RETHINKING THE SUPERCOLLIDER CLIENT...

Hanns Holger Rutz
University of Plymouth
Interdisciplinary Centre for Computer Music Research (ICCMR)
hanns.rutz@plymouth.ac.uk

ABSTRACT

We present ScalaCollider, a new client framework to connect to the SuperCollider sound synthesis server. It builds on top of the general purpose language Scala. Scala's ambition is to allow for the development of scalable systems, being equally comfortable both for small-scale scripting and large-scale modular projects.

Following an overview and comparison of the currently available clients for SuperCollider, we introduce the most important features of Scala, and show how its specific language elements can be exploited to design an elegant client that supports UGen graph composition, handles proxy objects, and restores part of the clarity lost in the entanglement of the original SuperCollider client's class library.

The problems of type-safety and approaches to concurrency are discussed, and an outlook on a high-level extension for declarative sound process specification is given.

1. INTRODUCTION

1.1 Specialised or General Purpose?

The 2002 paper [1] in which James McCartney describes the architecture of SuperCollider 3 – the division into a sound synthesis server (*scsynth*) and a language client which communicate through the Open Sound Control protocol –, is framed by the question: «Is a specialised computer music language even necessary?» McCartney presents a list of concepts employed in modern computer programming languages, many of which are implemented in the SuperCollider language (*sclang*), a language specially designed for computer music. However, he concludes, the creation of such a specialised language was mainly pragmatically motivated: No single general-purpose programming language (GPL) exhausted the list of required criteria, but theoretically a GPL could well satisfactorily drive the SuperCollider server.

At the time of that writing, McCartney expressed interest in four GPL as candidates for SuperCollider client implementations: OCaml, Dylan, GOO, and Ruby. All four – like sclang – combine object-oriented and functional programming, while three of them are dynamically-typed, and only one is statically-typed (OCaml).

1.2 The Client Landscape

Eight years later, nine clients based on GPL can be identified [1] , including Ruby and Standard ML – like OCaml a dialect of the ML language –, but also new languages that have emerged since then, namely Processing (a simplified Java dialect for graphics programming) in 2001, Scala in 2003, and Clojure (a LISP-dialect) in 2007. Figure 1 gives an overview over these languages and the corresponding SuperCollider clients.

The variety of clients reflects their different roles and scenarios, different strengths and weaknesses. p5_sc and the Python client, for instance, do not provide a means to construct graphs of Unit Generators (UGens), so they still require sclang as part of the SuperCollider interface. JCollider was not designed to compete in conciseness and expressiveness with sclang, nor as an environment for interactive computer music programming, hence it does not offer a run-eval-print-loop (REPL) mode. It refrains from elegant UGen graph syntax, as it is targeting application programming on the Java Virtual Machine where typically only a limited set of Synth Definitions are required, as in the case of the SwingOSC user interface server or the Eisenkraut audio file editor.

The similar named rsc3, smlsc3 and hsc3 were all developed by Rohan Drape. The Scheme client rsc3 evolved out of a separate sound synthesis engine that Drape had written in C and which was controlled from Scheme. SuperCollider was only used after it became available on the Linux platform, and so rsc3 was born. The switch from interpreted Scheme to compiled Haskell was motivated by the insufficient performance of Scheme to deal with more complex works. Within this move, the library was also significantly simplified and thinned out [12]. Apart from direct control of the SuperCollider server, hsc3 has several extensions such as an audio file library and pattern generators.

The Overtone project primarily targets a live-coding

[1] We define clients such that they transcend a plain general OSC library, by providing representations for the processes on the server, such as UGen graphs, proxies for objects such as Synth, Group, Bus, Buffer etc., or specialised methods to call the server's OSC interface.

Language	Dynamically typed	Statically typed	Object-oriented	Functional	Client	UGen graphs	Musical Scheduling	Interactive mode	Domain Specific GUI	Reference
Clojure	+	–	–	+	Overtone	+	+	+	+	[2]
Haskell	–	+	–	+	hsc3	+	–	+	–	[3]
Java	–	+	+	–	JCollider	+	–	–	–	[4]
Processing	–	+	+	–	p5_sc	–	–	–	* [2]	[5]
Python	+	–	+	+	(pkaudio)	–	+	+	–	[6]
Ruby	+	–	+	+	scruby	+	+	+	–	[7]
Scala	–	+	+	+	ScalaCollider	+	* [3]	+	–	[8]
Scheme	+	–	–	+	rsc3	+	–	+	–	[9]
Standard ML	–	+	–	+	smlsc3	+	–	+	–	[10]
SuperCollider	+	–	+	+	sclang	+	+	+	* [4]	[11]

Figure 1. Comparison of SuperCollider clients. The attributes are just coarse descriptors, and many more could be found (e.g. compiled versus interpreted, availability of step-debugger etc.)

audience. Jeff Rose, one of its main developers, is working on Overtone as part of his PhD thesis, and in the long term aims at a collaborative music system based on peer-to-peer technology. Another layer will sit on top of the direct SuperCollider server control to provide high-level instrument creation and graphical control using a patcher-metaphor. Although judged as unlikely, a future version could also see a pure Java synthesis engine backend instead of scsynth [13].

