Java Music Specification Language, an introduction and overview

Nick Didkovsky, Philip L. Burk

email: didkovn@mail.rockefeller.edu, philburk@softsynth.com
www.algomusic.com

Abstract

Java Music Specification Language (JMSL) is a new Java-based development tool for experiments in algorithmic composition, live performance, and intelligent instrument design. JMSL is the evolutionary successor to the Hierarchical Music Specification Language (Polansky, Rosenboom, and Burk, 1987). While HMSL was Forth-based, JMSL is written in Java.

JMSL's features include:

- *Stylistically neutral core*
- *Polymorphic hierarchical scheduling*
- *Device abstraction. JMSL supports Robert Marsanyi's JavaMIDI, Softsynth's JSyn, MidiShare (Orlarey and Lequay 1989), and Sun's JavaSound at a level that hides their implementation.*
- *An algorithmically extensible common music notation editor called JScore which features an algorithmic notation and transformation plugin API.*
- *Its Java core. As opposed to a closed system with a proprietary language, JMSL allows the programmer to leverage off the vast resources available to Java developers, including Java's database connectivity, networking tools, 2D and 3D graphics packages, servlet API, and numerous third party packages.*
- *The composer can create stand-alone JMSL applications or deploy JMSL applets on the web.*
- *JMSL offers a freely downloadable "Lite" version.*
- *Runs on Windows, MacIntosh, and Linux platforms*

1 Introduction

The goal of the Java Music Specification Language (JMSL) is to provide an algorithmic music composition and performance API that is flexible, stylistically neutral, and portable. To this end, the Java programming language serves us well. Java is a language that has much to offer computer music composers including good object oriented support, extensive auxiliary APIs for networking, graphics etc., and the ability to run on multiple platforms including web browsers.

JMSL extends Java with classes for hierarchical scheduling of composition objects, sequence generators, distribution functions and other music related tools. JMSL also features a non-core package called JScore, which is a programmable music notation editor. JScore supports an API for adding notes to a score as well as an API to transform notated musical material.

JMSL's stylistically neutral core, its flexible framework for hierarchical scheduling and instrument design, and its ability to notate and transform algorithmically generated music offers rich new territory for composers to explore.

2 History

The Java Music Specification Language was motivated by the need for an evolutionary successor to the Hierarchical Music Specification Language (HMSL). HMSL was designed and programmed at the Mills College Center for Contemporary Music by Phil Burk, Larry Polansky, and David Rosenboom. Some key ideas found in HMSL began to be ported to the Java programming language by Nick Didkovsky in 1997.

After the premiere of an interactive piece that served as a JMSL proof of concept (Didkovsky 1997), JMSL was both simplified and (r)evolutionized by Phil Burk and Didkovsky, and taken well beyond a straightforward port of HMSL.

Didkovsky has taught computer music using JMSL in his Java Music Systems course at NYU since 1999. JMSL was officially released in July, 2001, at algomusic.com.

3 The Composable Interface

The notion of hierarchies is a key one in JMSL. A hierarchy is a network of parent/child relationships. A song form provides an example, where a parent called mySong might have four children: verse1, chorus, verse2, chorus. In JMSL, mySong would be a SequentialCollection.

```
            mySong
     __________|___________
     |       |      |       |
   verse1  chorus verse2 chorus
```

Any Java class that implements JMSL's Composable interface can be put in a JMSL hierarchy. Composables implement methods for launching themselves on a schedule, repeating, reporting their finish time back to a parent, and applying a "timeStretch" factor to their duration.

With this scheme, JMSL can schedule arbitrary events over time (not just musical ones). For example, MusicJob is a JMSL class that implements the Composable interface. Its user-defined action, which is repeated over time, might draw graphics, print text, send a midi note, harvest performance statistics, set a value on a JSyn circuit's port, etc.

The user can define what a MusicJob does by overriding its repeat() method, as shown below. This MusicJob prints a message and changes its own scheduling.

```
public double repeat(double playTime) {
  JMSL.out.println("My playTime is " + playTime);
  return playTime + JMSLRandom.choose(1.0, 5.0);
}
```

4 Hierarchies of Composables

JMSL hierarchies are created by adding Composables to JMSL's various collections: SequentialCollection, ParallelCollection, and QueueCollection. Being Composables themselves, collections can be nested.

