The SysSon Platform

Technical Report TR-2016-08-1
Institute of Electronic Music and Acoustics, Graz
(Status: completed)

Hanns Holger Rutz

August 2016

1 Statement

The next sub-sections lists the main points of the WTZ project, followed by an outline for the preparation of the first milestone.

1.1 General Mission Statement

- to understand the specific expertise of artists and the potentials of transfer of knowledge from research in the arts into applied contexts
- to define the interfaces necessary to warrant a successful application of approaches and prototypes that have been developed in long-term practice
- conduct a series of self-standing case studies, generate best practices examples
- interdisciplinary process of artistic research and natural sciences and engineering: assess and document
- production of a catalogue of criteria for future reference translation of artistic research "results" from proof-of-concept status to a status with greater number of recipients and users.

1.2 Project Year 1 – SysSon

- a case study is developed based on the prototype of a sonification software
- collaboration between IEM and WegC
- WegC staff is user group along with which prototype is elaborated, creating a product usable in their work
- development of sonification templates; creation of a suitable platform, distribution and integration
- systematic documentation; basis for successive project year

2 Areas for Development

2.1 Sonification Design

- Seamless transitions between server and client side processing
- Modularity of sounding components
- Better programmatic access to matrices
- Pre-processing stages for matrices
- UI: programmatic building blocks for control elements

2.2 Sonification Usage

- Integration: IPython / Jupyter, etc.; Scala in “Data Science”
- Stronger linkage between sounding and visual components
- Wizards, guided building blocks
- Import/export interfacing
- UI: docking framework, usability
- Specific scenarios, e.g. presentation tool for a conference, export for a web-site

2.3 Overall Computer Music Perspective

- UI: Linkage between text editing and visual components
- “Thinking tool”; note-pad, mark-down, hyper-text, mind-maps, small-talk

3 Next Planning Stage

In the meeting of 12-Aug-2016, it was decided that the strategy will focus on the development of the sonification templates, in iteration with select individuals from WegC over a longer period. Development of the software must be carefully planned in order to produce achievable milestones with the available resources. As a first step, the work effort for implementing the following changes should be assessed:

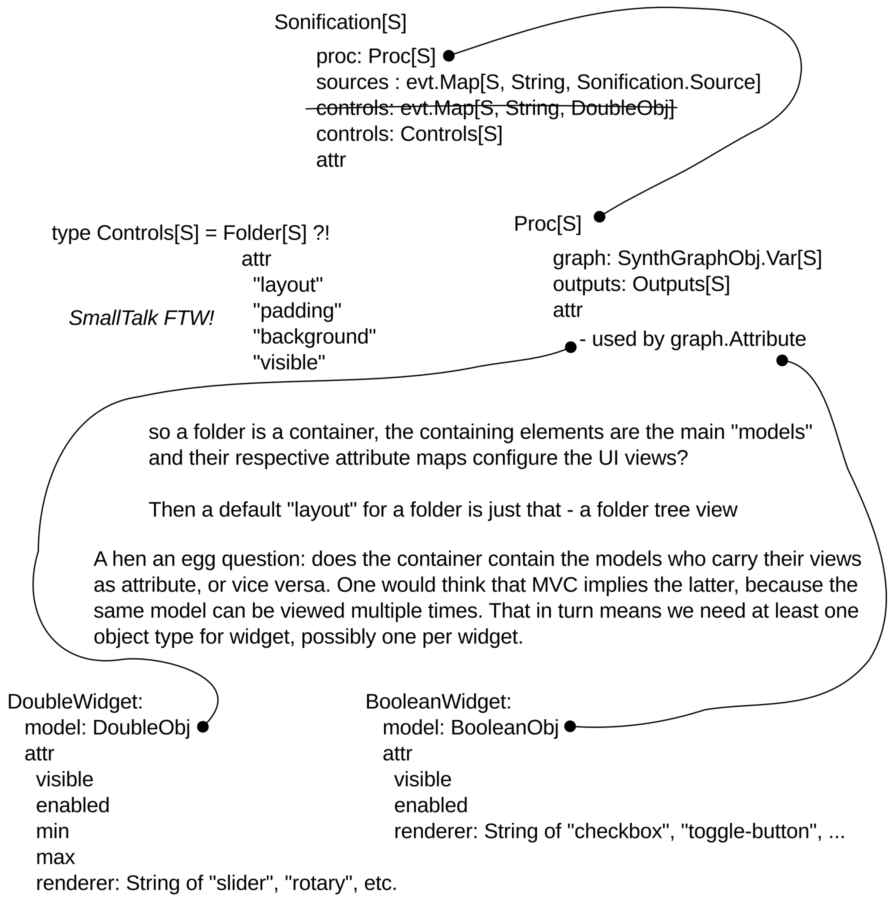
- Improving the sonification editor user interface by adding a few widgets such as buttons, toggle switches etc., being able to control their layout broadly.
- In conjunction, being able to branch in the synth graph, i.e. being able to select different sub-programs based for example on the user interface elements.
- Possibility of subjecting these branches to real-time interactive control (vs. simply evaluating at the beginning of the play action)

3.1 Sonification UI Elements

The current code for determining control elements is in `SonificationViewImpl` method `updateGraph`, with the following scanning process:

```
val userValues  = g.sources.collect { case graph.UserValue(key, default) =>
  val cell = CellView.exprMap[S, String, Double, DoubleObj](controls, key)
  val view = DoubleSpinnerView.optional[S](cell, name = key.capitalize,
    default = Some(default))
  (key, view)
}
```

In other words, we require that `UserValue` elements are found on the top level, and in the order of their appearance we build double-spinner-views. The idea is shown in Fig. 1



```
Sonification[S]
    proc: Proc[S]
    sources : evt.Map[S, String, Sonification.Source]
    ~~controls: evt.Map[S, String, DoubleObj]~~
    controls: Controls[S]
    attr

type Controls[S] = Folder[S] ?!
                   attr
                     "layout"
                     "padding"
                     "background"
                     "visible"

Proc[S]
    graph: SynthGraphObj.Var[S]
    outputs: Outputs[S]
    attr
    - used by graph.Attribute
```

SmallTalk FTW!

so a folder is a container, the containing elements are the main "models" and their respective attribute maps configure the UI views?

Then a default "layout" for a folder is just that - a folder tree view

A hen an egg question: does the container contain the models who carry their views as attribute, or vice versa. One would think that MVC implies the latter, because the same model can be viewed multiple times. That in turn means we need at least one object type for widget, possibly one per widget.

```
DoubleWidget:
  model: DoubleObj
  attr
    visible
    enabled
    min
    max
    renderer: String of "slider", "rotary", etc.

BooleanWidget:
  model: BooleanObj
  attr
    visible
    enabled
    renderer: String of "checkbox", "toggle-button", ...
```

Figure 1: Concept for simple UI definitions.

Discussion

User values are then stored in the `controls` field of the sonification object, a map from control names to double objects. If we ensure that an `S#Var` is always used here, we can uses that as a reference for further UI coding. Let's assume we had the possibility to create an expression for an option of a map entry, like `controls.getRef(key): DoubleOption`. Then we could somehow boil that down to a flat expression, like `.getOrElse(DoubleObj.Const(0)): DoubleObj`.[1] Then we can transform it into a boolean, like `.>(thresh): BooleanObj` and store it inside a user interface element attribute such as `visible` or `enabled`.

Then the next question would be how we can define those expressions and expression-combinators from within *Mellite*? This looks like a bit more work. Perhaps the best option here is to have built-in context menu functions to produce the most common expressions.

