allows the user to quickly adjust sonification parameters without being penalised by long waiting times, as long as the parameters do not affect the structural equality of the program (e.g. a different part of the input matrix must be analysed).

Manipulating structural data: Partly related, a recurring problem or question is the transport of structural data that is not well captured by the streaming of plain numeric values. This is clearly showing here in the adaptation of the matrix specification from input to output: We need to be able to remove the altitude dimension and replace it by a new "record" type of dimension for the blobs here, the transformation from `specIn` to `specOut`. The DSL must foresee all possible scenarios since, because of the program storage constraints, it is not possible to use the expressive Scala language feature of applying a higher-order function, e.g. writing `specIn.filterNot(_.name == d2.name)`. But even if this was possible, we have a mismatch with the streaming protocol. In `Blobs2D`, we have regular vectorial data of fixed window sizes flowing into its input, but for each window we have a varying (much smaller) number of hierarchical output records that would ideally be objects in the form of

```
class Blob(bounds: Rectangle, slices: List[Slice])
class Slice(extent: Int, mean: Double, std: Double)
```

Remarkably, there are only few discussions on representing record-type data in streaming systems. There was work on this for *FAUST* [7], but even there records are supposed to bundle together flat objects of equal rate components, and nested structures are not addressed. Systems either force all data into vectorial representations—for instance, *OpenCV* puts polygon output from analysis into a standard matrix, just as we output the blob contours as a stream of pairs of x/y coordinates demarcated by an independent outlet for `numVertices`—or they divide the world into vector data and message or event data, as in *Pure Data* and *Max/MSP*, where records can be represented as message lists, although the problem of nested data structures is addressed just as little.

Multi-rate alignment: The line `val winB = BufferDisk(win)` indicates a problem. The streaming infrastructure we have chosen, *Akka Streams*, uses back-pressure, a combination of dynamic push (data-driven) and pull (demand-driven) approaches with bounded queues between

nodes. This is generally helpful when real-time properties are not required, because the system tends to adjust itself to the particular data-rates produced in the nodes, while automatically blocking unlimited growth of unprocessed messages (resulting in running out of memory), where a consumer node cannot catch up with the pace of a producer node. Nonetheless, whenever the program contains a diamond topology—a sink and source are connected by more than one path—this can result in a stalling process, and appropriate intermediate buffers must be inserted. *Akka* currently has no means to automatically detect these situations, and so it is the programmer's responsibility to manually insert buffers. `BufferDisk` is the most simple form of such a buffer, as it is unbounded and written to disk if it becomes too large. Alternatively, we could have just duplicated the `valueWindow` call, although we would then also duplicate a few more nodes and thus require more computation.

In a traditional signal processing scenario, it is easy enough to calculate the necessary buffers, and a future version of *FScape* can probably accomplish this automatically, but with non-vectorial structural data such as the output of the blob detection, we fear that manual control will still be required. Again, some research on multi-rate management exists, e.g. [8], but this is focused on real-time scenarios (with properties such as de-coupling that do not apply here) and vectorial data. Another related aspect is the complexity of implementing multi-rate UGens with many inputs and outputs. As Arumí Albó has noted, handling multi-rate signals inside modules "yields complex code in every concrete module" [9, p. 145].

