James McCartney
850 Stendhal Ln.
Cupertino, California 95014 USA
james@audiosynth.com

Rethinking the Computer Music Language: SuperCollider

Designing a new computer music language requires one to answer certain questions. Some of these questions may at first glance seem trivial but on further examination are rather deep. The following questions fit this category. What is a computer language? What is the difference between a high-level and a low-level language? What do current computer music languages do for you? What should a computer music language do? How can computer language abstractions be applied to computer music? Is a specialized computer music language even necessary? This article discusses the above questions, how they led to creating the SuperCollider language, and some current development and further directions for SuperCollider.

What Is a Computer Music Language?

A computer language presents an abstract model of computation that allows one to write a program without worrying about details that are not relevant to the problem domain of the program. The greater the power of abstraction of a language, the more the programmer can focus only on the problem and less on satisfying the constraints and limitations of the language's abstractions and the computer's hardware. Some of the abstractions available (roughly in order from less abstract to more abstract) are listed in Table 1.

The abstractions provided by the Music N languages, including Csound (www.csounds.com), are the abstraction of a unit generator, the audio sample computation loop, the representation of the connections between unit generators, and instrument instantiation and de-allocation. These abstractions make writing signal-processing algorithms easier, because they abstract a number of cumbersome details. However, the Music N family provides few control structures, no real data structures, and no user functions. SAOL (www.saol.net) improves the Music N paradigm by providing the kinds of abstractions found in the C language, such as control structures, functions, and some data structures. Max (www.cycling74.com/products/maxmsp.html), which is quite a different kind of programming language, provides an interesting set of abstractions that enable many people to use it without realizing they are programming at all. Max provides some object-oriented features, including dynamically typed data, dynamic binding, but no inheritance, limited data types, but only one data structure for inter-object messaging and few control structures. The Max language is also limited in its ability to treat its own objects as data, which makes for a static object structure. Later evolutions of Max, such as jMax (www.ircam.fr/produits/logiciels/log-forum/jmax-e.html) and Pd (www.pure-data.org), do various things to expand the data structure limitations of Max but still have a generally static object structure.

A computer music language should provide a set of abstractions that makes expressing compositional and signal processing ideas as easy and direct as possible. The kinds of ideas one wishes to express, however, can be quite different and lead to very different tools. If one is interested in realizing a score that represents a piece of music as a fixed artifact, then a traditional orchestra/score model will suffice. Motivations for the design of SuperCollider were the ability to realize sound processes that were different every time they are played, to write pieces in a way that describes a range of possibilities rather than a fixed entity, and to facilitate live improvisation by a composer/performer.

Computer Music Journal, 26:4, pp. 61–68, Winter 2002
© 2002 Massachusetts Institute of Technology.

Table 1. **Some abstractions available in modern computer programming languages**

Abstraction	*Description and Purpose*
Variable names	Provide human readable names to data addresses
Function names	Provide human readable names to function addresses
Control structures	Eliminate ''spaghetti'' code (The ''goto'' statement is no longer necessary.)
Argument passing	Default argument values, keyword specification of arguments, variable length argument lists, etc.
Data structures	Allow conceptual organization of data
Data typing	Binds the type of the data to the type of the variable
Static	Insures program correctness, sacrificing generality.
Dynamic	Greater generality, sacrificing guaranteed correctness.
Inheritance	Allows creation of families of related types and easy re-use of common functionality
Message dispatch	Providing one name to multiple implementations of the same concept
Single dispatch	Dispatching to a function based on the run-time type of one argument
Multiple dispatch	Dispatching to a function based on the run-time type of multiple arguments.
Predicate dispatch	Dispatching to a function based on run-time state of arguments
Garbage collection	Automated memory management
Closures	Allow creation, combination, and use of functions as first-class values
Lexical binding	Provides access to values in the defining context
Dynamic binding	Provides access to values in the calling context (`.valueEnvir` in SC)
Co-routines	Synchronous cooperating processes
Threads	Asynchronous processes
Lazy evaluation	Allows the order of operations not to be specified. Infinitely long processes and infinitely large data structures can be specified and used as needed.