Now if we have a simple additional UI editor anyway—because we want to be able to style the elements slightly, position them etc.—then it would make sense to remove `graph.UserValue` and simply use the existing `graph.Attribute`. And then indeed we should introduce a *ScalaCollider*-like `"key".kr` shortcut. The sonification editor could still listen to graph updates and "notify" the developer about newly appearing attributes, offering to create UI elements for them. We could repurpose `.controls` so that we rely on the double-object's attribute map to store the type of widget and widget parameters, adopting `ParamSpec` from *Wolkenpumpe*, etc. We wouldn't even need the `.getRef` because now we could refer to the `S#Var` directly.

Since this principle is general, it should be decomposable into things directly within *SoundProcesses*. For example, we could even completely diminish `.controls` if attributes are stored with the `Proc`'s attribute map. Instead we would just associated a simple `Panel` object. It makes sense to start a new sub-module within the *Lucre-Swing* project.

3.2 Conditional Synth Sub-Graphs

This is a quite complex issue, as it touches the internals of *SoundProcesses* (how and when to build synth graphs, how they access the outside context, etc.) and the serialisation of synth graphs. One has to distinguish two cases

- determination at initialisation time *on the client side*
- determination *on the server side* (the distinction between scalar rate or initialisation time and audio rate is less relevant)

It also affects the question whether a sound model should be described by one monolithic synth graph or rather by a connected collection of graphs.

Discussion

Let us first explore the case where branching is solely solved **inside one monolithic synth graph function.** There are two possibilities: Stick to the `GE` (graph elements) "interpreter" approach, where a `SynthGraph` is directly serialised. This is the more "functional" style. Or allow instead to execute a `Function0[SynthGraph]`, similar to the `Action` object, and going for a more "imperative" style.

In the functional style,[2] we would have a restricted set of pseudo-UGens or graph elements that handle the branching. For client-side initialisation time, we might have something like

1. Another possibility is to have `controls.getOrElseRef(...): DoubleObj` and avoid having to introduce the option object, which is probably a good ideas as we still have no support for objects with type-parameters and would have to define an option object for any peer type.

2. This becomes a sort of proof for "Greenspun's tenth rule": `https://en.wikipedia.org/wiki/Greenspun's_tenth_rule`

```
// possibly: IfAttrGt.ir
val amp: GE = ???
val res: GE = IfAttrGt("key", 0) {
  WhiteNoise.ar(amp)
} ElseIfGt(1) {
  SinOsc.ar(441) * amp
} Else {
  DC.ar(0)
}
```

The disadvantages here are that

- we might need a large number of graph elements for the possible comparison cases, `IfAttr("boolean")`, `IfAttrEquals("key", value)`, etc.
- there is no possibility to perform general arithmetic operations on the attribute value. For example, if the attribute is a buffer, we might want to check the duration of the buffer. It does not make sense to define an entire new language interpreter here.

A potential advantage is that we could extend this syntax to theoretically support real-time branching:

```
val amp: GE = ???
val res: GE = IfAttrGt.kr("key", 0) {
  WhiteNoise.ar(amp)
} ElseIfGt(1) {
  SinOsc.ar(441) * amp
} Else {
  DC.ar(0)
}
```

However, this produces two difficult implementation problems:

- We need to capture the dependency of the inner branches on the outer context, i.e. here the use of the signal `amp` that must—upon expansion—be modelled by a bus between two synths, where the branches are conditionally created synths.
- We need to capture the dependency of the outer context on the result of the conditional, i.e. here the further use of the signal `res` that again must be modelled by a bus between two synths.

If this implementation is solved, it will make this approach very powerful, though, and client vs. server solution could be transparently determined from pattern matching. E.g.

```
If ("key".ir > 0) { ... }
```

This would capture a `BinaryOpUGen` which could be analysed for its rate and operands, possibly erasing to a real boolean check on the client side if the rate is `scalar`, one operand is an attribute and the other a constant. This way, still arbitrary arithmetic expressions can be used, although without the guarantee that they are indeed erased to a boolean type and so might end up creating a second (possibly paused) synth.

The pausing synth of course could cause other problems, but it might be a sensible approach, as the programmer can still explicitly go for entirely non-paused branches using standard UGen comparators such as `sig_==` and multiplications with zero or one.

In the imperative style, instead of evaluating code that produces a SynthGraph and serialising that graph, we serialise a function that when applied creates the SynthGraph. This makes it possible to ask directly for the value of an attribute and branch based on it:

```
val amp: GE = ???
val keyVal: Double = "key".value
val res: GE = if (keyVal > 0) {
  WhiteNoise.ar(amp)
} else if (keyVal > 1) {
  SinOsc.ar(441) * amp
} else {
  DC.ar(0)
}
```

This looks nice at the first look, but indeed is not that different from the `If ("key".ir > 0) ...` variant that we sketched out before using UGens only. It is also clear that with this imperative example, we do not yet have a solution for real-time control of branches. In Scala, we also cannot enforce the side-effect freedom of the function, while the function might have to be evaluated multiple times, for example if we wish to scan the sources of the synth graph for matrix variables. We introduce two control exceptions now that have to be thrown if the context information is insufficient to build the graph; first within the function that builds the SynthGraph, then in the UGen builder when the graph is expanded. In other words, this also introduces additional complexity, even before we evaluate the performance impact by having to go through one extra interpretation step. It would thus appear that the functional style variant is generally preferable, although we do not know yet whether an implementation is feasible.

Going away from a monolithic synth function, branches could also be represented by **multiple** Proc **or generally** Obj **instances**. For example, if one takes the Ensemble type, there is a boolean expression that controls its playback. We would then relax the constraint that Sonification wraps a single Proc instance, instead allowing any type of Obj to be used as peer, for example a Folder or Ensemble. The user interface could view models from not one but multiple objects, or models (attributes) could be shared among multiple objects. Then communication between branches must be explicitly established by the programmer, using regular Output instances and connections. The aural views would automatically manage bus allocation as disabled (paused) objects would not produce a running synth (?). It feels almost like an *orthogonal* approach to the monolithic one, in the sense that we could implement both independent of each other, as either might be better suited in a particular scenario. This multiple-objects approach certainly requires more work for the user interface, in order to make the management enjoyable without too much confusion. Additional problems might be the correct selection of attribute maps and the question of how running time information is provided for the objects that are facultatively added to the mix.

Summary

I would therefore think that the strategy should be to start with the implementation of the If graph element and then evaluate how far this gets us. There are most likely follow-up problem that must be addressed, such as the increased demand of inter-synth bus channels that asks for a smart bus-reuse in the topology, an endeavour currently not yet undertaken.

4 Overview of Areas for Improvement

Fig. 2 shows the main user interface windows for interaction of the domain scientists, along with a list of possible improvements. In terms of integration, focus should probably lie on seamless connection to the common workflow of these users, as shown in Fig. 3.

5 Conditional Synth Sub-Graphs – Implementation

The case where pattern matching leads to early client-side resolution seems easy enough, so the implementation plan needs to focus on the question how to coordinate multiple synth instances on the server.

Let's take the original example again:

```
val amp: GE = ???
val keyVal: GE = "key".kr
val res: GE = If (keyVal > 0) {
  WhiteNoise.ar(amp)
} Else If (keyVal > 1) {
  SinOsc.ar(441) * amp
} Else {
  DC.ar(0)
}
Out.ar(0, res)
```

There are two possibilities for a branch activation: One is that a paused synth is resumed, the other is that the synth is restarted. This has a huge impact on UGens that change over time. Take for instance a `Line` object. Both scenarios might be useful, thus we either define two different `If` UGens, or we add an auxiliary UGen `Resumed`, or we add a second parameter to `If` that specifies the behaviour, or of of the latter: `If (cond: GE, restart: GE = 0)` and `Resumed()`. Where `Resumed` could be a simple control input for a predefined name such as `$resumed`. This would allow the programmer to decide if and how to reset particular UGens when the branch is activated.

