Java Music Specification Language, v2 update

Nick Didkovsky
Algomusic
nick@didkovsky.com

ABSTRACT

Java Music Specification Language (JMSL) is a Java package for algorithmic music composition, notation, interactive performance, and intelligent instrument design. Its Java foundation and support for MIDI and JSyn (Java Synthesizer) offers the composer real-time musical performance applications and non-real time compositional systems. This paper describes a number of new features that have been added to JMSL and its notation package JMSL Score, including:

- *Support for pure Java JSyn V16+, including polyphonic signal processing instruments*
- *Support for JavaSound MIDI and javax Soundbanks*
- *Support for SMuFL Music Font in JMSL Score*
- *GNU LilyPond support*
- *Finer grain of instrument control: instrument.on(), update(), off()*
- *Token based management of polyphonic voice allocation*
- *RenderedMessageBean allows custom drawing commands to be embedded in a JMSL Score.*
- *Plug-ins*
- *Global control over JMSL's clock rate, allows for smooth playback changes of hierarchies*

1. INTRODUCTION

Java Music Specification Language (Didkovsky, Burk 2001) provides a flexible, portable, and stylistically neutral Java API for algorithmic composition and performance. Based on Hierarchical Music Specification Language (Polansky, Rosenboom, Burk 1987), its Java foundation along with support for MIDI, JSyn, and user-defined music devices offers the composer the power of a full featured programming language with flexible compositional objects and musical tools.

A number of advances have been implemented since v103 (Didkovsky 2004). The latest v2 release is available at www.algomusic.com.

2. SUPPORT FOR PURE JAVA JSYN V16+

JSyn is an audio Synthesis API for Java (Burk 1998). The current JSyn release is pure Java and includes a number of improvements to the API. JMSL supports the latest JSyn in its com.softsynth.jmsl.jsyn2 package which includes a new Instrument class that supports polyphonic voice allocation and control over all public input ports of any JSyn UnitVoice. JMSL also supports polyphonic signal processing instruments for any JSyn UnitVoice that implements UnitSink.

JMSL's JSynUnitVoiceInstrument accepts a JSyn UnitVoice class name and a polyphony value in its constructor. The following code excerpt creates an instrument with 8 voice polyphony, using a band limited filtered sawtooth JSyn circuit:

```
instrument = new JsynUnitVoiceInstrument(8,com.-
softsynth.jmsl.jsyn2.unitvoices.FilteredSaw-
toothBL.class.getName());
```

Upon construction, JSynUnitVoiceInstrument inspects the public inputs of the UnitVoice, and builds a JMSL DimensionNameSpace from their names, their low and high limits, and defaults. A JMSL MusicShape can be constructed from its DimensionNameSpace and filled with algorithmically generated data or with random values for each dimension using the convenient prefab() method like so:

```
 MusicShape s = new MusicShape(instrument.getDi-
mensionNameSpace());
 s.prefab();
 s.print();
```

The MusicShape print() shows that it mapped the JSyn input ports to dimensions:

```
musicShape_1
dim 0: duration, dim 1: pitch, dim 2: amplitude,
dim 3: hold, dim 4: resonance, dim 5: cutoff,
dim 6: rate
```

JSynUnitVoiceInstrument also supports the notion of a preset, so a constructor like the following...

```
ins = new JSynUnitVoiceInstrument(4, FMVoice.-
class.getName(), FMVoice.PRESET_WOODDRUM);
```

...supports the convenient setting of the UnitVoice's port values to achieve the desired sound.