Applying Language Abstractions to Computer Music

The SuperCollider language provides many of the abstractions listed above. SuperCollider is a dynamically typed, single-inheritance, single-argument dispatch, garbage-collected, object-oriented language similar to Smalltalk (www.smalltalk.org). In SuperCollider, everything is an object, including basic types like letters and numbers. Objects in SuperCollider are organized into classes. The UGen class provides the abstraction of a unit generator, and the Synth class represents a group of UGens operating as a group to generate output. An instrument is constructed functionally. That is, when one writes a sound-processing function, one is actually writing a function that creates and connects unit generators. This is different from a procedural or static object specification of a network of unit generators. Instrument functions in SuperCollider can generate the network of unit generators using the full algorithmic capability of the language.

For example, the following code can easily generate multiple versions of a patch by changing the values of the variables that specify the dimensions (number of exciters, number of comb delays, number of allpass delays). In a procedural language like Csound or a ''wire-up'' environment like Max, a different patch would have to be created for different values for the dimensions of the patch.

```
(
{
var exciterFunction, numberOfExciters, numberOfCombs, numberOfAllpass;
var in, predelayed, out;
  // changing these parameters changes the dimensions of the patch.
numberOfExciters = 10;
numberOfCombs = 7;
numberOfAllpass = 4;
  // a function to generate the input to the reverb,
  // a pinging sound.
exciterFunction = { Resonz.ar(Dust.ar(0.2, 50), 200 + rand(3000.0), 0.003) };
  // make a mix of exciters
  // Mix.arFill fills an array with the results of
  // a function and mixes their output.
in = Mix.arFill(numberOfExciters, exciterFunction);
  // reverb predelay time :
predelayed = DelayN.ar(in, 0.048);
  // a mix of several modulated comb delays in parallel:
out = Mix.arFill(numberOfCombs, {
   CombL.ar(predelayed, 0.1, LFNoise1.kr(rand(0.1), 0.04, 0.05), 15)
  });
  // a parallel stereo chain of allpass delays.
  // in each iteration of the do loop the Allpass input is the
  // result of the previous iteration.
numberOfAllpass.do({
  out = AllpassN.ar(out, 0.050, [rand(0.050), rand(0.050)], 1)
});
  // add original sound to reverb and play it:
in + (0.2 * out);
}.play
)
```

One way of conceiving of a composition is as a sequence of events. SuperCollider supports this abstraction via the concept of a stream. A stream is an object to which the `next` message can be sent to get the next element. A stream ends when it returns `nil` in response to `next`. A finite stream is one that eventually returns `nil`; an infinite stream is one that never does. By default, all objects in SuperCollider respond to `next` by returning themselves, so any object can be used as an infinite stream of itself. There are also Stream classes that define operations on streams and Pattern classes that can create multiple streams from a single specification.

```
// a Pattern specifying a stream that
// returns 1, 2, 3, 1, 2, 3, nil
p = Pseq([1, 2, 3], 2);
s = p.asStream;
// create the stream
// from the Pattern s.next;
// get the next value 1 s.next;
// get the next value 2 s.next;
```

```
// get the next value 3 s.next;
// get the next value 1 s.next;
// get the next value 2 s.next;
// get the next value 3 s.next;
// get the next value nil
```

SuperCollider defines an event stream as one that responds to `next` by returning dictionaries that map symbols to values. The synthesis instrument looks up the parameters it needs to start an event from those in the dictionary. Thus, the composition code does not need to know anything about an instrument's argument list order. Using dictionaries also means an event may contain any set of parameters.

```
p = Pbind(\midinote, Pseq([60, 62, 63],
2), \dur, Pseq([0.5, 0.25], 3));
s = p.asEventStream(Event.new);
s.next;
Event[ (dur -> 0.5), (midinote -> 60) ]
s.next;
Event[ (dur -> 0.25), (midinote -> 62) ]
s.next;
Event[ (dur -> 0.5), (midinote -> 63) ]
s.next;
Event[ (dur -> 0.25), (midinote -> 60) ]
s.next;
Event[ (dur -> 0.5), (midinote -> 62) ]
s.next;
Event[ (dur -> 0.25), (midinote -> 63) ]
s.next;
nil
```

SuperCollider Server

SuperCollider was originally designed to be a close-as-possible marriage between a high-level language and a synthesis engine. SuperCollider version 2 (SC2) represents this approach. However, although this close coupling has some advantages, it is not the only approach, and there are a number of reasons to separate a composition language from a synthesis engine. In the closely coupled architecture, some synthesis processing time must be consumed generating events. If the composition language were removed from the synthesis engine, it could run in the background generating events ahead of time. SuperCollider Server is an architecture that goes in this direction.