Technically the pausing is probably much simpler than restarting the synth, which would require a client responder and might run into trouble with high frequency triggering? In contrast, we do have a `Pause` UGen that operates at control rate. The restarting might also be implemented using `Action` objects. For paused synths, the question remains whether we get a high CPU peak when initialising the synth.

The most flexible approach now would be to pause—no need to add another argument to `If`—and add the `Resumed` UGen.

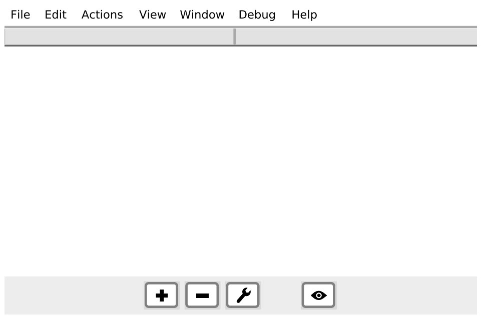
*

The problem in the example with connecting buses for `amp` and `res` is that if we don't want to accept latency, this cannot be a feedback. That is, there is another implicit split in the outer context, with one part containing everything but the sub-trees that use `res`, and one part containing the sub-trees beginning at `res`, so that the topology would be `amp -> branches -> res-usage`. So we need a general mechanism for partitioning the graph into partial graphs, i.e. around boundaries.

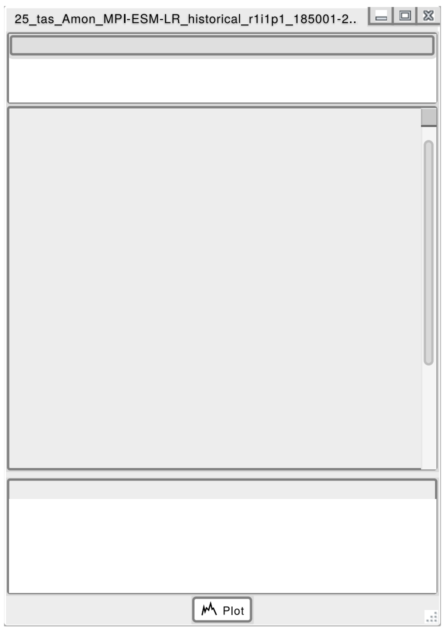
An experimental project has been set up to develop syntax and implementation: `https://github.com/iem-projects/sysson-experiments`. Syntax is currently developed, along this idea: `http://stackoverflow.com/questions/39036332`.

The outcome of these experiments are discussed in next technical report.

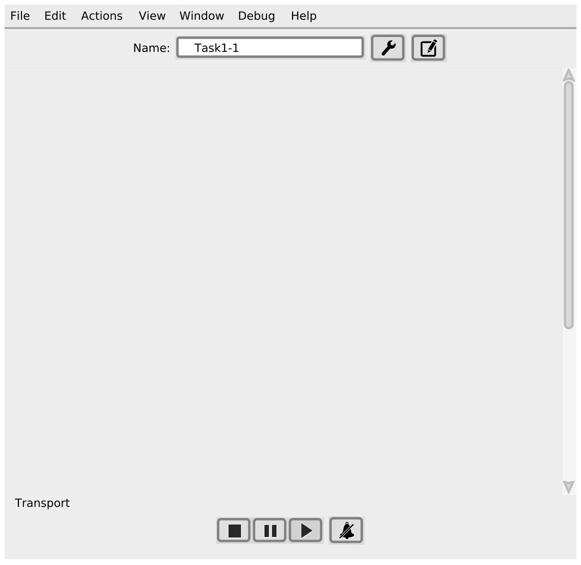
"User Perspective"



Workspace
- direct embedding of data source entries
- simplification of artifact-location
- possibly: unification with simple-UI
  (container = folder view, layout)

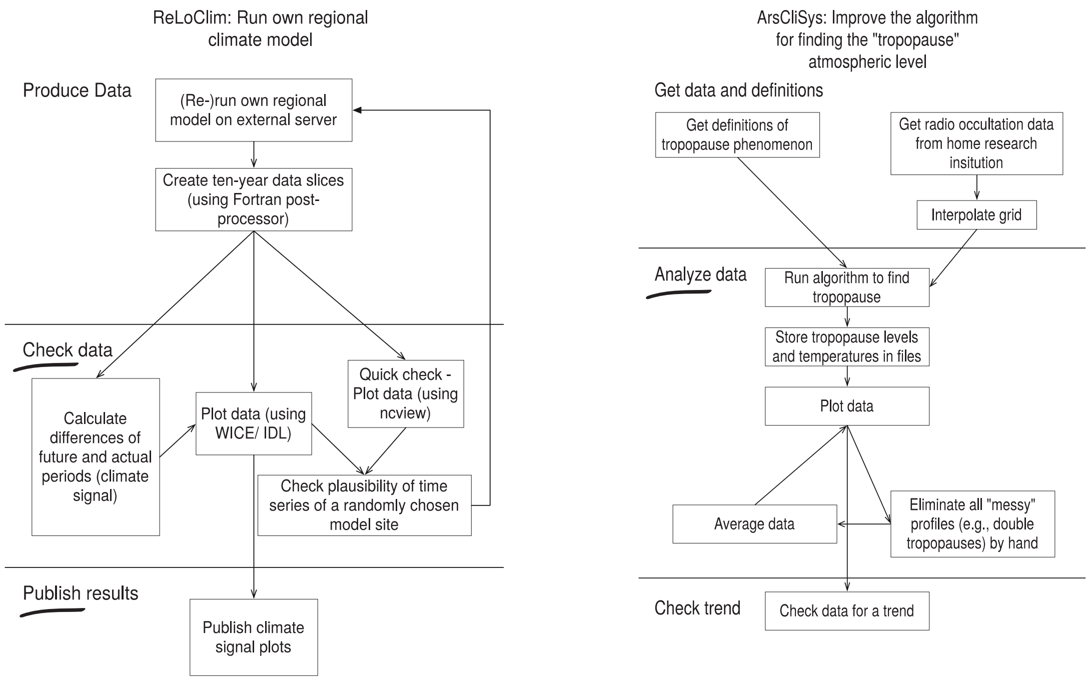


Data Sources
- improve 1D plots / plot selection
- table view (data cells)
- reference markers for REPL ?
- easy creation of subsets of data



Sonification Editor
- improve UI controls
- improve transport controls
- improve error handling
- improve UI feedback
  (e.g. render information about
   matrix size, number of sound
   sources etc.)
- possibly: "lock" / "read-only" /
  "open-copy" for presentations

Figure 2: Current problems and areas for improvement from the perspective of the domain scientist (Rutz, Vogt and Höldrich 2015).



ReLoClim: Run own regional climate model
Produce Data
(Re-)run own regional model on external server
Create ten-year data slices (using Fortran post-processor)
Check data
Calculate differences of future and actual periods (climate signal)
Plot data (using WICE/ IDL)
Quick check - Plot data (using ncview)
Check plausibility of time series of a randomly chosen model site
Publish results
Publish climate signal plots
ArsCliSys: Improve the algorithm for finding the "tropopause" atmospheric level
Get data and definitions
Get definitions of tropopause phenomenon
Get radio occultation data from home research insitution
Interpolate grid
Analyze data
Run algorithm to find tropopause
Store tropopause levels and temperatures in files
Plot data
Average data
Eliminate all "messy" profiles (e.g., double tropopauses) by hand
Check trend
Check data for a trend

Figure 3: Results from workflow analysis of climatologists (Rutz et al. 2013).

6 Libraries and Modules

This section serves as a status-quo document, detailing the current module structure of the software, with possible points of entry for further development and improvement.