A JSyn UnitVoice that implements UnitSink has a getInput() method. If a UnitSink is passed to the constructor of a JSynUnitVoiceInstrument, it will automatically support signal processing. The JSynUnitVoiceInstrument will support the method addSignalSource(). The following code excerpt shows how to connect the output of one instrument into a signal processing instrument:

```
delayInstrument = new JSynUnitVoiceInstrument(8,
com.softsynth.jmsl.jsyn2.unitvoices.SimpleDe-
lay.class.getName());
  delayInstrument.addSignalSource(myInstrumen-
t.getOutput());
```

A hardware delay unit typically has settings that stay unchanged over the course of a musical passage, or might have its parameters changed over time by control values. The JMSL/JSyn delayInstrument describe above, by contrast, is polyphonic and like any JMSL Instrument, can be played by a MusicShape that controls all its input ports. This means that virtual delay units with independent settings will be instantiated on demand, turned on

and off and scheduled over time. The composer can play a "chord" of delay units, for example, each with its own delay, feedback, and sustain values. JSynUnitVoiceInstruments can be assigned to staves in JMSL Score (Didkovsky 2004), offering the composer, in this case, the ability to write out delay events on a staff, where each Note has its own delay characteristics and processing the signals of instruments performing notes on other staves.

3. JAVASOUND MIDI SUPPORT

The MIDI capabilities of the Java Sound API are now fully supported by JMSL, including support for sound banks. The following code excerpt demonstrate how to open a JavaSound MIDI device, and load a soundbank:

```
JMSL.midi = new MidiIOFactory().getMidiIO(MidiIOFactory.MidiIO_JAVASOUND);

((MidiIO_JavaSound) JMSL.midi).setSoundbankFile(new File(LOCATION_OF_SOUNDFONT));

JMSL.midi.setOutDevice("Gervill");

JMSL.midi.open();
```

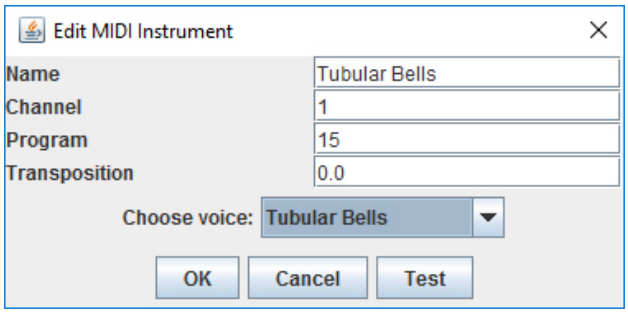


Figure 1. JMSL's Midi Instrument Editor allows you to interactively choose Midi channel, program change, transposition, and instrument name.

4. SUPPORT FOR SMUFL MUSIC FONT

Standard Music Font Layout, or SMuFL, is an open standard for music font mapping (www.smufl.org). JMSL's Score notation package supports SmuFL music fonts. By default it looks for and loads Bravura if it can find it on the user's system.
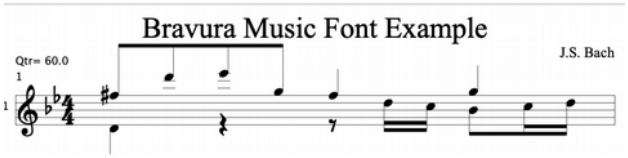


Figure 2. JMSL Score displaying a measure of music using the SMuFL music font "Bravura"

JMSL lays out a Score using abstract glyph tokens and their (x, y) positions. A glyphToken might be "noteheadblack" or "quarterrest". These glyphTokens are managed by a MusicGlyphRenderer, whose render() method maps the token to the symbol in a given SmuFL font, and may make minor adjustments to its x and y position, like:

```
//...
if (glyphToken.equalsIgnoreCase("noteheadwhole")) {
    glyph = BravuraFontConstants.NOTEHEAD_WHOLE;
            x -= 3 * zoom;
            y += 0.5 * zoom;
}
```

5. LILYPOND SUPPORT

LilyPond (www.lilypond.org) is a music engraving program that produces high quality scores from text input. JMSL supports LilyPond with its LilyPondWriter class. Each staff of the JMSL Score is exported to its own .ly file, then combined in a master .ly file to describe the full score. Note that JMSL continues to support MusicXML export as well.
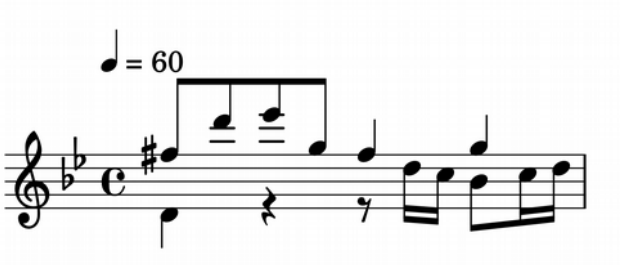


Figure 3. JMSL Score exports to Lilypond, producing the score shown here.