2. THE SCALA LANGUAGE

ScalaCollider which we will now discuss in detail, is written in the Scala language [14]. Scala combines an object-oriented with a functional approach. The object system is based on multiple inheritance, where a class can inherit at maximum from one direct superclass, but may mix-in any number of so-called traits. Traits are similar to abstract classes in that they can contain methods that are (abstractly) declared, but may also define method bodies. Other than classes their constructor must be parameterless, and they are typically used to add *roles* to a class, or to inject specialised behaviour by overriding an existing method.

Scala's functional capabilities include functions as first-class objects and higher-order functions, anonymous functions and call-by-name parameters (thunks or closures), lazy evaluation, and pattern matching. Scala comes with a large collection library whose application programming interface (API) is predominantly functional and which comes in both mutable and immutable variants.

[2] Since Processing is a language for graphics programming, there are essentially many libraries easily accessible that provide such functionality.

[3] Through the `SoundProcesses` extension package.

[4] While the GUI library is generic, there are extensions and Quarks which provide specific GUI components.

Scala has an extensive type system which is statically checked, and provides good type inference – albeit not as far-reaching as in OCaml or Haskell – in order to eliminates superfluous type specification and to achieve concise code. Through so-called *implicit conversions* and structural-typing, a much more dynamic feeling can be provided than one would expect from a statically-typed language. Implicit conversions allow the compiler to automatically apply methods to arguments in order to convert them to an expected type, and will be explained later in more detail. In the following example, structural-typing is used to define a method `dispose` which accepts an argument of any type which has a method `free` with a unit-type result. It can thus be called equally with an instance of `Node`, `Buffer`, or `Bus`:

```
def dispose(f: {def free: Unit}) {
    println("Disposing " + f)
    f.free
}
val n = Synth.play("default")
dispose(n)
val b = Buffer.read(s, "sounds/a11wlk01.wav")
dispose(b)
```

Mechanisms like implicit conversions, along with the allowance of symbols (e.g. mathematical operators) as identifiers, infix operator syntax, pattern matching, and built-in support for XML literals and parser combinators, make Scala a good candidate for internal domain specific languages (DSL) or language extensions. Since Scala runs on the Java Virtual Machine, a large set of libraries and frameworks from the Java world are instantly accessible, and the compiler and the Scala plug-ins for integrated development environments (IDE) such as Eclipse, NetBeans, and Intellij

```
; Overtone
((synth (let [
  o (mul-add (lf-saw:kr [8 7.23]) 3 80)
  f (mul-add (lf-saw:kr 0.4) 24 o)
  s (* (sin-osc (midicps f)) 0.04)
] (comb-n s 0.2 0.2 4))))

................................................

(let*  ; rsc3
  ((o (mul-add (lf-saw kr (mce2 8 7.23) 0) 3 80))
   (f (mul-add (lf-saw kr 0.4 0) 24 o))
   (s (mul (sin-osc ar (midi-cps f) 0) 0.04)))
  (audition (out 0 (comb-n s 0.2 0.2 4))))

................................................

-- hsc3
let { o = lfSaw kr (mce2 8 7.23) 0 * 3 + 80
    ; f = lfSaw kr 0.4 0 * 24 + o
    ; s = sinOsc ar (midiCPS f) 0 * 0.04 }
  in audition (out 0 (combN s 0.2 0.2 4))

................................................

(* smlsc3 *)
val m = mul_add (lf_saw kr (mce2 (c 8.0) (c 7.23))
  (c 0.0)) (c 3.0) (c 80.0)
val f = mul_add (lf_saw kr (c 0.4) (c 0.0))
  (c 24.0) m
val s = mul (sin_osc ar (midi_cps f) (c 0.0))
  (c 0.04)
val _ = audition (out (c 0.0)
  (comb_n s (c 0.2) (c 0.2) (c 4.0)))

.....................  .....................

# scruby
SynthDef.new :analogbubbles do
  o = LFSaw.kr([8, 7.23], 0, 24, 80)
  f = LFSaw.kr(0.4, 0, 24, o)
  s = SinOsc.ar(f.midicps) * 0.04
  Out.ar(0, CombN.ar(s, 0.2, 0.2, 4))
end send
Synth.new :analogbubbles

................................................

// ScalaCollider
{ val o = LFSaw.kr(List(8, 7.23)).madd(3, 80)
  val f = LFSaw.kr(0.4).madd(24, o)
  val s = SinOsc.ar(f.midicps) * 0.04
  CombN.ar(s, 0.2, 0.2, 4) }.play

................................................

// sclang
{ var o = LFSaw.kr([8, 7.23], 0, 3, 80);
  var f = LFSaw.kr(0.4, 0, 24, o);
  var s = SinOsc.ar(f.midicps) * 0.04;
  CombN.ar(s, 0.2, 0.2, 4) }.play
```