SysSon is built with the *sbt* build tool.[3] *Sbt* manages module and library dependencies though a software called *Ivy* and using so-called *Maven* artefacts. Such artefacts are composed of a group-identifier, artefact-identifier, and a version string. For example, SysSon itself has group-id `at.iem.sysson`, artefact-id `sysson`, and current version `1.8.0-SNAPSHOT`. The version scheme is the one proposed as *Semantic Versioning*[4], i.e. `MAJOR.MINOR.PATCH`. The `MAJOR` version indicates an entire new architecture, where the first stable generation is usually indicated by the digit `1`. The `MINOR` version indicates incremental changes, while the `PATCH` version indicate bug fixes without change in functionality. In the Scala eco-system, the terminology is slightly different, where the major version is called "epoch", the minor version is called major version and the patch version is called minor version. So Scala 2.11.8, the latest release, has epoch 2, major version 11, and minor version 8 in the Scala terminology. We will henceforth use this naming scheme. The special suffix `-SNAPSHOT` indicates that the version is not a stable released artefact but work in progress. Stable artefacts of libraries are published to a repository such a the public Maven Central,[5] and build tools can thus automatically download required artefacts based on a description of an application's dependencies.

The Maven based build tool *sbt* assumes a *binary compatibility* between all minor versions, whereas the major version must be incremented when a binary incompatible change is introduced. A binary incompatibility means that the Java byte code of the library contains changes that make it unsafe for use with a caller that was compiled against a different version. This happens for

3. `http://www.scala-sbt.org/`
4. `http://semver.org/`
5. `http://central.sonatype.org/`

example if a method has been removed from public API or changed in signature. Sbt automatically chooses the highest minor version if there are several transitive dependencies on the same artefact but with different minor versions, while it warns when transitive dependencies exist for the same artefact with different major versions. This also applies to the Scala version a library was compiled against, meaning that a binary file resulting from compilation against Scala 2.10.x (where x is any minor version) is binary incompatible with a binary file resulting from compilation against Scala 2.11.x (with 2.11.8 being the currently most recent release version). To ease this difficulty, sbt has introduced a mechanism called cross-versioning. Often Scala releases are still source compatible, and thus it is possible, for example, to compile the same source code against Scala 2.10.x and 2.11.x without changes. The artefacts will then be "tagged" by appending the major Scala version, e.g. the artifact-id becomes `sysson_2.11` when compiling against Scala 2.11.x (this is independent of the regular Maven version).

This versioning scheme can be observed in the dependency graph that shows all the libraries and transitive libraries,[6] generated by the *sbt-dependency-graph* plugin[7] and shown in Fig. 4.[8] We can see the SysSon main artefact on the left-hand side with arrows pointing to immediate dependencies (libraries), which in turn point to other transitive libraries. Some libraries are published as a set of related artefacts, often bound together by one "virtual" meta-package. For example, the *Mellite* computer music environment is actually composed of two artefacts `mellite-core` and `mellite-views`, combined as virtual artefact `mellite`. The `-views` artefact connects to the graphical user interface, whereas `-core` only contains data structures. Since `-views` depends on `-core` but not vice versa, it means we can develop a library merely based on the data structures without requiring the dependency on any GUI.

6.1 Licensing Questions

This approach helps keeping the code base modular, and furthermore can improve the licensing situation: In general we try to avoid dependency on GPL licensed code,[9] because it enforces the GPL on the entire product, something that is not the case with the LGPL[10] or even more permissive licenses. We try to decouple GPL dependencies as far as possible, allowing for LGPL terms in most cases. GPL artefacts are often commercially developed products, because a commercial client company will likely not want to release their own product as open source under GPL terms and is thus forced to buy a commercial license from the library vendor, an approach that is called dual licensing. For example, the *iText* PDF library is available for free under GPL terms or by buying a commercial license that allows the client to develop their product without requiring to open source it. The same is true for the *Berkeley DB* database (developed by Oracle) and the *WebLaF* user interface look-and-feel. We currently do not own any commercially bought licenses, and thus **SysSon must be licensed under GPL version 3** terms. However except for the mentioned GPL based libraries, we have control over most other libraries, and thus in the future a different license might be chosen if replacements are found for the GPL libraries (or commercial licenses purchased).

6.2 Description of Modules

We will now give a brief overview of the modules and libraries on which SysSon relies and which were shown in Fig. 4.

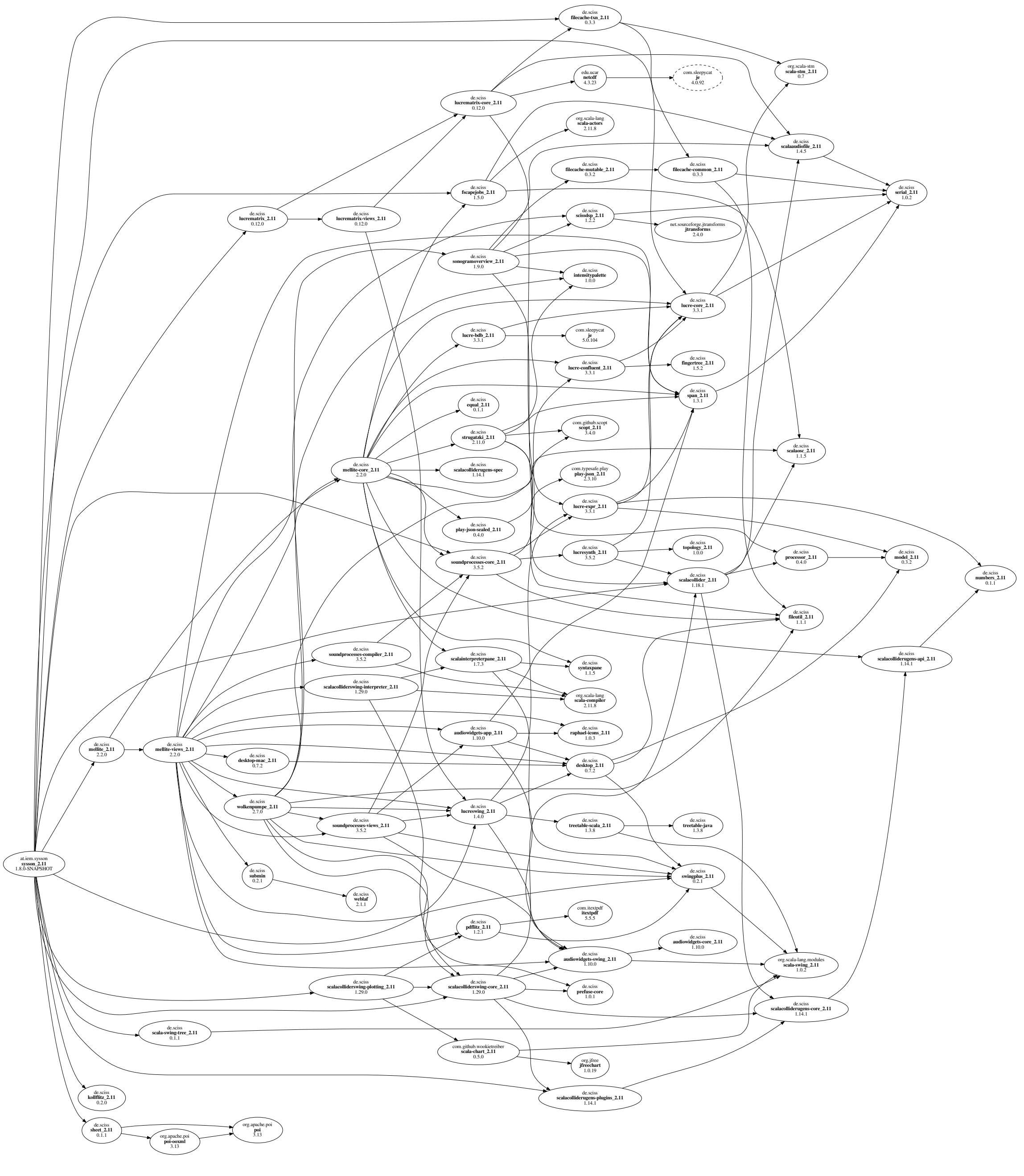
6. Transitive means a library A is used by another library B, and B is used by SysSon. Then A is a transitive dependency of SysSon.

