Java Music Specification Language and Max/MSP

Nick Didkovsky[*], Langdon Crawford[†]

[*]Rockefeller University
didkovn@mail.rockefeller.edu, www.algomusic.com
[†]New York University
langsound@langdoncrawford.com

Abstract

Java Music Specification Language (Didkovsky, Burk 2001) is a Java package for algorithmic music composition, notation, and interactive performance. Max/MSP (Puckette, Zicarelli) is a graphical environment for music, audio, and multimedia. The introduction of a Java API to Max/MSP offers new possibilities for rich interaction between JMSL and Max. This paper presents new tools for bidirectional interaction between JMSL and Max/MSP. A MaxObject is presented that transcribes and notates Max-generated melodies using JMSL's Score package. Then we present a general purpose interface through which JMSL can control Max/MSP patches in real-time.

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 (Burk 1998) 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.

Max/MSP is a widely used graphical environment for creating computer music and multimedia works using a paradigm of units and connections. While Max is not a general purpose programming language, the rich body of work created in Max speaks for its flexibility and ease of use.

The introduction of a Java API to Max offers us the unique opportunity to program interactions between the two environments. Max users can leverage off JMSL's polymorphic hierarchical scheduler, true object oriented programming with well defined musical abstractions, transcription, and common music notation. JMSL users can benefit from the rich set of units available for Max, such as units that read real time sensor data, the MSP sound engine, and the ease with which graphical user interfaces can be built.

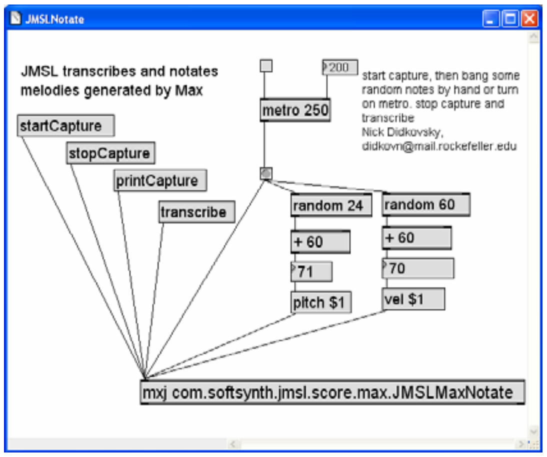
2 Max to JMSL: transcribing Max-generated melodies

While generating algorithmic melodies with Max is straightforward, notating these melodies is not. It is of course possible to capture a performance as a MIDI file and load the MIDI file into a commercial notation program, but not all such applications handle transcription well, and in all cases, much musical intelligence is lost in the process of writing out a MIDI file.

JMSL's Transcriber performs a heuristic search on musical input and transcribes it to common music notation (Didkovsky 2004). Via Max's Java API, we show here a straightforward tool for transcribing Max melodies into JMSL without ever leaving the Max environment.

JMSL ships with a Java class called JMSLMaxNotate which extends MaxObject. JMSLMaxNotate supports the following messages: startCapture, stopCapture, printCapure, and transcribe. It has public "pitch" and "vel" ports, through which it receives MIDI style note data. A bang message causes it to capture a time-stamped pitch/vel pair from Max.

The patch in figure 1 generates a melody using chance operations. The pitch and vel outputs of the melody generator connect to the JMSLMaxNotate unit.


JMSL transcribes and notates melodies generated by Max

start capture, then bang some random notes by hand or turn on metro. stop capture and transcribe
Nick Didkovsky,
didkovn@mail.rockefeller.edu

Figure 1. An example of the JMSLMaxNotate patch, showing an algorithmic melody generator which JMSL captures and notates

When the user clicks startCapture, JMSLMaxNotate starts logging the pitch/vel data it receives on every bang. Sending a "transcribe" message triggers JMSL's Transcriber to operate on captured event data, generating a JMSL Score, which opens in a new window (see fig 2).

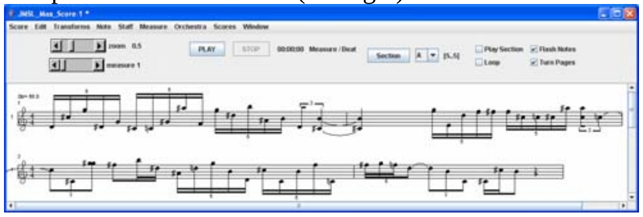


Figure 2. JMSL transcription of a melody generated by Max

The score can be further edited by hand or algorithmically mutated using JMSL's various transform classes and plug-ins. It can be saved and reopened later in JMSL outside the Max environment. The score can also be exported to San Andreas Press's SCORE Music Typography System, or in MusicXML format for import into Finale.