Figure 2. The «Analog Bubbles» example in various clients.

IDEA allow for mixed Scala / Java projects. Scala's syntax also deliberately follows Java's in order to attract programmers with Java background.

3. SYNTH GRAPHS

We call Synth Graph the ensemble of interlinked UGens (the UGen Graph) along with their set of constant input values and parameters which form the interface to external control. The parameters are better known by the term Controls. The vertices of the graph are the UGens, and the edges connect the UGens. If we assume they are directed from a UGen *to its inputs*, we typically find at the roots of the graph the UGens which output sound to the speakers. Only acyclic graphs are permitted, and special UGens are required to achieve feedback.

By associating a Synth Graph with a name, Synth Definitions are formed, and when a Synth is instantiated, the server looks up the corresponding definition through this name. The dissociation of the classes `SynthGraph` and `SynthDef` has a practical reason: It allows to compare the graphs for equality, which can be used for analysis and caching purposes.

3.1 Creation

There are two possible approaches to Synth Graph creation. Either the UGens are created outside any specific context and the roots of the graph are explicitly registered, or the creation is performed as a function evaluation within a special context. The first approach is used for example in hsc3 and JCollider. Although it has the advantage of being "purely functional", its disadvantage is that care must be taken to include all relevant roots in the registration process. For example, if a sound process in monitored by a `DetectSilence` UGen, this UGen forms an additional root. hsc3 provides a function `mrg` to gather the multiple roots, in JCollider we use class `GraphElemArray` which is also handling the multi-channel-expansion.

In the second approach, the new instance of `SynthDef` (for sclang, scruby) or a transient `SynthGraphBuilder` (in the case of ScalaCollider) is marked as the current context during the application of the graph generating function. The UGens instantiated in this function automatically register with the current context, and the resulting graph is build by optimising and sorting the collected UGens.

3.2 Immutable and Marked UGens

ScalaCollider enhances this process in three ways: Firstly, it does away with the extensive entanglement between UGens and Synth Definition found in sclang. It reduces the UGen – which in sclang contains seven mutable fields – to an immutable object by using a dedicated internal intermediate representation during topological sorting. Secondly, all UGen classes are so-called *case-classes*. Not only can UGens thereby

be used in pattern matching, opening interesting applications in graph analysis and transformation, but they are also equipped with an `equals` method based on the UGen's value. The effect is that, as UGens are collected into a Set, duplicate UGens (in terms of value-equality) are automatically eliminated. Consider this element from "harmonic tumbling", one of SuperCollider's standard examples:

```
trig = XLine.kr([10,10], 0.1, 60)
```

The UGen is expanded to two channels, as it is supposed to individuate two `Dust` generators. Since in sclang two instances of `XLine` will never be equal – equality is defined by reference and not by value –, they will be both found in the resulting graph, although one would be sufficient. Of course, a careful user might spot this and use an expression such as `XLine.kr(10, 0.1, 60).dup(2)` instead. While the amount of CPU impact saved through elimination will most likely be only a few percent, it could become more important in the scenario of algorithmically crafted Synth Graphs.