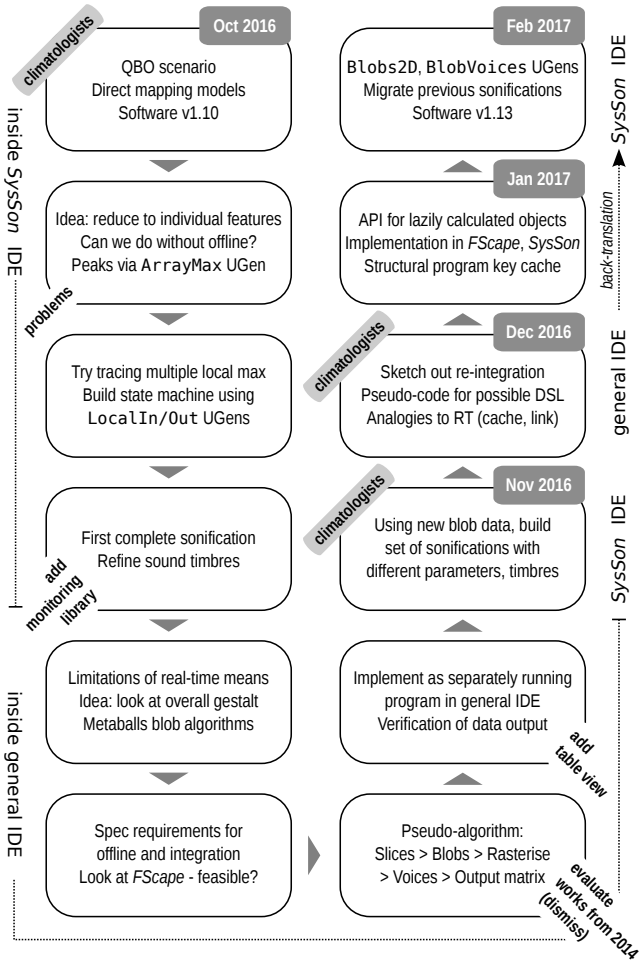


Figure 9. Stages in the development of the blob sonification

4. DISCUSSION AND CONCLUSION

We have presented a new architecture within the *SysSon* sonification platform to integrate real-time sound synthesis with offline pre-processing stages, using a similarly constructed DSL for writing UGen graphs. The work is the result of quick iterations that responded to feedback from climatologists who we are trying to give a tool at hand that they can use all the way from data import to model selection and parametrisation. The architecture was elaborated through the case study of using a blob detection algorithm to sonify the QBO phenomenon. After presenting a set of rendered sounds to the climatologists, [2] it became quite clear that allowing them to navigate interactively through the data and adjust different sound parameters would be a requisite for advancing the sound models, so having an integrated data pre-processing stage was an important step towards this goal.

Another important result of this research is a better understanding of the development and transfer processes that take place at the various stages of experimenting with new sonification approaches and feeding them back into a stable platform. It emerges a pattern of oscillation in this process between the greenfield experimentation in the external IDE, and reformulation within the inner IDE for deployment. This movement is illustrated in Figure 9.

A number of concrete suggestions can be made with respect to the future development of this architecture:

- It would be extremely interesting to have an interactive worksheet or REPL mode for offline programs. That is to say, the ability to execute these programs step by step, validate the intermediate outputs, readjust, etc., akin to the "notebook" mode of operation that has gained much traction with *iPython* and *Jupyter* [10].
- Offline programs could become a formalisation within the application code base itself for various situations:
  - Real-time UGens could be added that implicitly spawn pre-processing stages, freeing the authors of sound models to create separate offline objects for trivial tasks. For example, querying the minimum and maximum value of matrix would be better suited as operators directly available within the real-time program.
  - The user (scientist) facing side of the editor could have a richer palette of matrix transformation operators. For example, ad-hoc averaging across a dimension or down-sampling would be a very useful operation offered to the user, and of which a particular sonification model

[2] See the revised sets labelled `WegC_170508-...` at `https://soundcloud.com/syssonproject/sets/` (accessed 18-May-2017)

would not need to have any knowledge or presumption. Such additional matrix reduction or transformation operations could transparently spawn corresponding offline programs before feeding the sonification.

  - The program already provides a set of dedicated transformation utilities, e.g. for the calculation of anomalies. These could now be implemented coherently through the offline DSL.

- The UGen implementation is quite involved, especially when structural and multi-rate data is needed. We suggest to define another intermediate layer on top of *Akka Streams* that provides more formal definitions for finite state machines. The ad-hoc state machine for `BlobVoices` is several hundred lines of code, and thus way too complex. We need better abstractions or macros for the implementing classes. Had we fused `Blobs2D` and `BlobVoices` into the same UGen, the process would have been obviously much simpler—as we would have skipped the stream representation of nested data structures—but this experiment was exactly aimed at seeing if passing around structural data was feasible within the existing streaming framework. We can conclude that it is feasible but demands better library support to make writing these types of UGens less work and error-prone.