Source code for JMSLMaxNotate is provided in the JMSL distribution, so users have a model to extend and write their own custom versions. Customizations might choose to preserve higher level musical ideas such as algorithmically generated lyrics, dynamic marks, and expression symbols. These could be sent to new public fields added to a subclass of JMSLMaxNotate. Such properties, which would otherwise get lost by writing to a MIDI file, could be piped directly to JMSL at the moment they are generated by Max, preserving their musical meaning.

3 JMSL to Max

We have created a general purpose interface through which JMSL can control Max/MSP instruments in real-time. This consists of tools built in both JMSL and Max. Both sets of tools are detailed below.

3.1 Supporting JMSL classes

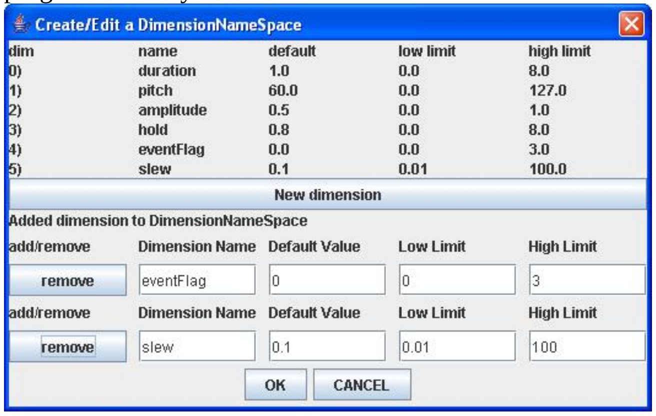
On the JMSL side, the essential new classes are MaxInstrument, MaxDimensionNameSpace, and JMSLInstrumentToMax. MaxInstrument is a JMSL Instrument which sends time-stamped performance data to a static instance of JMSLInstrumentToMax. The latter is a subclass of MaxObject, and so provides the actual conduit from Java to Max. After repackaging the data passed to it from a MaxInstrument as an array of float, JMSLInstrumentToMax passes it to its Max outlet(). JMSLInstrumentToMax is also responsible for synching JMSL's clock to Max's clock, and translating JMSL timestamps to Max timestamps. Through this scheme, the composer can take advantage of JMSL's polymorphic hierarchical scheduler to create flexible musical forms and output to Max/MSP.

More details about MaxInstrument JMSL has a well-defined Instrument interface which we leverage as the means to send performance data to Max. Instrument defines the following methods:

```
   public   Object   on(double   playTime,   double
timeStretch, double dar[])

   public   Object   off(double   playTime,   double
timeStretch, double[] dar)

   public   double   play(double   playtime,   double
timeStretch, double[] data)

   public  double  update(double  playtime,  double
timeStretch, double[] data)
```

All Instrument methods receive a time stamp, performance data, and a "timestretch" value to scale duration. The on() method will typically start a sound, while off() will turn it off. Play() is used to sound a voice at a particular time, and sustain it for a specified "hold" time. Finally, update() is used to change parameters on a voice that was either started with a call to on(), or a voice that is still sustaining from play().

JMSL Instruments also contain a DimensionNameSpace, which is a map of indexes (array positions 0, 1, 2, 3...) to meaningful names like "duration", "pitch", "amplitude", "hold", etc. The new MaxDimensionNameSpace adds a dimension called "EventFlag" which is used by Max to identify incoming data as on(), off(), play(), or update(). Dimensions with an index higher than 4 can be given custom interpretation, such as "slew", "cutoff", etc. The composer may use JMSL's DimensionNameSpaceEditor (fig 3) to create a custom MaxDimensionNameSpace that is meaningful for a particular MSP patch. Of course, the DimensionNameSpace may also be generated programmatically.