Wouldn't this elimination also affect aforesaid `Dust`? UGens with *indeterminate* behaviour or *side-effects* pose a problem for viewing them as purely functional values. For instance, hsc3 needs special handling of a case such as `WhiteNoise.ar - WhiteNoise.ar` which does not denote silence, but two independent white noise processes being added up. Indeterminacy and side-effects are also specially addressed in ScalaCollider, but in a fully transparent way. Indeterminate UGens are extended by an additional constructor argument, an identifier. Here is the example of `Dust`:

```
case class Dust(rate: Rate, density: UGenIn,
  _indiv: Int) extends SingleOutUGen(density)
```

Along with this class comes the so-called companion object, a singleton of the same name which is used to provide factory methods. Typically the methods represent the calculation rate of the UGen:

```
object Dust extends UGen1ArgsIndiv {
  def ar : GE = ar()
  def ar(density: GE = 1): GE = arExp(density)
  def kr : GE = kr()
  def kr(density: GE = 1): GE = krExp(density)
}
```

Object `Dust` *mixes in* trait `UGen1ArgsIndiv` that provides the methods `arExp` and `krExp`, taking care of multi-channel-expansion and generating the individuating identifier. We also see that a default value for the density argument is provided. [5]

The individuation does not render pattern matching impossible, as we can use a wild card in any argument place. The following method transforms a collection of UGens by replacing any occurence of `Dust.ar` with `Dust2.ar` running at double density:

```
def transf(ugens: GE*) = ugens map {
  case Dust('audio', den, _) => Dust2.ar(den * 2)
  case x => x
}
```

Some UGens with side-effects require individuation, some not. For instance, two equal-valued `DetectSilence` UGens could be collapsed into one instance without alteration of effect (as they would trigger the same done-action at the same moment). On the other hand, two `Out` UGens writing the same inputs to the same bus must be individuated as the signals on the bus sum up. [6]

As a third enhancement, in order to provide a helping indicator for graph analysis, as well as for optimisation, UGens with side-effects are additionally marked with trait `SideEffectUGen`:

```
case class FreeSelf(in: UGenIn)
extends SingleOutUGen(in)
with ControlRated with SideEffectUGen
```

In the current implementation, sub-trees of the graph whose root is not a side-effect UGen, will automatically be discarded, freeing up more CPU usage. Again, this scenario is most likely in machine-generated graphs, but an extension could warn the user that she has maybe overseen something, similar to the warnings emitted by a language compiler or IDE presentation compiler.

4. TYPE CONSIDERATIONS

4.1 Strict Types

The polymorphism facilitated by sclang with its dynamic method dispatch poses a dilemma for a statically-typed language: Either one tries to preserve this polymorphism as much as possible, likely giving up static type check guarantees and ending up with very unspecific types, or one sticks to the strict types and limits expressiveness. Certainly the UGen graph creation, in particular considering the use in rapid prototyping or live coding, calls for a more relaxed handling. ScalaCollider tries to balance the two sides, providing convenient syntax for graph creation, but maintaining as much type safety as possible. It does so mainly through two mechanisms: Pattern matching and implicit conversions.

Let us first consider which elements can be used to construct a UGen graph. Figure 2 depicts one of the standard examples of SuperCollider, the "Analog Bubbles" produced by a sine oscillator, frequency-modulated by a combination of a slower and a faster sawtooth oscillator, and reverberated by a comb-filter. The faster sawtooth is expanded to two channels which oscillate at slightly different speeds, producing a wide stereo image (the expanded graph is shown in figure 3). All implementations produce reasonably compact text with smlsc3 requiring the most characters

[5] The parameter-less methods are added so that one can write `Dust.ar` instead of `Dust.ar()` which Scala prohibits by default as it would cause ambiguity regarding currying.

[6] One could argue of course that such an occurrence would most likely be a mistake by the user.

due to to the way constants are formatted [7]. Syntactically closest to the sclang version are Ruby and Scala.

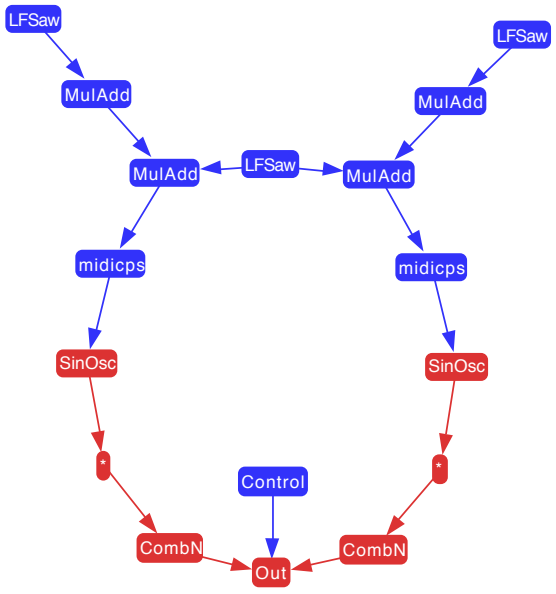

Figure 3. The UGen Graph of "Analog Bubbles". The symmetry is a direct result of the stereo-expansion. Blue colour indicates control-rate, red colour indicates audio-rate calculations. The figure was produced by the ScalaCollider-Swing extension, utilising the Prefuse data visualisation library [15].