6. INSTRUMENT ON(), OFF(),UPDATE()

JMSL's Instrument interface now affords the user a finer grain of control. Its play() method still accepts a timeStamp and performance data as it always did, and can be used to play a singular event, but under the hood, play() calls instrument.on(), update(), and off(). These are public methods so the composer can, for example, turn an instrument on, update the sound it makes over time by passing in new data, then finally turn it off. The on() method returns an Object which can be used to identify the allocated voice when calling update() and off(). This provides voice identification for polyphonic instruments. In the case of JMSL's JSyn2 instruments, the on() method returns an Integer voice token which can be later passed to update() and off() . This way the user accesses the actual JSyn Circuit that on() allocated.

The following code excerpt illustrates turning a voice on, updating it with new data, then turning it off. This is taken from jmslexamples.jsyn2.SimpleTimbralEvolution example included in the JMSL distribution.

```
Object obj = getInstrument().on(playTime, 1.0, data);

myVoiceToken = ((Integer) obj).intValue();

// update the sustained sound (this can happen any number of times with different playTimes and data values)

((JSynUnitVoiceInstrument) getInstrument()).update(playTime, 1.0, data, myVoiceToken);

// turn it off:

((JSynUnitVoiceInstrument) getInstrument()).off(playTime, myVoiceToken);
```

7. RENDEREDMESSAGEBEAN

A bean in Java is a class with a no-argument constructor that can be serialized. JMSL defines a RenderedMessageBean class which contains an arbitrary message of type String, an x and y offset, and a name. The user can put any String into the message, such as "drawline 0 0 100 100" and specify offsets like x=10 and y=20 for example. These values are primitive data types and will save and load back from a Score file. The user can attach a RenderedMessageBean to a Note, a Staff, or a Measure. The bean will be activated when the object it is attached to is rendered. The bean's commands are interpreted by a user-defined RenderedMessageListener, which executes the commands relative to the drawing anchor of the object. A SimpleGraphicsRenderedMessageListener is included with JMSL which supports the commands drawline, drawarc, and setcolor. Of course the user can build their own RenderedMessageListener and pass commands to it in any format.

The RenderedMessageListener includes the execute() method shown below. Note that the zoom value of the Score is included so the listener can scale any rendering commands it interprets. Note too that the command in the bean can be any String, and since the user can write their own listener, there are opportunities to trigger arbitrary (non graphic) behavior when an object is rendered.

```
public void execute(String message, Score score, Object container,RenderedMessageBean renderedMessageBean, Graphics g, Point2D p, double zoom);
```

In the code excerpt below, a bean is created that draws a line. It is added to a Staff. When the Staff is rendered, the drawline command will be executed by the RenderedMessageListener, which will draw the line relative to the drawing anchor of the Staff.

```
RenderedMessageBean lineDrawingBean= new RenderedMessageBean();

lineDrawingBean.setMessage("drawline 0 0 30 50");

myStaff.addUserBean(lineDrawingBean);
```

In the example below, numerous beans, each with a single drawline command, were added to the notes in a Score. The drawline commands were generated by a recursive method and render Sierpinski Triangles.

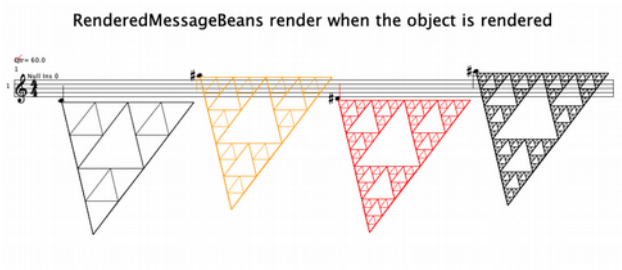


Figure 4. RenderedMessageBeans with "drawline" commands were added to Notes in a JMSL Score, rendering Sierpinski Triangles of various depths.

