Java Music Specification Language, v103 update

Nick Didkovsky

didkovn@mail.rockefeller.edu
www.algomusic.com

Abstract

Java Music Specification Language (JMSL, Didkovsky, Burk) 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, Burk) offers the composer real-time web deployment as well as the creation of stand alone musical applications. This paper describes a number of new features that have been added to JMSL and its notation package JScore, including:

- *A flexible transcriber which notates algorithmically generated music using heuristic search paths*
- *MusicXML export of notated scores*
- *A DimensionNameSpace interface which maps dimensions to names*
- *Instrument classes which provide tighter integration with JSyn*
- *Plug-in API that allow the composer to add algorithmic extensions to the notation editor.*

1 Introduction

Since its initial release, Java Music Specification Language has provided a flexible, portable, and stylistically neutral Java API for the algorithmic composer and performer. Its Java foundation along with its support for audio output via MIDI and JSyn enables compositions and interactive performance instruments to be deployed on the web and as standalone applications, while offering the composer the power of a full featured programming language.

A number of advances have been implemented in the latest v103 release, available from www.algomusic.com. Some of these changes extend and improve upon existing features, while others provide powerful new notions that extend the core philosophy of JMSL in important ways.

2 Transcribing algorithmically generated music

JMSL's Score package (JScore) provides the user with programmable common music notation. The new Transcriber class offers the composer transcription and notation of algorithmically generated music.

Consider the problem of notating event durations which do not conform to traditional durations. The user may have any number of note events scattered arbitrarily over time, generated stochastically for example, and wishes to create a score of this material in common music notation. JMSL's Transcriber analyzes the timestamps of such musical material, and loads a Score with a transcription.

The input to JMSL's Transcriber is a MusicShape (an ordered list of timestamped musical events) and a list of time signatures and tempos which provide a template for the transcription. The output is a notated Score. The power of the transcriber lies in its generality and customizability.

2.1 Customizing the transcriber

Besides providing a template of time signatures and tempos, the composer can customize the beat subdivisions considered by the transcriber (ie triplets, quintuplets, etc). The user can use a default list of beat subdivisions, or may specify precisely which beat subdivisions the transcriber may consider (allowing for example quarter note triplets, eighth note quintuplets, sixteenth note septuplets, and disallowing all others). All such "BeatDivisionSchemes" under consideration are contained in a BeatDivisionSchemeList. By adding BeatDivisionSchemes to this master list, the user can customize the transcriber, which will limit its analysis to those in the list. Excluded subdivisions are not considered.

Additional rhythmic customization can be achieved by specifying the minimum number of elements to be present for a particular BeatDivisionScheme to be considered. For example, one might specify that the minimum number of notes in an eighth note triplet be three, eliminating the possibility of a triplet containing two notes. One can massage this threshold and in general, specify a minimum of any value between 1 and N notes for an N-subdivision.

A powerful call-back mechanism is provided during the transcription process. The composer can create a TranscriberListener and register it with the Transcriber. Every time a Note is added to the Score, the listener is notified. This can been used to add algorithmically generated lyrics, slurs, dynamics, and markings to the Note.

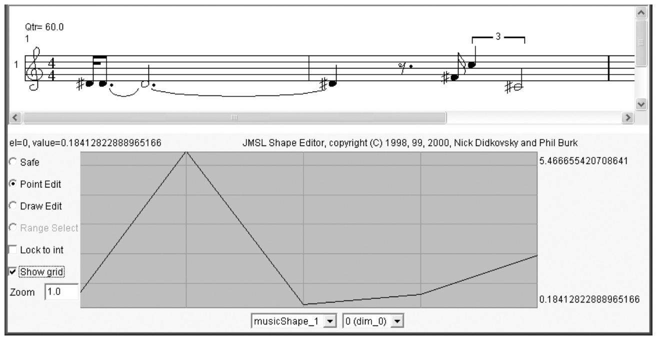


Figure 1. Exponentially distributed musical events notated by JMSL's Transcriber. The MusicShapeEditor in the bottom half of the figure shows the duration in seconds of the five events that were transcribed.