7. The call is `sbt dependencyDot`.

8. In the figure, lower level transitive third-party dependencies have been omitted for a clearer overview.

9. `https://www.gnu.org/licenses/gpl.html`

10. `https://www.gnu.org/copyleft/lesser.html`



de.sciss filecache-txn_2.11 0.3.3
org.scala-stm scala-stm_2.11 0.7
edu.ucar netcdf 4.3.23
com.sleepycat je 4.0.92
de.sciss lucrematrix-core_2.11 0.12.0
org.scala-lang scala-actors 2.11.8
de.sciss scalaaudiofile_2.11 1.4.5
de.sciss filecache-mutable_2.11 0.3.2
de.sciss filecache-common_2.11 0.3.3
de.sciss fscapejobs_2.11 1.5.0
de.sciss serial_2.11 1.0.2
de.sciss lucrematrix_2.11 0.12.0
de.sciss lucrematrix-views_2.11 0.12.0
de.sciss scissdsp_2.11 1.2.2
net.sourceforge.jtransforms jtransforms 2.4.0
de.sciss sonogramoverview_2.11 1.9.0
de.sciss intensitypalette 1.0.0
de.sciss lucre-core_2.11 3.3.1
de.sciss lucre-bdb_2.11 3.3.1
com.sleepycat je 5.0.104
de.sciss lucre-confluent_2.11 3.3.1
de.sciss fingertree_2.11 1.5.2
de.sciss span_2.11 1.3.1
de.sciss equal_2.11 0.1.1
de.sciss strugatzki_2.11 2.11.0
com.github.scopt scopt_2.11 3.4.0
de.sciss scalaosc_2.11 1.1.5
de.sciss mellite-core_2.11 2.2.0
de.sciss scalacolliderugens-spec 1.14.1
com.typesafe.play play-json_2.11 2.3.10
de.sciss lucre-expr_2.11 3.3.1
de.sciss play-json-sealed_2.11 0.4.0
de.sciss topology_2.11 1.0.0
de.sciss lucresynth_2.11 3.5.2
de.sciss processor_2.11 0.4.0
de.sciss model_2.11 0.3.2
de.sciss soundprocesses-core_2.11 3.5.2
de.sciss scalacollider_2.11 1.18.1
de.sciss numbers_2.11 0.1.1
de.sciss fileutil_2.11 1.1.1
de.sciss soundprocesses-compiler_2.11 3.5.2
de.sciss scalacolliderugens-api_2.11 1.14.1
de.sciss scalainterpreterpane_2.11 1.7.3
de.sciss syntaxpane 1.1.5
de.sciss scalacolliderswing-interpreter_2.11 1.29.0
org.scala-lang scala-compiler 2.11.8
de.sciss audiowidgets-app_2.11 1.10.0
de.sciss raphael-icons_2.11 1.0.3
de.sciss mellite_2.11 2.2.0
de.sciss mellite-views_2.11 2.2.0
de.sciss desktop-mac_2.11 0.7.2
de.sciss desktop_2.11 0.7.2
de.sciss wolkenpumpe_2.11 2.7.0
de.sciss lucreswing_2.11 1.4.0
de.sciss soundprocesses-views_2.11 3.5.2
de.sciss treetable-scala_2.11 1.3.8
de.sciss treetable-java 1.3.8
at.iem.sysson sysson_2.11 1.8.0-SNAPSHOT
de.sciss swingplus_2.11 0.2.1
de.sciss submin 0.2.1
de.sciss weblaf 2.1.1
com.itextpdf itextpdf 5.5.5
de.sciss pdflitz_2.11 1.2.1
de.sciss audiowidgets-core_2.11 1.10.0
de.sciss audiowidgets-swing_2.11 1.10.0
org.scala-lang.modules scala-swing_2.11 1.0.2
de.sciss scalacolliderswing-plotting_2.11 1.29.0
de.sciss scalacolliderswing-core_2.11 1.29.0
de.sciss prefuse-core 1.0.1
de.sciss scalacolliderugens-core_2.11 1.14.1
de.sciss scala-swing-tree_2.11 0.1.1
com.github.wookietreiber scala-chart_2.11 0.5.0
org.jfree jfreechart 1.0.19
de.sciss kollflitz_2.11 0.2.0
de.sciss scalacolliderugens-plugins_2.11 1.14.1
de.sciss sheet_2.11 0.1.1
org.apache.poi poi-ooxml 3.13
org.apache.poi poi 3.13

Figure 4: Module/library dependencies for SysSon version 1.8.0

Mellite By far the strongest dependency is on the *Mellite* computer music environment. In fact, SysSon can be understood as a domain specific extension of *Mellite*, adding abstractions for sonification and matrix data, although the GUI is also slightly different. Mellite is a graphical front end and application built on top of the *SoundProcesses* library. It introduces a tree based workspace structure with graphical views and editors for various objects.

Artefacts: `-core` contains objects in addition to *SoundProcesses*, such a color, the workspace and cursors abstractions, and a set of editing operations. `-views` contains the graphical user interface, and undo-redo editing facilities.

Assessment: The workspace abstraction should probably be moved directly into the *Sound-Processes* project, making the `-core` artefact superfluous. It could be useful to define public API in a separate `-api` artefact and create sub-modules for specific editors that do not necessarily have to be bundled in every application. An example is the *Wolkenpumpe* live improvisation tool that is currently a dependency of `-views`. A big challenge is the inter-linkage between visual representations and text-based editing. While the model-view approach already makes programmatic changes instantly visible across the UI, there is no global UI protocol through which the views "know each other" such that for example selecting one view implies a visual highlight on related views. Also because of the traditional windowing system, there are challenges in connecting objects (e.g. through "cords") across different windows. A radical revision of *Mellite* could focus on making text-editing a first class citizen, e.g. by allowing to activate a text-based view for every object on the UI. In terms of SysSon, an interesting model to look at might be the "notebook" approaches taken with software such as Jupyter.[11]

Lucre-Matrix Defined as a separate library, this module provides a matrix data abstraction, bridging *NetCDF* and the transactional-memory model defined by *Lucre* and underlying *Sound-Processes*. It makes it possible to build data-flow graphs, e.g. by applying a dimensional reduction operation to an input matrix, yielding a graph that constitutes a new compound matrix object.

Artefacts: `-core` contains the data structures, and a mechanism for caching matrix data in sound files (SysSon's way of passing matrix data to *ScalaCollider*). `-views` contains the graphical user interface element, noticeable as the matrix editor component in SysSon's sonification editor.

Assessment: The number of operations that can be performed on the matrices is quite small. It would also be worth exploring the possibilities of using a different matrix abstraction than the one of *NetCDF* as blue print, linking this library closer with standard "data science" frameworks. Another huge question arises from the way sonification models are built in SysSon. It appears that it may become much more important to be able to pre-process matrix data before they become part of the real-time sound synthesis. One way to conceive this is through additional "graph elements" in the sonification synthesis definition, i.e. as is done with elements such as `Dim#values`. The down-side of this is that we risk duplicating the functionality on the "Lucre side" vs. the "ScalaCollider side". Therefore, the development of *Lucre-Matrix* very much depends on the evolution of the sonification model in SysSon and the way in which one will be able to represent "sonification programs", the way in which these programs can be "co-edited" with synthesis definitions.