"Analog Bubbles" contains three types of UGen inputs: A list or array, constant numbers, and other UGens. Their *strict* types in ScalaCollider all inherit from trait `GE` (graph element). They are `UGenInSeq` and `Constant`, while UGens are represented by three types: A `SingleOutUGen` – the most common type of UGen – has exactly one output channel and can thus be directly plugged into another UGen's input. A `MultiOutUGen` is decomposed into its output channels which are represented by instances of `UGenOutProxy`. Finally, Controls are treated specially as they are mostly created indirectly in the form of `ControlProxyLike` objects which, like multi-channel UGens, are decomposed into their output channels, using type `ControlOutProxy`. Similar to sclang, the actual control UGens, separated by calculation rates and types (regular, lagged, triggered), are internally created at the end of the building stage.

4.2 Implicit types

The compact text of figure 2 is made possible not via massive overloading but a set of implicit conversions:

```
implicit def intToGE(i: Int): Constant
implicit def floatToGE(f: Float): Constant
implicit def doubleToGE(d: Double): Constant
implicit def seqOfIntToGE(x: Seq[Int]): UGenInSeq
implicit def seqOfFloatToGE(x: Seq[Float]): ...
implicit def seqOfDoubleToGE(x: Seq[Double]): ...
implicit def seqOfGEToGE(x: Seq[GE]): UGenInSeq
```

When the compiler sees a statement such as `LFSaw.kr(0.4)` where the `kr` method expects an argument of type `GE`, it looks into the implicit conversions in the current scope. In this case it will find `doubleToGE` which takes a `Double` and returns a `Constant` (which extends `GE` and is thus allowed).

Scala distinguishes between floating point numbers of 32-bit precision (`Float`) and 64-bit precision (`Double`), and hence we have defined different conversions. The case of `LFSaw.kr(List(8, 7.23))` is particularly tricky and shows the limits of Scala's type system: Scala uses implicit numeric widening, so `Int 8` becomes `Double 8.0`, and the argument is of type `List[Double]`. Since `List` is an extension of the general sequence type `Seq`, conversion `seqOfDoubleToGE` applies which produces an instance of `UGenInSeq` (again an extension of `GE`). If we were to make a list of a constant number and another UGen, for example `List(8, DC.kr(7.23))`, the code would not compile, as there is no common supertype of `Int` and `UGen`, resulting in type `List[Any]`. [8] We would be required to specify that the list element type is `GE` so that the conversions would be working again as intended. [9] When translating the standard examples that come with SuperCollider, however, there were no such corner cases.

4.3 Unary and Binary Operators

sclang employs a uniform approach regarding unary and binary operators. The same operations can be performed, no matter if the receiver is a number, a UGen, a collection, or a function: `-3.abs`, `SinOsc.kr.abs`, `[-3.14, 20].abs`, and `{|i| i.log10}.abs` are all valid and behave uniform. As a disadvantage a novice might occasionally get confused when the impedances do not match in a particular case, such as an `if` statement whose semantics shift when applied to UGens.

We have tentatively implemented part of this "homomorphous" interface in ScalaCollider via three additional implicit conversions:

```
implicit def intWrapper(i: Int): RichInt
implicit def floatWrapper(f: Float): RichFloat
implicit def doubleWrapper(d: Double): RichDouble
```

Operators such as `cpsmidi` are now applicable to both numbers and graph elements. Types are preserved, so `440.cpsmidi` yields a number and `Pitch.kr(x).cpsmidi` a graph element (`GE`). Since the number classes are *final* in Scala and since it is also not possible to add methods to existing classes, another approach is necessary to make this possible. Martin Odersky, the author of the Scala language, has coined the approach «Pimp my Library» [16]: An implicit conversion is defined that lifts the inaccessible type

[7] JCollider is actually much more verbose and has been omitted from the figure due to its size.

[8] `Any` is at the root of the Scala class hierarchy.

[9] Alternatively, a method such as hsc3's `mce` could be added.