2.2 Heuristic Search for path of minimum error, some technical details

The transcriber's goal is to discover a path of BeatDivisionSchemes that expresses the measure's events with minimum error. An example of a winning path through a measure of 4/4 might be quarter note triplets spanning the first two beats of the measure, followed by a eighth note triplet for the third beat, ending with a single quarter note. The search space of all possible paths through a measure's beats grows quickly even when a modest number of possible BeatDivisionSchemes are considered. By default JMSL considers 21 distinct BeatDivisionSchemes. A measure of 4/4 yields a total of 21^4=194,481 possible paths. To find the minimum cost path efficiently, the transcriber limits this search space by following a heuristic search strategy. Instead of expanding all search paths for every beat, it expands at each beat only the one path which is ranked with minimum error. It expands this lowest cost path into the next beat by building multiple copies of it, each with a new BeatDivisionScheme at its head. After recalculating error of these new paths as well as all old paths, it resorts and again chooses the minimum cost path for expansion into the next beat. An old path might then have the lowest error, which promotes it for expansion. This process is repeated until the last beat is reached and an overall winning path is determined. Note that backtracking is implemented here, since a path which did not promise minimal error early in the search may win in the end.

3 MusicXML support

MusicXML is an XML standard defined by Recordare LLC (Good) that describes common western music notation. JMSL's Score package now exports scores in MusicXML format providing a bridge to a variety of commercial music applications. Recordare's Dolet plug-in for example, imports MusicXML files directly into Finale for professional quality publishing. Currently, the Dolet plug-in runs on Windows platforms only. Recordare has announced that an OS X version is under development.

The MusicXML file exported by JScore preserves tempo and time signature changes, marks, dynamics, crescendos, beaming, etc. JMSL's ability to generate algorithmic music, to notate it with its transcriber, and then have it rendered in Finale offers the composer a solution path for getting algorithmically generated music onto the printed page.

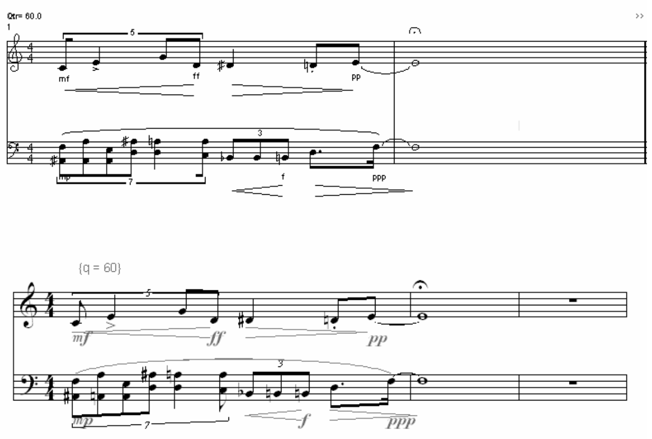


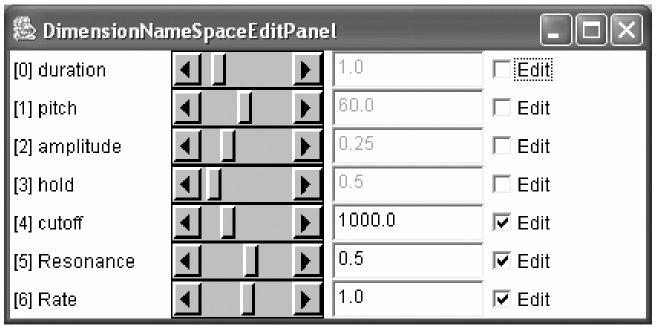
Figure 2. The JMSL Score pictured in the top half of the figure was exported in MusicXML format and loaded into Finale via the Dolet plug-in. The Finale excerpt is pictured in the bottom half of the figure.

4 Dimension Name Spaces

A new JMSL interface called DimensionNameSpace enables new tools and an enhanced runtime environment. One of the designs that JMSL inherited from HMSL (Polansky, et al) is MusicShape. Each row of MusicShape is an array of double[] called an element. Each column is a dimension. MusicShape data is sonified or otherwise interpreted by JMSL Instruments. MusicShape has always provided the user with the ability to assign names to its dimensions, such as "duration" for dimension 0, "pitch" for dimension 1, etc. The new DimensionNameSpace interface provides an abstract definition of this mapping, outlining methods that assign and retrieve integer/name relationships.

With the DimensionNameSpace interface, mapping dimensions to names is no longer unique to MusicShape. For example, Instrument now contains a DimensionNameSpace field which it may use to access performance data by name instead of by array position. Later we will see how this gives rise to powerful new JSyn support classes.

Besides the convenience of being able to address a dimension index by name, the DimensionNameSpace interface enables the creation of general utilities like DimensionNameSpaceEditPanel which builds an editor dynamically according to the DimensionNameSpace of the object passed in.