Lucre The name *Lucre* serves as a placeholder for an approach to memory and data modelling based on software transactional memory (STM). *Lucre* defines an abstract system layer that usually occurs in objects as a type parameter `S`, e.g. `Sonification[S]`. At runtime, different systems can be initiated: an ephemeral pure in-memory system, a durable system where objects are automatically persisted to hard-disk, and a confluent system which extends the durable system with a version history of the modification of objects. This kind of temporal database approach has

11. `http://jupyter.org/`

been developed extensively through my thesis (Rutz 2014b), and forms the basis for *SoundProcesses*, *Mellite*, and SysSon respectively.

Artefacts: There is currently no umbrella meta package. `-core` is now a rather comprehensive artefact that includes four packages `stm`, `geom`, `data`, `event`. The reactive event propagation mechanism is now baked into the fundamental `stm` package (this was not the case with older versions). This package defines the basic types `Sys` (memory-model abstraction), `Txn` (transaction), `Cursor` (transaction initiator), `Elem` (an element that publishes events and that can be copied), and `Obj` (an element that in addition to `Elem` has a unique identifier and an attribute map). Most data types in SysSon and and *SoundProcesses* inherit from `Obj`. The `event` package defines the infrastructure for event dispatch (publisher-subscriber mechanism). The `data` package contains several data structures needed for the implementation of various systems, e.g. lists, trees, spatial search structures. The second artefact is `-expr` that introduces reactive expressions, with implementations for primitive types such as strings, boolean values, integer and floating point numbers, time spans (a pair of 64-bit integers), collections—`BiPin` for a "flat" sequence or breakpoint function, `BiGroup` for a sequence with possibly overlapping elements—and `Artifact` (file object). While in-memory and ephemeral-durable systems are defined in the base package, actual database back-ends are defined by `-bdb` and `-bdb6`, both using the *Berkeley DB Java Edition* (BDB DB JE), introducing the dependency on the GNU GPL. `-bdb` uses version 5, compatible with the JVM version 6, and `-bdb6` uses version 6, requiring JVM version 8. The `-confluent` artefact implements the confluently-persistent (versioned) system.

Assessment: Changes in *Lucre* fundamentally affect all parts of SysSon, as it is a deep dependency that is used by *SoundProcesses*, *Mellite*, *Lucre-Matrix* and others. The current version is stable and reliable but has a number of shortcomings: On the programming level, the syntactic overhead of carrying around system type parameter `S <: Sys[S]]` as well as transaction context method argument `implicit tx: S#Tx` is high, making it a rather bad fit for a light-weight domain specific language to define snippets in *Mellite* or SysSon. In the medium term future, this might be alleviated by the next generation Scala language (Dotty research project), where on the one hand type projections are seriously constrained, probably making it necessary to redefine the way system abstraction is visible in *Lucre*, but on the other hand also allowing to "hide" type parameters as type members and implicit arguments as implicit capabilities of a function's return type. However, this is a perspective of several years into the future, so in the shorter term, probably the most convincing solution is to add a separate DSL layer where transactional semantics are injected into the interpreter's code execution. Another question concerns the cost of `Obj` instances. In previous versions we distinguished between expressions and these objects, making it for example possible to have cheap constant values. Constants appear very frequent, for example if one edits a numeric value in SysSon through a slider, many new objects have to be created (and persisted to the database) for each intermediary state in the dragging operation. With `Obj` being basically the atoms for most operations, this means that now we have to allocate a lot of throw-away object identifiers and the space usage in the database is much higher than with the old "simple" constants, where only the containing `S#Var` variable (mutable placeholder) was requiring a one-time allocation. It is thus desirable to rethink this atomicity and try to combine the simplicity of one layer `Obj`-everywhere with the performance of having cheap constants. Somehow related to this is the lack of any sort of automatic garbage collection for transitory values. If we had a reference counting ability, we could evict objects from the database once they are not reachable any longer, or those transitorily produced during a transaction without having an actual linkage to the workspace at the commit-time of the transaction. A reference tracking could also help with automatically detecting "bottom-up" dependencies between objects in the GUI. Furthermore, we need to assess the necessity for the full confluent persistence in versioning—while theoretically elegant, we cannot abstract over the meld operation in the system layer, because there is no equivalent for ephemeral systems. Therefore, we are forced to introduce a object-copying mechanism agnostic of the system capabilities. Also, confluent meld could become expensive if used excessively, as the version paths grow linearly with the levels of meld. It might thus be more pragmatic to simplify the versioning system by moving to the less powerful partial persistence and improving on the definition of object-copying. Object-copying could also preserve information about the source object from which a copy was made, as this information is currently lost and thus undermines the effort to version objects. Another problem is the size of transactions. Transactions can become quite large which poses a problem for the database commit phase, as ACID requires that we wait for the durability guarantee at the end of each transaction. There is a ticket[12] to evaluate other database back-ends, something that will also interact with the *Serial* module. It seems that especially in concurrent situations with conflicting transactions, we face performance problems or even impasses where the database times out. Going away from STM as the memory model would however require an entire new architecture, and so far STM has proven a very robust approach, so it is probably advisable to stay with it for the time being, instead evaluating strategies to undo the bottle-necks with concurrent transactions.

So far we have used BDB DB JE 5 to allow SysSon to run on legacy computers with Java 6, such as Mac OS X 10.6. The next version of SysSon can probably drop that constraint and require presence of Java 8 and thus the newer database version.

Scala-STM The reference implementation for software transactional memory in Scala.

Assessment: This library is quite stable, error free, and fast. The only bottle neck is using a decider-hook when coupling to a database transaction that must be completed with the *Scala-STM* transaction. For performance reasons, implicit transaction values have to be carried around when defining a transactional API, although the library also allows the lookup of transactions from a thread-local variable and the automatic coalescence of transactions if one unknowingly starts a new `atomic {}` block. Care has to be taken in nested functions to prevent catching a stale implicit transaction context from an other function. From experience, the best solution to this problem is to move transactional methods all to the top-level of an object and avoid nesting them.

Lucre-Swing Published as a separate library, this module provides a set of simple widgets that can interface with expressions and other data structures from the *Lucre* project.

Serial This module defines a small serialization layer used by *Lucre* and other libraries.

Assessment: The module is optimised for speed and size, making serialization quite efficient. However, a current drawback is vulnerability to schema evolution. Serialised objects simply become unreadable if the schema is altered without any precautions in place that can parse older versions. In many cases it may simply not be possible to parse older versions, and in those cases it would be required to entirely "upgrade" an objects, for example allocating additional objects. While such upgrades are currently not foreseen, there are also made nearly impossible by the fact that the same entity might be written multiple times in the database. If every `Obj` with an identifier `S#ID` would be written as a small reference to one and the same location in the database, perhaps upgrading schemas would become feasible. For robustness, it would also be very helpful if an automatic export to other formats such as JSON would be available. This would require the serialisation API to change, for example including field names. Another current limitation is the prohibition of reference cycles within the same serialisation, something that can currently be worked around by the indirection of a `S#Var`. Finally, one must currently write serialisers by hand which from our experience is *very* prone to errors, and also clutters the source

12. `https://github.com/Sciss/Lucre/issues/6`

code unnecessarily. After deciding on an updated API, it might thus be advisable to look into macro-based synthesis of serialisers.

SoundProcesses Sound synthesis abstractions are provided by the *SoundProcesses* module which can be understood as bridge between lower-level sound synthesis library *ScalaCollider* and the *Lucre* memory model. *SoundProcesses* defines an abstraction `Proc` that encapsulates a sound synthesis function with a dictionary of controllers and resources. It makes it safe to connect and alter sound processes transactionally, automatically providing correct node topology, resource management, and real-time dispatch. A notion of aural-views is introduced, translating the MVC concept from the visual domain to sound. *SoundProcesses* is described in detail in my thesis, as well as in several conference papers (e.g. Rutz 2014a).