A Sequential Collection launches its children in sequence, waiting for each to finish before the next is launched. It can optionally have a user-defined Behavior assigned to it, which it consults to select the next child every time it repeats, liberating it from a strictly indexed order. JMSL's ParallelCollection launches all its children at the same time, waiting for all to report their finish time before continuing. A QueueCollection removes a child after playing it. The excerpt below builds a Sequential Collection, adds two MusicJobs to it, and launches it.

```
SequentialCollection seqCol = new SequentialCollection();
seqCol.add(new MusicJobExample());
seqCol.add(new MusicJobExample());
seqCol.launch(JMSL.now());
```

5 Notes on Hierarchical Scheduling

JMSL uses Java Threads to play multiple Composable objects in parallel. Timestamps are passed up and down the hierarchy to ensure that the proper timing relations are preserved. A Composable is executed by calling launch() with a beginning time-stamp. A new Thread is created and the object's run() method is called, which in turns calls play(time).

The play(time) method of a MusicJob calls the start(time), repeat(time) and stop(time) methods, which can be overridden by the composer. The play(time) method blocks until the job has finished playing. It then returns the time that it has finished. A SequentialCollection simply has a repeat(time) method that calls each of its children's play(time) methods in sequence, passing the returned time on to the next child. The children execute in the parent's thread.

A ParallelCollection's repeat(time) method launches each of its children at the same time, so they each run in their own thread. It then waits until the last child has finished executing. It does this by checking for a done flag for each child, and calling wait() on each child that is not finished. As each child finishes, it sets its done flag, and uses notify() to wake up the waiting parent. The children also pass the parent the time that it finishes. The ParallelCollection uses the maximum child completion time as its own completion time.

Starting and running threads in Java has proven to be very easy. When they finish they exit gracefully. But unfortunately, stopping them prematurely is problematic. Sun originally provided suspend(), resume() and stop() methods for Threads. But these methods were deprecated because of problems with resources getting unlocked improperly. So the preferred method for stopping a thread is to set a flag, and then interrupt() the thread. The thread will wake up, see the flag and bail out through the normal path. One can call join() which will block until the thread exits. Unfortunately, both interrupt() and join() are seriously broken in Netscape's 4.77 (and earlier) JVM. So we are still experimenting to find the optimal solution.

6 MusicShape

An ordered table of abstract, multi-dimensional numerical data is contained in a MusicShape. MusicShape implements Composable, and so, can be independently launched, or added to a hierarchy. Melodies could be stored in a MusicShape, for example, where dimension 0 might stand for duration, dimension 1 might be pitch, and dimension 2 might be loudness. Data of more conceptual nature can be stored in a MusicShape as well, like parameters that feed a complex musical algorithm. It is the job of JMSL's Instruments and Interpreters to deliver this interpretation (see below).

The following code shows how to create a MusicShape and add it to a MusicShapeEditor, which allows real-time, mouse-based editing (see Fig 1).

```
// 3 dimensions
MusicShape s = new MusicShape(3);
// add as many elements as you like
s.add(0.50, 10, 100);
s.add(1.50,  9, 101);
s.add(0.75, 12, 102);
MusicShapeEditor se = new MusicShapeEditor();
se.addMusicShape(s);
```

Each row of data is called an element. Launching the MusicShape shown above results in an enumeration of its elements over time, using, in this case, a printing interpreter.

```
Output:
Interpreter_1 called by ins_0 with { 0.5, 10.0, 100.0, }
Interpreter_1 called by ins_0 with { 1.5, 9.0, 101.0, }
Interpreter_1 called by ins_0 with { 0.75, 12.0, 102.0, }
```

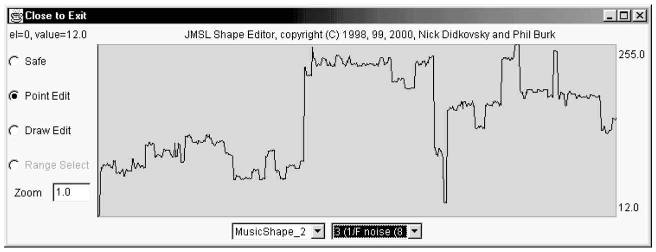


Figure 1. JMSL's MusicShapeEditor, showing a MusicShape loaded with 1/f data

7 Instruments and Interpreters

"Instrument" is a JMSL interface which is responsible for specifying the custom interpretation of MusicShape data. A MusicShape hands an element to its instrument as an array of doubles, receives an updated playtime, waits, then proceeds to the next element. A convenience class called InstrumentAdapter is included in JMSL, which already implements Instrument, and allows the programmer to simply override the play() method.

```
public double play(double playTime, double timeStretch,
double dar[]) {
        double duration = dar[0];
        double someOtherValue = dar[1];
        // do whatever you want here
        return playTime + duration * timeStretch;
}
```