8. JMSL PLUGINS

ScoreFrame is the JMSL class that hosts numerous Scores and provides a variety of menus. Upon startup, ScoreFrame scans the classpath and if it finds a directory called "jmsl_plugins", it will look for plug-ins and build menus for them automatically. It scans for Instruments, UnaryCopyBufferTransform, BinaryCopyBufferTransform, JSyn UnitVoice, JSyn UnitSink, and ScoreOperator. To qualify as a plug-in, a class must extend one of the classes above and implement the JMSLPlugin interface. The JMSLPlugin interface does not require any methods to be implemented. It is simply used to identify a class as being a plug-in. This gives the user control over exactly which classes they want to populate ScoreFrame's menus.

The use of the JMSLPlugin interface makes it easy to design UnitVoices in Phil Burk's graphic patch editor Syntona (www.softsynth.com/syntona/), export the Java source, and simply add "implements JMSLPlugin" to the exported source. For example, "BehemothKachlerMod" is a JSyn Circuit that was designed in Syntona by one of the author's students. The designer exported the Java source, added "JMSLPlugin" to the source as shown below, compiled, and finally copied the .class files to the jmsl_plugins folder.

```
public class BehemothsLeadKachlerMod extends Circuit implements UnitVoice, JMSLPlugin
```

When ScoreFrame launches it detects this plug-in and builds the hierarchical menu shown in Fig. 5 below. The user can select this voice and assign the resulting JSynUnitVoiceInstrument to a Staff.

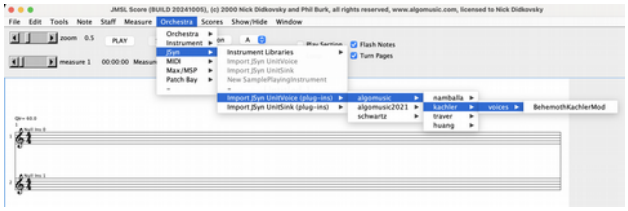


Figure 5. JMSL's ScoreFrame builds a hierarchical plug-in-ins menu based on the package names of the JMSLPlugins it detects.

9. VARIABLE CLOCK RATE

The current version of JMSL adds JMSL.clock.setRate() which globally changes the rate of any objects currently performing. The distribution includes an example where a nested hierarchy of MusicShapes is performed, and the user can change the overall playback rate with a scrollbar. The user hears all events speed up and slow down and stay in synch with each other while the rate changes.

Note this this is not the same as changing the timeStretch of the hierarchy. Changing the timeStretch does not interrupt sleeping events, so whole notes, for example would not stay in synch with finer grained 16th notes. By contrast, changing the JMSL clock's rate changes the the time scaler smoothly, so slow events stay in synch with faster ones.

9. CONCLUSIONS

Since its v103 release, Java Music Specification Language has seen a number of important improvements to its design. Some of these new features were prompted by the evolution of MaxScore (Hajdu, 2012) but remained unavailable to the JMSL composer/developer writ large until the current v2 release. JMSL v2 affords rich development tools for the composer / programmer in a powerful object oriented general purpose programming language.

Acknowledgments

The author would like to acknowledge the vital contributions Larry Polansky made to the contemporary music community with his ground breaking and insightful compositions, theories, and publications. Larry was a co-creator of HMSL, upon which JMSL is based. This paper and indeed JMSL itself stands on his shoulders. He is deeply missed.

10. REFERENCES

[1] 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.

[2] Didkovsky, N., Burk, P.L., (2001). "Java Music Specification Language, an introduction and overview" Proceedings of the International Computer Music Conference. International Computer Music Association, pp. 123-126.

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

[4] Didkovsky, N., (2004). "Java Music Specification Language,v103 update" Proceedings of the International Computer Music Conference. International Computer Music Association, pp. 123-126.

[5] Hajdu, G. and Didkovsky, N., (2012). "MaxScore - Current State of the Art." Proceedings of the International Computer Music Conference. International Computer Music Association.