(e.g. `Float`) to a wrapper class (e.g. `RichFloat`) which provides the additional methods. For this to work transparently, the methods of the enriched class always return the primitive type (`Float`), making the intermediate representation disappear. For example:

```
class RichFloat(f: Float) {
   def dbamp = (math.pow(10, f * 0.05)).toFloat
   ...
```

All in all, as with all forms of "magic", great care has to be taken in deciding on the implicit conversions, as they also open up for unexpected or difficult to trace behaviour. With the above conversions, `List(330, 440).cpsmidi` results in a `UGenInSeq`, but `List(330, 440).map(_.cpsmidi)` in a `List[Float]`. Furthermore, we must establish a hierarchy of conversions such that upon encountering `440.cpsmidi`, the compiler does not give up with an ambiguity error as there are two possible implicit conversions in scope. Clearly, we would like `intWrapper` to take precedence over `intToGE`, and we can accomplish that by having `intToGE` appear in a supertype of the object that defines `intWrapper`.

There are a few more conversions available, for example `"gain".kr` produces a control-rate control proxy named `"gain"`, and in the case of `{...}.play` the closure is converted into a Synth Graph and spawns a Synth playing this graph. The relationship between the different traits and conversions mentioned so far is illustrated in figure 4.

4.4 Pattern Matching

Inevitably, different types get aliased within method signatures, since overloading becomes impossible as the permutations of allowed types grow exponentially. One mechanism to retain fine grained control is pattern matching. The following example is taken from the `MulAdd` UGen, where method `make1` is called after multi-channel expansion to instantiate each UGen:

```
private def make1(rate: Rate, in: UGenIn, mul:
                  UGenIn, add: UGenIn) : GE =
    (mul, add) match {
      case (c(0),  _)    => add
      case (c(1),  c(0)) => in
      case (c(1),  _)    => in + add
      case (c(-1), c(0)) => -in
      case (_,     c(0)) => in * mul
      case (c(-1), _)    => add - in
      case _ => this(rate, in, mul, add)
   }
```

Identifier `c` here is short for `Constant` and the underscore character represents the wildcard, so the pattern matching is used to optimise performance by returning UGens simpler than `MulAdd` if either of the arguments is a specific constant number. Similar optimisations occur in the unary and binary operator UGens.

The compiler will automatically detect matching errors as would arise from cases which are unreachable, and if pattern matching is applied to *sealed* types, it will also emit warnings if the match is not exhaustive.

5. CONCURRENCY AND "STATE"

As the current trend goes towards multi-core processors, concurrency has become the buzz word in computer programming languages. Thanks to the server-client model of SuperCollider, it is already possible to run several instances of the server, however there is no communication between them such as shared buffers or buses. The supernova project [17] tries to overcome this by parallelising the server architecture directly. Still, on the client side, sclang employs cooperative multithreading. While this makes reasoning simple, as all the routines behave deterministically, it also neglects the presence of multiple processors, and forbids certain tasks, such as rendering chunks of audio data in the background on the client side.

5.1 Actor-based Concurrency

Advocates of functional programming usually blame the presence of "state" for problems in concurrency. State though is an inept term, as the problem is rather mutability (the changes inbetween-states). This semantic flaw reveals the fundamental assumption of a functional program: being timeless. This of course is contrary to our everyday experience which takes place in time, and dynamicity is at the core of sound and music. So there is a mismatch between "purely functional" and sound process.

Mutability and immutability are not opponents, but can be combined in a careful balance. The problem is often one of stable identifiers that denote a dynamic object. For instance, on the server, a Synth object is identified through its node-ID. If one wishes to change a parameter of this node, we send a message to the server, specifying the node-ID, the parameter name (another identifier) and the new value, for example `("/n_set", 1000, "freq", 880)`. To query a parameter, we send a query message to the server, say `("/s_get", 1000, "freq")`, and wait for a reply message. This reflects exactly the Actor-model of concurrency [18], where each side maintains its own local knowledge of the system's state, and knowledge is exchanged by sending messages which are handled through a mailbox queue.

Scala's standard libraries include a dedicated actors package [19] which is used in the `Server` class of ScalaCollider. We follow the naming conventions of Scala-Actors: The `!` method (aka "bang") dispatches an OSC message to the server and returns immediately. The double-bang method `!!` also returns immediately, but with an instance of `Future`. Futures are references to asynchronous return values. They define a method `isSet` which can be called to find out whether the result has already been received or not. To dereference the future, method `apply` is used.