An Instrument may optionally contain an Interpreter, whose interpret() method does the low level data handling and returns an updated time. The interpreter's interpret() method is passed a handle to the Instrument that invokes it.

```
playTime = interpreter.interpret(playTime, timeStretch,
dar, this );
```

8 Integration with JSyn

JMSL features tight integration with Phil Burk's JSyn synthesis API (Burk 1998). JMSL provides a SynthClock which can convert to native JSyn time.

```
JMSL.clock = new com.softsynth.jmsl.jsyn.SynthClock();
```

An Instrument which plays a JSyn SynthNote might implement play() as follows:

```
public double play(double playTime, double timeStretch,
double[] dar) {
  double dur = dar[0];
  double frequency = dar[1];
  double amplitude = dar[2];
  double hold = dar[3];
  int onTime = (int)JMSL.clock.timeToNative(playTime);
  int offTime = onTime + (int)(JMSL.clock.getNativeRate()
* hold * timeStretch);
  mySynthNote.noteOn(onTime, frequency, amplitude);
  mySynthNote.noteOff(offTime);
  return playTime + dur * timeStretch;
}
```

Additional support classes include JMSL's SynthNoteInstrument which allows the user to plug any predefined JSyn SynthNote into a JMSL Instrument.

9 JScore

JMSL's JScore Notation Package is a programmable music notation editor. It has a simple, but powerful API for notating algorithmically generated music, as well as API's for transforming notes that are already part of a score.

9.1 Notating Algorithmic Music

Adding algorithmically generated notes to a score is done with the addNote() method. Here we use a 1/f generator to add a stream of quintuplets to a score. See Fig 2 for notated results.

```
for (int i=0; i<40; i++) {
// dur, pitch, amp, sustaindur
 Note n = score.addNote(0.20, NoteFactory.MIDDLE_C +
oof.next(), 0.5, 0.10);
// beam groups of 5
  n.setBeamedOut(i % 5 != 4);
}
```

The duration passed to addNote() is compared to a table of triplets, quintuplets, septuplets, 11-tuplets, as well as all non-tuplet, dotted and un-dotted durations from whole note through 128th note. The closest duration is notated.

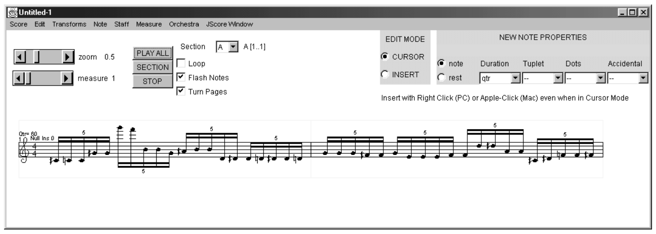


Figure 2. JScore notates algorithmically generated music

9.2 Unary and Binary Musical Transforms

JScore provides two abstract classes that apply transformations to notes selected by the user. The first is called UnaryCopyBufferTransform, which operates on the notes in the copy buffer (scrambling or retrograding a melody for example). The other is BinaryCopyBufferTransform which operates on two copy buffers (finding the mutation mean between two melodies for example (Polansky and McKinney 1991)). The programmer implements the operate() method of these classes, providing custom functionality. The source for ScrambleTransform's operate() method is shown below.

```
public void operate(CopyBuffer copyBuffer) {
 for (int i=0; i < copyBuffer.size(); i++) {
  Object temp = copyBuffer.elementAt(i);
  int swap = JMSLRandom.choose(copyBuffer.size());
  copyBuffer.setElementAt(copyBuffer.elementAt(swap), i);
  copyBuffer.setElementAt(temp, swap);
 }
}
```

The results of applying a BinaryCopyBufferTransform to two source melodies is shown in Fig 3.