In SuperCollider Server, the synthesis engine and the SuperCollider language engine are separate applications. They communicate via a slightly modified version of the Open Sound Control (OSC) protocol (Wright 1998). This allows users to run multiple copies of the synthesis engine either on the same machine (to exploit multiple processors) or on machines distributed across a network. Controlling the synthesis engine is as simple as opening a socket and sending commands, so any program (e.g., Max, a Java applet, or a C/C++ program) could control it, not just a program written in the SuperCollider language.

Both the synthesis and language engines are Macintosh OS X command-line applications, with a full graphical user interface (GUI) version of the language engine also available.

Features of the Synthesis Engine

The SuperCollider 3 Synth Sever is a simple but flexible synthesis engine. While synthesis is running, new modules can be created, destroyed, and re-patched, and sample buffers can be created and reallocated. Effects processes can be created and patched into a signal flow dynamically at scheduled times. Patching between modules is done through global audio and control busses.

All commands are received via TCP or UDP using a simplified version of Open Sound Control. The Synth Server and its client(s) may be on the same machine or across a network. The Synth Server does not send or receive MIDI; it is expected that the client will send all control commands. If MIDI is desired, it is up to the client to receive it and convert it to appropriate OSC commands for the synthesis engine.

Synthesis definitions are stored in files generated by the SuperCollider language application. Unit generator definitions are Mach-O bundles (not to be confused with CFBundles). The unit generator applications programming interface (API) is a simple C interface.

I have written two versions of the SC Server synthesis engine. One uses a block computation model and unit generator plug-ins. Instruments are loaded as files that describe patches of these unit generators. Another version is implemented as a single-sample computation model with the instruments loaded as compiled plug-ins.

Tree of Nodes

All running modules are ordered in a tree of nodes that define an order of execution. A Node is an addressable node in a tree of nodes run by the synthesis engine. There are two types of Nodes: Synths and Groups. The tree defines the order of execution of all Synths, and all nodes have an integer identifier (ID). A Group is a collection of Nodes represented as a linked list. A new Node may be added to the head or tail of the group. The Nodes within a Group may be controlled together. The Nodes in a Group may be both Synths and other Groups. At startup, there is a top-level group with an ID of zero that defines the root of the tree. A Synth is a collection of unit generators that run together. They can be addressed and controlled by commands to the synthesis engine. They read input and write output to global audio and control busses. Synths can have their own local controls that are set via commands to the server.

Audio and Control Busses

Synths send audio and control signals to each other via a pair of global arrays of audio and control busses. Busses are indexed by integers beginning with zero. Using busses rather than connecting Synths to each other directly allows Synths to connect themselves to the community of other Synths without a priori knowledge about them. The lowest-numbered audio busses get written to the audio hardware outputs. Immediately following the audio output busses are the audio input buses, which are read from the audio hardware inputs. The number of bus channels defined as inputs and outputs do not have to match that of the hardware.

Figure 1 illustrates how the tree of nodes and the buses work together to create a synthesis architecture. Circles represent nodes, and rectangles represent busses. This tree of nodes is organized in a way that ensures that execution proceeds in order starting with control nodes, followed by instruments, effects, and finally a mixer module.

Features of the Block Computation Version

This version includes the control rate and an audio rate dichotomy like SC2, Csound, and other languages. As in SC2, all unit generators are implemented so that they always linearly interpolate a control input whenever not doing so would result in a discontinuity. The language's class library can generate instrument definition files to be loaded by the synthesis engine.