[0] duration	1.0	☐ Edit
[1] pitch	60.0	☐ Edit
[2] amplitude	0.25	☐ Edit
[3] hold	0.5	☐ Edit
[4] cutoff	1000.0	☑ Edit
[5] Resonance	0.5	☑ Edit
[6] Rate	1.0	☑ Edit

Figure 3. This DimensionNameSpaceEditPanel configured itself at runtime to edit an object's synthesis parameters.

4.1 Translating between name spaces

The DimensionNameSpaceTranslator translates an array of double from one DimensionNameSpace to another, preserving data by matching common names. For example, if one DimensionNameSpace had dimension 4 mapped to "cutoff" and another DimensionNameSpace mapped the same name to dimension 5, then translating from the former to the latter would return an array where the value in dimension 4 of the source would be copied into dimension 5 of the target. MusicShape now uses a DimensionNameSpaceTranslator to mediate between its own DimensionNameSpace and that of its Instrument. This ensures that the data it passes to the Instrument conforms to the expectations of the DimensionNameSpace contained in the Instrument. This translation feature affords the composer the flexibility of passing abstract data around to a variety of Instruments, preserving as much of its semantics as possible (for example, sharing the same MusicShape data between two different instruments which have a dimension in common named "attackRate").

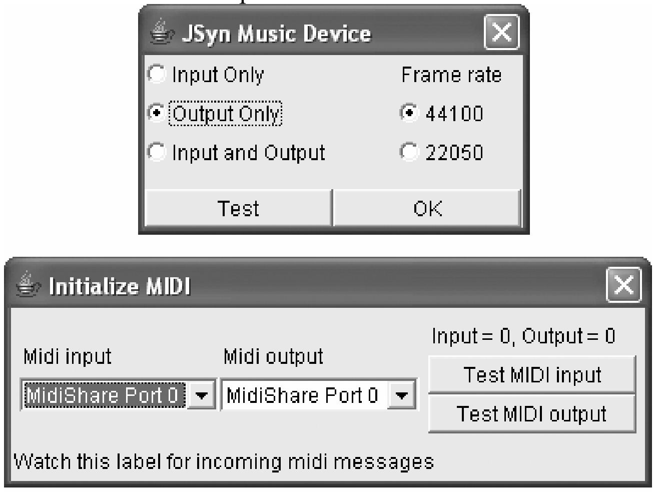
4.2 Runtime control of all synthesis parameters

The first JMSL release provided a manageable but somewhat inconvenient path to build a JMSL Instrument that provided timbral control over a JSyn SynthNote. This involved using a code generator class and compiling the Java source it exported.

The current JMSL release offers a runtime approach that is much simpler and much more powerful. The new SynthNoteAllPortsInstrument class is passed a JSyn SynthNote class name at runtime and automatically builds a DimensionNameSpace and a voice allocator for that SynthNote. Each call to Instrument.play() provides control over all input ports of the SynthNote. Together with the new Instrument.update() method, which updates parameters without retriggering a SynthNote's "on" stage, JMSL now offers intimate control of synthesis, supporting note on/off events as well as continuous control.

5 Music Devices

New to the current JMSL release is the MusicDevice interface. It consists of three methods: open(), edit(), and close(), and provides a useful abstraction layer through which JMSL's output devices such as JSyn and Midi can be defined and managed. JSynMusicDevice and MidiMusicDevice are singleton classes that implement MusicDevice. When open()'ed, they register themselves with JMSL which simply sees them as abstract MusicDevices. This permits convenience methods such as JMSL.closeAllMusicDevices() which calls close() on all registered MusicDevices (typically called at the end of a piece). A MusicDevice's edit() method returns a custom panel which can be used to set startup parameters unique to the MusicDevice's implementation.



Input Only [✓ Output Only] Input and Output

Frame rate: [✓ 44100] 22050

Midi input: MidiShare Port 0

Midi output: MidiShare Port 0

Input = 0, Output = 0

Watch this label for incoming midi messages

Figure 4. The MusicDevice interface includes an edit() method which returns a device-dependent GUI. Editors for JSynMusicDevice and MidiMusicDevice are shown here.