Artefacts: There is currently no umbrella meta package. The `-synth` sub-module (full artefact-identifier `lucresynth`) is a transactional layer on top of *ScalaCollider*. For every imperative API in *ScalaCollider* we have a transaction-safe equivalent here, e.g. playing a `Synth`, allocating and filling buffers, etc. A new *Lucre* system layer is defined that handles OSC messages to the *SuperCollider* server. The `-core` sub-module then is the meat of *SoundProcesses*. It defines the `Proc` object, several general purpose objects such as folders (reactive lists), graphemes (breakpoint functions), actions (functions triggered for example by a `Proc`), fade-specifications, and curvatures. Furthermore, an offline and real-time scheduler and transport mechanism, and a timeline object. The `AuralView` type introduces the model-view separation on the audio level. The `graph` sub-package defines a number of pseudo-UGens necessary for accessing the attributes associated with a `Proc` (controllers, buffers, etc.). The `-compiler` sub-module defines a small abstraction for an embedded Scala compiler, which in turn is provided by the *Scala-Compiler* dependency. This compiler is used in the user-interface for ad-hoc compilation of source code fragments. The `-views` sub-module defines a number of specific widgets, such as a volume meter, time display and transport control buttons.

Assessment: The automatic resource management and OSC timing can be quite intricate, and as a result, some of the code base internals are quite complex and could benefit from simplification and modularisation. The aural view implementations also become quite complicated because of the necessity to update actual real-time views according to the appearance and disappearance of model dependencies (e.g. when a missing buffer association is added or removed). This is also in part due to the way synth graphs are assembled, currently requiring control exceptions to be thrown with partially built synth graphs to be properly maintained and incrementally updated. Composition poses another difficulty. For example, in SysSon, the `Sonification` object encapsulates a `Proc`, but it must make its own attribute map available for the aural-view building, requiring a variant of the aural view that relies on the internal implementation of the aural view of `Proc`. A general drawback is that we cannot directly monitor "capabilities" of a sound process introduced by specific (pseudo-)UGens. For example, in SysSon we must scan the synth graph for objects that have consequences on the sonification editor, such as matrix and dimension variables appearing or disappearing. Providing for better "functionally reactive" filtering of the synth graph could improve the situation. In general, the monolithic shape of the synth graph could be seen as problematic. For example, in the SysSon "Science by Ear 3" workshop, some participants familiar with *SuperCollider* felt constrained by the lack of support for small functions, patterns and routines, and so on. Chaining this functionality together with imperative `Action` objects seems rather inelegant. Currently, there are no guarantees with respect to the precise OSC bundle scheduling. The scheduler is based on a client CPU timer, bundles are properly constructed with respect to transactional atomicity, but it is not possible to ensure that for example an event happening exactly one second after another will execute on the server after the corresponding number of sample frames. This is mainly a problem for timeline composition and less so for sonification, but it will become a problem if we add more client side timing, such as routines, to sonification. Another timing issue occurs with preparation times. There is a rudimentary mechanism in place to allow for a `Proc` to prepare resources asynchronously—e.g. loading a very large buffer—and this is also used by SysSon for caching matrix data. But again, for sequencing objects, we need to add information about expected latencies or at least allow for the customisation of the preparation pre-delay. In terms of the GUI, it may seem counter-intuitive that one has to wait sometimes quite long for data to be prepared after pressing the play button. We need at least a clear visual indicator here.

ScalaCollider *ScalaCollider* is a client for the *SuperCollider* server, driving its real-time sound synthesis engine. It is stable and quite feature complete and should allow anyone familiar with *SuperCollider* to be able to quickly write sound synthesis definitions, even if the implementing language—Scala—is a different one.

Artefacts: The unit generators (UGens) and their basic API are separately defined in the `-ugens` projects with its own sub-modules `-api` (abstract UGen API only), `-core` (standard UGens bundled with *SuperCollider*), `-plugins` (incomplete set of third-party UGens), and `-spec` (XML meta-data with UGen descriptions). The main project consists of one artefact only. It adds an actual UGen graph builder and server object, bus abstractions, an imperative layer for running synthesis processes. Another project `-swing` introduces an integrated development environment (IDE), a desktop application a bit simpler than the original *SuperCollider IDE*. It is split into the sub-modules `-core` which contains individual widgets (e.g. node tree visualisation, server status panel), `-interpreter` which connects to the Scala interpreter, `-plotting` which adds simple plot command based on *Scala-Chart*, and `-app` which bundles all together as a standalone desktop application.

Assessment: *ScalaCollider* is a more or less "finished" product, the architecture is simple and works as expected. It currently does not include any abstractions for client-side scheduling, such a tasks or routines, but this is probably an advantage as we could in any case not use them directly but would have to translate them to the transactional *Lucre* model. The IDE uses the *Docking-Frames* window docking library which seems good enough for the task. While the *WebLaF* introduces its own docking mechanism in the upcoming release, sticking to *Docking-Frames* might be advisable also for *Mellite* and SysSon, because it is independent of the look-and-feel used and does not imposes the GNU GPL 3 license. Widget and plotting functionality could be extended and should be better integrated with or made available in SysSon whose interpreter mode is currently neglected.

Desktop This artefact provides simple abstractions for desktop application, based on Swing, such as the notion of a document in a multi-document application (MDA), or user preferences storage. For windowing, it has the notion of a shared menu-bar, and an abstraction for windows that can be realised both as standalone windows or windows internal to one screen-filling containing frame.

Artefacts: Apart from `-core`, there are sub-modules for the different operating systems, e.g. `-mac` and `-linux`. There are platform independent operations such as showing a file in the desktop manager, that are implemented differently per platform.

Assessment: The library is quite simple. Especially the menu-bar management comes still with some OS X baggage, where one shared menu-bar across windows is standard. It would also be helpful to directly incorporate some docking framework.

Audio-Widgets A library with audio-related widgets for graphical user interfaces, such as volume meters or transport controls.

Artefacts: The `-core` module contains individual widgets directly implemented on top of *Java-Swing*, while the `-swing` module wraps them for *Scala-Swing*. The `-app` module is very small and adds an abstraction for timeline based canvases.

Swing-Plus This library extends *Scala-Swing* with a few missing widgets.

Scala-Swing This library wraps the *Java-Swing* cross platform widget toolkit in an API that is more elegant to use from Scala. For example, event handling is greatly simplified.

Scala-Swing-Tree An improved model/view widget combination for tree based data structures.

Tree-Table A custom Swing widget combining a tree-like structure with a table (multiple columns) view. Artefacts exist for Java and a Scala wrapper.

Scala-Chart A simple Scala wrapper and DSL for accessing the *JFreeChart* charting library. *JFreeChart* is the most mature and widely used Java charting/plotting library based on Swing.

Sonogram-Overview A constant-Q based transform library for producing sonograms from sound files. This is used by Mellite editors to view sound files or sound regions as sonograms.

Assessment: The library implements a pyramidial decimation approach to creating multi-resolution images. The sonograms are written to cache files formally as audio files. There is currently no mechanism in place manage ad-hoc changes in resolution, such as zooming in or out of a sonogram that triggers more coarse or fine recalculations. There is a graphical glitch in region display where block boundaries are seen as vertical stripes. The algorithm is quite expensive, so calculating many sonograms for many sound files takes long time and occupies a lot of hard-disk space.

Intensity-Palette A psycho-optically optimised colour palette, adapted from the *Sonasound* software. Used as the default scheme for rendering sonograms.