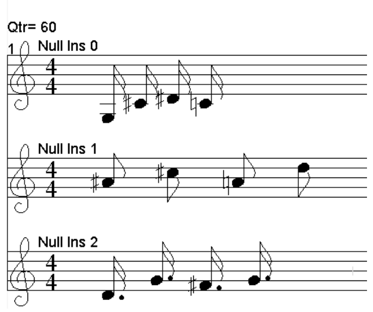


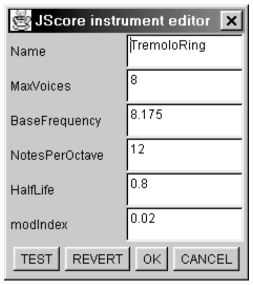
Figure 3. Mutation Mean Transform. Source melodies in top two staves. Result pasted into the third.

A custom transform can be plugged into JScore's menu with one line of code:

```
addBinaryCopyBufferTransform(new MutationMeanTransform())
```

9.3 JSyn and Midi Support

JScore supports orchestras made up of both Midi and JSyn instruments. It allows the user to import any JSyn SynthNote by simply typing in its class name. The ports of the SynthNote are sniffed using Java reflection, and JScore provides an editor which allows the user to change the SynthNote's port values, thus controlling its timbre (Fig 4).



JScore instrument editor

Name	TremoloRing
MaxVoices	8
BaseFrequency	8.175
NotesPerOctave	12
HalfLife	0.8
modIndex	0.02

Figure 4. JScore opens a dialog for editing public JSyn ports

9.4 Music Publishing

JScore exports its notation in a format readable by San Andreas Press's SCORE Computer Music Typography System, for professional music publishing.

10 Selected Pieces

"The Monkey Farm" by Didkovsky, 2001. A setting for stories by CW Vrtacek, read by Valeria Vasilevski, with music performed by Doctor Nerve. Live and sampled voice processing software by Didkovsky, Burk, and Marsanyi using JMSL and JSyn. www.doctornerve.org/monkeyfarm

"MandelMusic", by Didkovsky and student contributors, 1998-present. An applet which sonifies the mathematics of the Mandelbrot set. JMSL governs the scheduling and provides the API that allows a variety of composers to contribute their own JSyn/JMSL instruments. http://www.punosmusic.com

"Hell Café", by Nick Didkovsky. Created as part of the collaborative music theatre production "The Technophobe and the Madman" (Didkovsky, *et al*. 2001), "Hell Café" is a live JMSL/JSyn instrument which generates techno-inspired music, while processing and scratching live vocals.

"Slim In Beaten Dreamers", by Nick Didkovsky, 2000. Commission for the Meridian Arts Ensemble. The first composition composed entirely in JScore, for brass quintet and drum set. Commissioned by the Mary Flagler Cary Charitable Trust, with support from Harvestworks.

11 Future Work

A JMSL class browser and interactive hierarchy builder would be powerful additions. We would also like to improve JMSL's MusicShapeEditor, provide it with an API for transforming ranges of selected MusicShape data directly within the editor, as well as enabling the importing and exporting MusicShapes between the MusicShapeEditor and JScore. JScore shall support more common music notation requirements, transcribe from live Midi input, and read/write Midi files. We would like to see JScore output to the GUIDO notation system as a publishing alternative to SCORE. We also have plans to integrate JSyn's Wire patch editor with JScore.

12 References

Burk, P.L., 1998. "JSyn - A Real-time Synthesis API for Java." *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 252-255.

Didkovsky, N., 1997. "Schubert's Imp. in Eb Maj Op. 90, arr Minsky Popolov, a statistical deconstruction/ resynthesis." Premiered at The Alternative Schubertiade, American Opera Projects / Downtown Arts Festival, September 12, 1997. Released on The Alternative Schubertiade (Composers Recordings Inc, CRI CD809).

Didkovsky, N., Henderson, T., Perle, Q., Ritter, D., Rolnick, N., Rowe, .R., and Vasilevski, V. 2001. "The Technophobe and the Madman" Live Internet-2 distributed musical, premiered Feb 20, 2001 at NYU Lowe's Theatre and RPI's iEAR Space.

Orlarey, Y. and Lequay, H., 1989. "MidiShare: A Real Time Multi-tasks Software Module for MIDI Applications." *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 234-237.

Polansky, L., McKinney, M., 1991. "Morphological Mutation Functions: Applications to Motivic Transformation and a New Class of Cross-Synthesis Techniques." *Proceedings of the International Computer Music Conference*, International Computer Music Association, pp. 234-241.

Polansky, L., Rosenboom, D., and Burk, P. 1987. "HMSL: Overview (Version 3.1) and Notes on Intelligent Instrument Design." *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp 220-227.