Among the benefits of this general approach is streamlining. For example, JScore's three Orchestra subclasses, JSynOrchestra, MidiOrchestra, and HybridOrchestra, have disappeared. Now there is only one Orchestra class to which Instruments of any type can be added. The Instruments themselves know what their MusicDevices are, and so, can be edit()'ed and open()'ed without other objects needing to have access to their implementation. Therefore, a well designed Instrument sets its preferred MusicDevice in its constructor, as is done for example, in the constructor of TunedSynthNoteInstrument which calls setMusicDevice(JSynMusicDevice.instance()). An Instrument's constructor should not assume the device is opened, and so, it should do all its device-specific building in a method called buildFromAttributes(). This enables the programmer to create a new Instrument object of any type, query it for its MusicDevice, optionally open a device editor, open the device itself, build the instrument, and continue with music creation.

6 Mixers

The Mixer interface describes general methods for mixing instrument output. Methods such as addInstrument(), setFaderMute(), start(), and stop(), describe common operations performed on a virtual mixer. Well designed instruments define their preferred mixer in their constructor, such as setMixerClassName("com.softsynth.jmsl.jsyn.JSynMixer") or setMixerClassName("com.softsynth.jmsl.midi.MidiMixer"). JMSL provides Midi and JSyn specific mixers but encourages the use of the general JMSLMixerContainer. Adding an Instrument to a JMSLMixerContainer queries the instrument for the class name of its preferred Mixer, and creates one if necessary. JMSLMixerContainer can therefore manage a heterogeneous collection of Midi and JSyn instruments by instantiating submixers for them. JMSLMixerContainer builds a GUI of Pan/Amp panels, one for each instrument. Any class that implements Mixer can be managed by JMSLMixerContainer, providing the programmer with the opportunity to define new Mixers for future sound engines and other output devices.

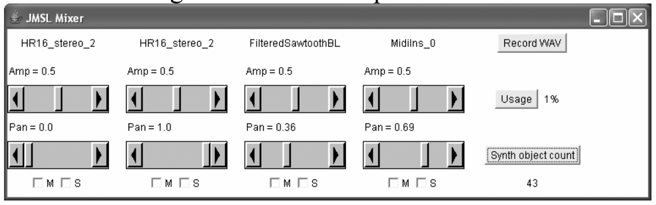


Figure 5. JMSLMixerContainer manages a hybrid collection of device-dependent Mixers. JSyn and MIDI instruments have been added to the mixer whose GUI is shown above.

7 Plug-ins

JMSL currently ships in a package that makes it convenient to double click on an icon, open a JScore, and begin editing in common music notation. The distribution also contains a folder called jmsl_plugins. User defined Java class files can be dropped into this folder. When Score starts up it scans jmsl_plugins for supported classes and builds hierarchical menus for these. Supported classes include JSyn SynthNotes and signal processors (which are SynthNotes with a SynthInput named "input"), Instruments, Unary and Binary CopyBufferTransforms, Score Reporters, and NotePropertiesTransforms. Composers can develop plug-ins themselves and customize a very personal set of compositional tools. This new feature also gives third party developers a uniform deployment target for plug-in development and distribution.

8 Web deployment of Scores

The JMSLScoreApplet class offers simplified deployment of notated scores on the web. The user passes the filename of a score as the “name” parameter to an applet tag which also specifies JMSLScoreApplet as its code parameter and jmslclasses.jar as its archive parameter. The applet loads and displays the score when the page is loaded into a web browser. Browsers with the JSyn plug-in installed can perform scores with CD quality audio directly on the client computer. The end-user can edit, copy, paste, add and delete notes, and interact in many other ways with the score, including using JMSL Transforms to develop and change material. Support for uploading scores to a server could be implemented in a straightforward way by subclassing JMSLScoreApplet. Since JScore has all the power of JMSL beneath it, self-generating, self-examining, and self-modifying scores can be deployed.

Score files can now be saved and loaded in ZIP format, making the download of even very lengthy scores quick and manageable.

9 Conclusion

JMSL has matured since its first release, offering a more streamlined and efficient API for composing algorithmic music and designing interactive music performance systems. Its new transcription capabilities and MusicXML support offers the composer a path that begins with arbitrarily complex musical algorithms and integrates with professional music publishing tools. Its support for the JSyn synthesis API has been simplified and offers algorithmic control and scheduling of user-designed synthesis patches. The abstraction of core notions such as dimension name spaces, mixers, and music devices benefit the current API and pave the way for growth. Its new plug-in API offers the composer rich and personal customization of a composition environment, and provides a standardized path for third party plug-in development.

10 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., 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.

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.

Good, M., (2001). “MusicXML for Notation and Analysis. In The Virtual Score: Representation, Retrieval, Restoration” *Computing in Musicology 12.* MIT Press, pp.113-124