Sciss-DSP A DSP library containing a few basic algorithms such as FFT, Wavelet decimation, band-limited resampling, FIR filters and window functions. FFT is now based on the *JTransforms* library.

JTransforms A fast Java FFT library with single and double precision, real and complex, and one- or two-dimensional support.

Assessment: The library is quite fast. By default parallelism is activated, but it can be configured to run single-threaded if needed.

File-Cache A simple library for managing hard-disk cache of expensive calculations. This is for example used for sonogram caching and for matrix-to-audio-file translation cache in SysSon.

File-Util A simply utility library for constructing files and paths.

Scala-Interpreter-Pane A Swing widget built around the Scala compiler/interpreter and a text editor with syntax highlight.

Assessment: The interpreter is a thin layer on top of Scala's own `IMain` interpreter structure. Similar to the *SuperCollider IDE*, pressing control and return executes the selected text or the current line. An interesting alternative could be that executed source code is appended to a sort of protocol so that it becomes easier to reproduce a REPL session. This might be similar to how the "notebook" approaches of IPython/Jupyter work. Another model currently seen is the "worksheet", where a "hot" interpreter continuously evaluates the entire source code and prints the results of each statement on the right-hand side of the code. With respect to the text editor, the code highlight is rather simple and based on the *Syntax-Pane* artefact, which is just a lexer without further code analysis. There are more advanced systems such as *ENSIME*[13] that improve on code completion, look-up, type checking and inspections, and so forth. There are also improved REPLs such as *Ammonite*[14] that claim better code completion. On the other hand it would be interesting to explore possibilities of representing placeholders for objects in the GUI in the text editor. Custom widgets such as live sliders are for example implemented in *Field*[15] (Downie 2008), and there are vocal proponents of more visually interactive systems such as Bret Victor[16].

Sheet This is a simple library for reading spread sheet data from Excel files, based on the Apache POI library. It is used as an import function in SysSon.

NetCDF The *NetCDF* is written by UCAR and was used in SysSon because all the climatology data was in `.nc` NetCDF format. The library contains its own matrix and array data types.

Assessment: The file access is not thread-safe, so one has to throw a `synchronized {}` block at it. We currently only use synchronous reading, but it might be the case that asynchronous I/O is also supported (to be evaluated).

Submin A combination of light (dark text on bright background) and dark (light text on dark background) "skins" for the Web-look-and-feel, the cross platform Swing-based look used in *Mellite* and SysSon.

Assessment: The underlying *Web-Look-And-Feel* is currently under development for a release version 1.29. The GNU GPL 3 licenses is unfortunate and we therefore strive to avoid any hard dependencies on this look and feel. All GUI in *Mellite* and SysSon is written agnotic to the look-and-feel selected. Therefore, depending on Web-LaF's docking framework would be a serious decision. The LaF is also quite large and not extremely fast in comparison to other Swing LaFs. There were a few issues with keyboard shortcuts on OS X that are probably resolved by now. On the up-side, it is probably one of the best looking LaFs available, with the light scheme directly taken from Web-LaF, and the dark scheme designed by me but closely based on the light scheme. SysSon is not yet fully tested with the dark scheme and might need some customisation to support both light and dark scheme.

Raphael-Icons The icon set used by *Mellite* and SysSon is mostly based on *Raphael.js*, and the original SVG graphics have been converted to a simple library dependency.

13. `http://ensime.github.io/`
14. `http://www.lihaoyi.com/Ammonite/`
15. `http://openendedgroup.com/field/`
16. See for example `http://worrydream.com/#!/TheFutureOfProgramming` and `http://worrydream.com/#!/MediaForThinkingTheUnthinkable`; also the work of Sean McDirmid on "live programming" (McDirmid 2013).

Assessment: Translating the SVG to code may have actually been a bad idea, as the library can become quite large. Perhaps a better approach is a fast SVG path parser and preservation of the SVG path strings.

Prefuse *Prefuse* is used to render dynamic graphs in the *Wolkenpumpe* live improvisation interface, but also other widgets such as *ScalaCollider-Swing*'s node tree view.

Wolkenpumpe Bundled with *Mellite* but should probably be optional. A live improvisation interface, otherwise not used in SysSon.

Strugatzki Bundled with *Mellite* but should probably be optional. A basic music-information-retrieval kit, based on *ScalaCollider*.

FScape-Jobs Bundled with *Mellite* but should probably be optional. A simple OSC client library to issue DSP jobs through *FScape*. We are currently rewriting *FScape* based on a finer UGen granularity, Scala, and Akka-Stream. This new version 2 will make *FScape-Jobs* eventually obsolete.

Processor An interface for defining asynchronous offline rendering jobs, extending Scala's `Future` with the ability to cancel the job and to monitor progress.

Model A simple library for publisher-subscriber functionality.

Finger-Tree A versatile functional data structure that is used here mainly as an immutable data structure for sorting intervals.

Topology A data structure for maintaining an online directed graph. When edges are inserted, the caller gets information on how vertices have to been moved to correctly preserve a linear order. Used for example by *SoundProcesses* to automatically reshuffle `Proc` instances a they are interconnected.

Span A pair of start and stop sample frames defines a time span in *SoundProcesses* and SysSon.

Scala-OSC The Open Sound Control library used by *ScalaCollider*.

Equal A simple library allowing for more type-safe comparisons of values through the `===` (triple-equals) and `!==` operators.

Play-JSON-Sealed An extension of the JSON library that is part of the *Play* framework, allowing the automatic macro based construction of serialisation code for sealed type hierarchies (algebraic data types).

Assessment: Scala does not have a standard JSON library (its standard library does contain some rudimentary JSON functions but they are generally considered inferior to any other library), and as a consequence there are now dozens of alternatives available. We found Play JSON to be fairly simple and straight forward, with its macro based serialisation generation freeing one from writing serialisers by hand. It might still be worth to evaluate other JSON libraries.

Numbers A library that enriches number types with operator often known from *SuperCollider* and *ScalaCollider*. For example we can write `2.0.sqrt` or `7.nextPowerOfTwo`.

PDFlitz A thin wrapper around *iText PDF*, allowing especially the export of any Swing or Java2D component as a PDF file. Enforces a GPL v3+ license.

Koll-Flitz A library that enriches collection types with useful operators such as iterating or mapping over pairwise groups.

References

Downie, Marc. 2008. „Field—a New Environment for Making Digital Art“. *Computers in Entertainment (CIE)* 6 (4): 54:1–54:34.

McDirmid, Sean. 2013. „Usable live programming“. In *SPLASH Onward! Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software,* 53–62. doi:`10.1145/2509578.2509585`.

Rutz, Hanns Holger. 2014a. „Sound Processes: A New Computer Music Framework“. In *Proceedings of the Joint 11th Sound and Music Computing Conference and the 40th International Computer Music Conference,* 1618–1626. Athens. `http://hdl.handle.net/2027/spo.bbp2372.2014.245`.

———. 2014b. „Tracing the Compositional Process. Sound art that rewrites its own past: formation, praxis and a computer framework“. PhD diss., Plymouth University. `http://hdl.handle.net/10026.1/3116`.

Rutz, Hanns Holger, Katharina Vogt and Robert Höldrich. 2015. „The SysSon Platform: A computer music perspective of sonification“. In *Proceedings of the 21st International Conference on Auditory Display (ICAD),* 188–192. Graz. `http://hdl.handle.net/1853/54126`.

Rutz, Hanns Holger, Katherina Vogt, Visda Goudarzi, Martin Jury and Andrea K. Steiner. 2013. „SysSon: A Systematic Design for the Sonification of Radio Occultation Data“. In *Joint OPAC-5 & IROWG-3 International Workshop on Occultations for Probing Atmosphere and Climate.* Leibnitz, AT.