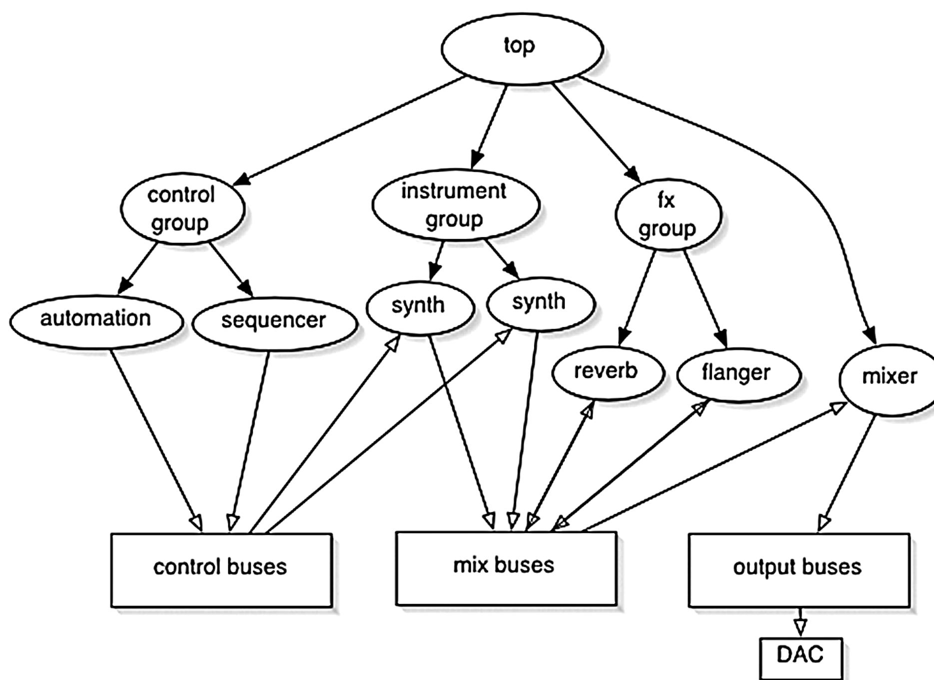
Features of the Single-Sample Computation Version

The language's synthesis class library can generate C++ code to be loaded by the synthesis engine. It is not necessary to use the SuperCollider language to generate synthesis code. The synthesis engine has a simple C linkage API that allows anyone to write loadable instruments. The code generator implemented in the class library has several interesting features, as described below.

Instead of a single control rate, any unit generator may run at any power of two division of the audio clock rate and is by default automatically linearly interpolated to the rate of any unit generator to which it is an input. Only source unit generators need to specify the computation rate. Modifier unit generators determine their rates from their inputs. Unit generators can be easily implemented by overriding a few methods that generate code, illustrated by the following example of a sawtooth oscillator:

```
LFSaw : UGen {
 *new { arg cdiv, freq = 440.0, iphase
= 0.0, mul=1.0, add=0.0;
  ^this.multiNew(cdiv, freq * (2.0 *
SampleDur(cdiv)), iphase) * mul + add
```

Figure 1. The tree of nodes (circles) executes from left to right, operating on the busses (rectangles).




```
 }
 decl {
  ^''
   FLT @phase;
  ``
 }
 start {
  ^''
   @phase = $iphase;
  `` ++ this.calc
 }
 calc {
   ''
   $out = @phase;
   FLT nextphase = @phase + $freq;
   if (nextphase >= 1.f) nextphase -=
2.f;
   else if (nextphase <=±1.f)
nextphase += 2.f;
   @phase = nextphase;
  ``
 }
}
```

New Features in SC3

Multiple synthesis engines can be supported either on the same machine or distributed across a network, simply by running multiple copies of the synthesis engine. This is a simple way to exploit multiple processors with the additional advantage of having each process protected from a crash by the other. Because the language engine and the synthesis engine are separate applications, they are also protected from each other's crashing. This makes live performances safer by providing more opportunity to recover gracefully from a crash in the middle of a performance.

Spawning new events is much faster with SuperCollider Server. Because instruments are compiled to a single object, the data for all unit generators can be allocated with a single call to the allocator. In the single sample version, linking unit generators together requires no extra work, since they have been compiled into code. Spawned Synths can also insert themselves into a graph via global busses.

Features Lost from Version 2

Some features of version 2 are lost with this decoupled architecture. Because instruments are compiled into code, it is not possible to generate patches programmatically at the time of the event as one could in SC2. The advantage is that it is much faster to allocate a new instrument, and CPU usage levels are more constant, with fewer "bursts." Users can still design instruments programmatically in the same way as in version 2; however, instruments now get compiled.

It is not possible in SC Server to run composition-level code synchronously in response to an audio trigger. (Asynchronous triggering is possible, however.) In SC2, an audio trigger could suspend the signal processing, run some composition code, and then resume signal processing. In SC Server, messaging between the engines causes a certain amount of latency.

Because unit generators get compiled into instruments, they do not exist as objects at synthesis time. Thus, polling instantaneous values of unit generators must be handled differently.