An overall picture emerges, where we need to reconsider the perspective on sonification. The existing system makes strong presumptions about the "object" nature of sonification—we have *a sonification*, we apply *a sonification*, a sonification *is a* sound process coupled with an associative dictionary of data sources, etc. Paying more attention to the details of the development process can help to understand sonification rather as a process itself, which brings into focus the translational competency of artistic researchers working in the field. Consequently, the challenge will be how this process view can be married to the (rightful) expectation of scientists to adopt these systems as stable and clearly delineated tools.

Acknowledgments

The *SysSon* platform was originally developed within project P 24159, funded by the Austrian Science Fund (FWF). Since 2016, the continued development at the Institute of Electronic Music and Acoustics (IEM) of the University of Music and Performing Arts Graz is funded by the Knowledge Transfer Centre South (WTZ Süd) Austria. Climate data sets and evaluation feedback is kindly provided by the Wegener Center for Climate and Global Change (WegC) at the University of Graz.

References

[1] K. Vogt, V. Goudarzi and R. Höldrich, 'SysSon – a systematic procedure to develop sonifications', in *Proceedings of the 18th International Conference on Auditory Display (ICAD)*, Atlanta, GA, 2012, pp. 229–230.

[2] H. H. Rutz, K. Vogt and R. Höldrich, 'The SysSon platform: A computer music perspective of sonification', in *Proceedings of the 21st International Conference on Auditory Display (ICAD)*, Graz, 2015, pp. 188–192. [Online]. Available: `http://hdl.handle.net/1853/54126`.

[3] M. P. Baldwin, L. J. Gray, T. J. Dunkerton, K. Hamilton, P. H. Haynes, W. J. Randel, J. R. Holton, M. J. Alexander, I. Hirota, T. Horinouchi, D. B. A. Jones, J. S. Kinnersley, C. Marquardt, K. Sato and M. Takahashi, 'The quasi-biennial oscillation', *Reviews of Geophysics*, vol. 39, no. 2, pp. 179–229, 2001. DOI: `10.1029/1999RG000073`.

[4] F. Grond and J. Berger, 'Parameter mapping sonification', in *The Sonification Handbook*, T. Hermann, A. Hunt and J. G. Neuhoff, Eds., Berlin: Logos Verlag, 2011, pp. 363–397.

[5] M. Odersky, E. Burmako and D. Petrashko, 'A TASTY Alternative', EPFL, Lausanne, Tech. Rep. 226194, 2016. [Online]. Available: `https://infoscience.epfl.ch/record/226194`.

[6] H. Miller, P. Haller and M. Odersky, 'Spores: A type-based foundation for closures in the age of concurrency and distribution', in *Proceedings of the 28th European Conference on Object-Oriented Programming (ECOOP)*, Uppsala, 2014, pp. 308–333. DOI: `10.1007/978-3-662-44202-9_13`.

[7] P. Jouvelot and Y. Orlarey, 'Dependent vector types for data structuring in multirate faust', *Computer Languages, Systems & Structures*, vol. 37, no. 3, pp. 113–131, 2011. DOI: `10.1016/j.cl.2011.03.001`.

[8] G. Essl, 'Playing with time: Manipulation of time and rate in a multi-rate signal processing pipeline', in *Proceedings of the 38th International Computer Music Conference*, Ljubljana, 2012, pp. 76–83. [Online]. Available: `http://hdl.handle.net/2027/spo.bbp2372.2012.013`.

[9] P. Arumí Albó, 'Real-time multimedia computing on off-the-shelf operating systems: From timeliness dataflow models to pattern languages', PhD thesis, Universitat Pompeu Fabra, Barcelona, 2009.

[10] H. Shen, 'Interactive notebooks: Sharing the code', *Nature*, vol. 515, no. 7525, p. 151, 2014. DOI: `10.1038/515151a`.