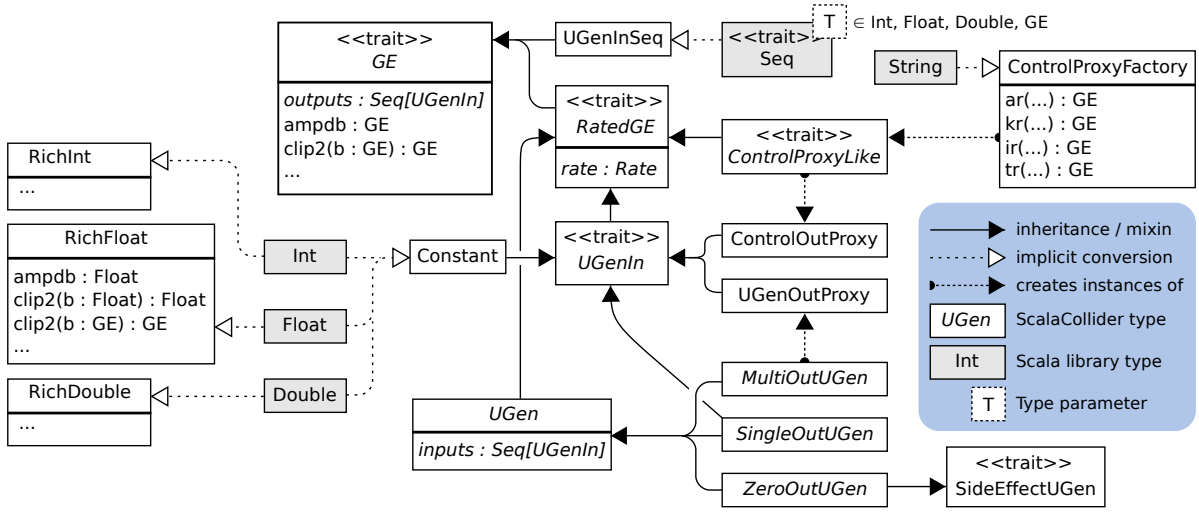

Figure 4. Traits and classes involved in the creation of UGen graphs.

However, futures are more powerful: They are implemented as actors themselves, and it is thus possible to wait for the result from within an actor. For a single asynchronous message, `Server` provides a convenient method `!?` which sends out the message and registers a `PartialFunction` (a pattern matcher) with the internal OSC receiving actor. Incoming reply messages from the server are tested against this function, and in the case of a match, the case-body is executed and the handler is discarded. If no matching message arrives within a given timeout period, the special message `TIMEOUT` is sent. Here is an example:

```
val c = Bus.control(s)
val x = {Out.kr(c.index,
   Pitch.kr(In.ar(NumOutputBuses.ir)))}.play
s !? (1000, c.getMsg, {
   case OSCMessage("/c_set", c.index, freq) =>
      println("Tracked frequency: " + freq)
})
```

Of course, this is usually hidden under convenience methods such as `ControlBus→get`. It becomes more interesting when needing to wait for several asynchronous messages to be completed. This will be addressed by the extension package SoundProcesses which is outlined in the next section.

5.2 Node Proxies and Sound Processes

The actors model is not the only possible way to deal with concurrency, and in particular the operation of reading a value, being an asynchronous query and reply, can be a performance penalty. We have thus refrained from using it more extensively at the core of ScalaCollider. Instead, the concept is to provide a base layer, on top of which more high-level systems can be constructed. The base layer reduces the amount of variables found in their sclang counterparts (we have already seen this with the `UGen` class). For instance, the `Node` class in sclang contains five mutable fields (`nodeID`, `server`, `group`, `isPlaying`, and `isRunning`) which are even publicly writable. `Synth` adds another one (`defName`).

While this seems harmless from a small-scale project or live-coding point of view, it is problematic in the design of a base library for scalable systems. We are currently developing a higher-level layer called SoundProcesses which will be used to describe sound processes.

This extension will provide automatic management of resources such as buffers and buses, of interconnectivity between processes such as graph order and control mapping, and of temporal behaviour such as logical time scheduling and atomic *transactions*. If several processes are engaged in an action, they might need to be synchronised through OSC bundles. Within such a transaction, an intermediate representation of the engaged nodes, buffers and buses must be maintained – for example, which nodes as a result of the transaction will be moved, freed or paused, etc. –, and in the case of one partner in the transaction aborting, this intermediate representation must be discarded. If a bundle is sent out or is successfully completed (by replying for instance with a `/synced` message), the intermediate representation becomes the current state of the system. Mutable fields in `Node`, `Buffer` and so on are thus inadequate representations, and they have been mostly dropped. [10]

We are currently working with software-transactional-memory (STM) which provides mechanisms to capture related modifications to data structures inside atomic transactions, with the possibility to roll them back if an error occurs or two transactions produce a concurrent conflict. There are two interesting libraries available: Akka [20] is a large framework that not only provides a combination of its own actors and STM (combining them into a construct called *transac-*

[10] As a concession to convenient rapid proto-typing, a few are left, such as the definition name of a Synth, or Buffer information such as number of channels and frames, but they will only be updated internally in a few defined cases.

tors), but also manages persistency to databases and allows actors to run distributed across several computers. CCSTM [21] focuses on STM, but is more compact, does not require bytecode weaving, and might play nicely along with other libraries such as Scala Actors. Our experiments with CCSTM have shown that it is extremely useful for creating robust systems, since unforeseen runtime errors cause safe rollbacks, preventing the system from being left in an inconsistent state (for instance regarding how sound processes are interconnected).