Using the Server from the Language

The following example shows how to control the server directly from the language. Some class libraries like the Pattern classes will be able to encapsulate management of some of the low-level interaction with the server illustrated below, making it easier to use.

In the example, the server is started, two groups are created, an instrument definition is created and loaded, and then 200 randomly generated events are played.

```
s = Server.default;
// get the default
// server address s.boot;
// start synth server program s.start;
// start audio running
// we need two groups: one each for
// instruments and effects.
Server.default.sendMsg(''/g_new'', 1,
0);
// create group 1 (for instruments)
Server.default.sendMsg(''/g_new'', 2,
0);
// create group 2 (for effects)
// define an instrument
SynthDef(''pulse'', { arg ioutchan = 0,
freq = 400, gate = 1.0;
 var env, out;
 // envelope generator
 env =
EnvGen.kr(Env.linen(0.01,0.1,0.7,-2),
gate);
 FreeSelfWhenDone.kr(env);
// automatically deallocates synth when
// done
// 2 channel pair of pulse waves into a
// pair of filters.
 out =
RLPF.ar(Pulse.ar(freq+[0,1],0.2,env),
freq*LinRand(2,16), 0.2);
 // mix to the output bus
 Out.ar(ioutchan, out)
}).writeDefFile;
// load the instrument
Server.default.sendMsg(''/d_load'',
''engine/synthdefs/pulse.scsyndef'')
// scheduling latency (depends on
// hardware and how far ahead you want
// to generate events)
lat = 0.05;
// define a routine to generate events.
r = Routine({
 200.do({ arg i;
  var dur, freq;
  // random duration
  dur = [0.15, 0.2, 0.3].choose;
  // random frequency
  freq = rrand(36, 96).midicps;
  // send OSC message to start the event
  // at time lat from thread's current
  // time
  s.sendBundle(lat, [''/s_new'',
  ''pulse'', 1000+i, 1, 'freq', freq]);
```

```
  // send OSC message to release the
  // gate of the envelope at time
  lat+dur
  s.sendBundle(lat+dur, [‘‘/n_set’’,
  1000+i, ‘gate’, 0]);
  // sleep the thread for the duration
  // of the event
  dur.wait;
 });
});
// play the routine
SystemClock.play(r)
```

SuperCollider on MacOS X

SuperCollider has been, mostly for historical reasons, a MacOS application from the beginning. Now that Apple is moving to OS X, work has been ongoing to port and redesign SuperCollider to take advantage of the benefits of this new operating system. The SuperCollider Server language application has been rewritten using the Cocoa framework in Objective-C.

The SuperCollider graphical user interface classes now support ‘‘drag and drop’’ of any SuperCollider object. This will, I think, lead to an easy way for non-programmers to use SuperCollider by using the ‘‘drag and drop’’ paradigm to make connections between pre-built modules. All objects can have inspectors opened for them, and their internal structure and the structure of the entire system can be traced by clicking along, following links.

Another version of SuperCollider, called version 3, had a newly rewritten unit generator class library and a dual language engine architecture, with one engine running the graphical user interface and another running the synthesis code. This architecture presented a number of difficult problems involving sharing data between two garbage-collected virtual machines running in different threads.

Conclusion

Is a specialized computer music language even necessary? In theory at least, I think not. The set of abstractions available in computer languages today are sufficient to build frameworks for conveniently expressing computer music. Unfortunately, in practice, some pieces are missing in the implementations of languages available today. Often, the garbage collection is not performed in real time, and often argument passing is not very flexible. If lazy evaluation is not included, then implementing Patterns and Streams becomes more complicated. I wrote SuperCollider to have the set of abstractions that I wanted for computer music to have something flexible to use. In the future, other languages may be more appropriate.

One goal of separating the synthesis engine and the language in SC Server is to make it possible to explore implementing in other languages the concepts expressed in the SuperCollider language and class library. Some other languages that I think may have interesting potential in the future for computer music are OCaml (www.ocaml.org), Dylan (www.gwydiondylan.org), GOO (www.googoogaga.org), and also possibly Ruby (www.ruby-lang.org), a scripting language which coincidentally shares many syntactic and semantic attributes with the SuperCollider language.

References

Wright, M. 1998. ‘‘Implementation and Performance Issues with Open Sound Control.’’ *Proceedings of the 1998 International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 224–227.