6. CONCLUSION

We have introduced ScalaCollider, a new client for the SuperCollider server. With ScalaCollider, several functions which were fragmented in sclang, are reunited in one language: The class library, the backbone primitives, the interpreter, the client side audiofile library, and GUI frontends – while sclang restricts the language in interpreter mode (it is not possible to define further classes at runtime), requires to write performance critical code as C primitives and complex custom graphical user interface elements in Java or Objective-C.

The next big effort lies in the design of the SoundProcesses extension which has received a preliminary outline, and which will provide a high-level layer for defining sound processes. While the Java Virtual Machine is quite powerful and the systems built on top of it have received "free" upgrades, such as the evolution of the HotSpot garbage collector in Java 5 and Java 6 (as well as the prospect of even better performance with the integration of the JRockit garbage collector in Java 7), warm-up times are currently severely high. Mechanisms must be developed to control the warm-up of the class-loader in order to achieve the best possible latencies.

The UGen classes are still handcrafted and the directory is not fully complete. Effort is currently invested in synthetic generation of these classes from a UGen descriptor database, an approach already taken by JCollider, scruby, rsc3, hsc3, and Overtone. In this process, we will try to enrich the types to create even stronger compile-time restrictions, such as the required calculation-rates of UGen input arguments. We also think that the possibility of algorithmic transformations of UGen graphs could become more valuable if the multi-channel-expansion was performed at a later stage in the building process.

Finally, an integration with two systems offers interesting possibilities: Android and Processing. The port of the SuperCollider server to the Android mobile platform is currently underway, and Scala itself runs nicely on Android. The Processing graphics programming framework has recently been ported to Scala [22], so bringing sound and visuals together will become even easier. We have conducted small proofs-of-concept that show that ScalaCollider in fact runs on Android and combines well with Processing.

7. REFERENCES

[1] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal*, vol. 26, pp. 61–68, Winter 2002.

[2] Overtone. http://rosejn.github.com/overtone/.

[3] hsc3. http://www.slavepianos.org/rd/sw/hsc3/.

[4] JCollider. http://www.sciss.de/jcollider/.

[5] p5_sc. http://www.erase.net/projects/p5_sc/.

[6] Python and SuperCollider. http://trac2.assembla.com/pkaudio/wiki/SuperCollider.

[7] scruby. http://github.com/maca/scruby.

[8] ScalaCollider. http://www.sciss.de/scalaCollider/.

[9] rsc3. http://www.slavepianos.org/rd/sw/rsc3/.

[10] smlsc3. http://www.slavepianos.org/rd/sw/smlsc3/.

[11] SuperCollider. http://supercollider.sourceforge.net/.

[12] R. Drape. Personal communication, May 2010.

[13] J. Rose. Personal communication, May 2010.

[14] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger, "An Overview of the Scala Programming Language," *Technical Report LAMP-REPORT-2006-001*, 2006.

[15] J. Heer, S. Card, and J. Landay, "Prefuse: a toolkit for interactive information visualization," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 421–430, 2005.

[16] M. Odersky, "Pimp my library," *Artima Developer Blog*, October 9, 2006. http://www.artima.com/weblogs/viewpost.jsp?thread=179766.

[17] T. Blechmann, "supernova, a multiprocessor-aware synthesis server for SuperCollider," *Proceedings of the 8th Linux Audio Conference*, pp. 32:1–32:5, 2010.

[18] G. Agha, "Actors: a model of concurrent computation in distributed systems," *MIT Artificial Intelligence Laboratory Technical Report 844*, 1985.

[19] P. Haller and M. Odersky, "Event-based programming without inversion of control," *Proceedings of the Joint Modular Languages Conference*, pp. 4–22, September 2006.

[20] Akka. http://akkasource.org/.

[21] N. Bronson, H. Chafi, and K. Olukotun, "CCSTM: A library-based STM for Scala," in *Proceedings of the First Scala Workshop*, 2010.

[22] Spde. http://technically.us/spde/.