Create/Edit a DimensionNameSpace

dim	name	default	low limit	high limit
0)	duration	1.0	0.0	8.0
1)	pitch	60.0	0.0	127.0
2)	amplitude	0.5	0.0	1.0
3)	hold	0.8	0.0	8.0
4)	eventFlag	0.0	0.0	3.0
5)	slew	0.1	0.01	100.0

New dimension

Added dimension to DimensionNameSpace

add/remove	Dimension Name	Default Value	Low Limit	High Limit
remove	eventFlag	0	0	3

add/remove	Dimension Name	Default Value	Low Limit	High Limit
remove	slew	0.1	0.01	100

OK CANCEL

Figure 3. JMSL's DimensionNameSpaceEditor specifies custom dimension names for an MSP patch

Each MaxInstrument maintains an index that identifies itself uniquely. When on(), off(), play(), or update() are called during a JMSL performance, MaxInstrument passes incoming playtime, timeStretch, performance data, and instrument index to Max via the static JMSLInstrumentToMax conduit. Each method sets the appropriate eventflag in data[4], identifying the event as having resulted from a call to on(), off(), play(), or update().

Details about JMSLInstrumentToMax When JMSLInstrumentToMax is instantiated, the first thing it does is synch JMSL.clock's current time to MaxClock's current time. From that moment on, JMSL time can be converted to a Max timestamp.

JMSLInstrumentToMax receives performance data from MaxInstrument through its sendToMax() method. It prepares a float[] array and loads it with values relevant to Max, dropping values that are only meaningful to JMSL. For example duration, which is the time between events scheduled by JMSL, is not passed to Max, nor is timestretch (we elect to scale sustain time before passing to Max). To summarize, the sendToMax() method packs an array with the following data before passing it to its outlet():

arr[0] MaxInstrument index
arr[1] MaxClock timestamp
arr[2] pitch (fractional pitches ok)
arr[3] amplitude (0..1)
arr[4] hold (ie sustain time)
arr[5..n] timbral values

3.2 Supporting Max Objects

Figure 4 shows a demo patch which ships with JMSL. When the user clicks the NewJMSLScore object, an empty JMSL Score opens, each staff assigned to a different MaxInstrument. Notes entered on a staff send their performance data to Max via the play() method. Each note can be edited so that different MSP synthesis parameters may be assigned to each (this can be done programmatically as well). Notes that are tied-in send their performance data to Max with a call to update(). Each tied-in note can also have different synthesis parameters, causing a sounding note's timbre to change over time.

While this demo uses common music notation, we emphasize that the model presented here is independent of notation. MaxInstruments can be performed by any JMSL Composable, in very abstract musical contexts. Also, since MaxInstrument supports on(), update(), and off(), real-time voice allocation and synthesis updates is implemented, allowing for musical forms that do not require prior "note-like" knowledge of how long a sound will last.

JMSLInstrumentToMax Any Java subclass of MaxObject can be instantiated in Max with an object box containing [mxj package.class]. JMSLInstrumentToMax is a MaxObject and provides the entry point for Max's receipt of real-time JMSL performance data.

As performance data arrives in JMSLInstrumentToMax, we first pass it through a [route] object to determine the MaxInstrument index. Figure 4 shows instrument 0 being sent to a square wave MSP patch while instrument 1 is sent to a sine patch.

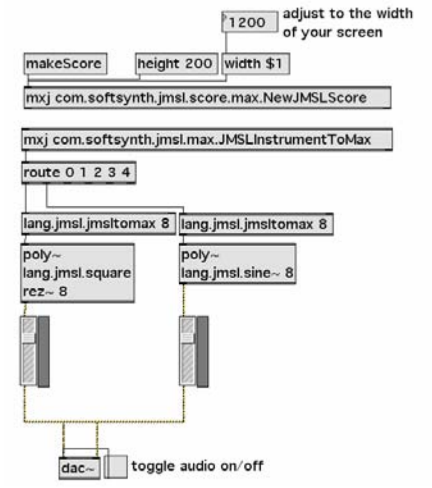

1200
adjust to the width of your screen
makeScore
height 200
width $1
mxj com.softsynth.jmsl.score.max.NewJMSLScore
mxj com.softsynth.jmsl.max.JMSLInstrumentToMax
route 0 1 2 3 4
lang.jmsl.jmsltomax 8
lang.jmsl.jmsltomax 8
poly~ lang.jmsl.square rcz~ 8
poly~ lang.jmsl.sine~ 8
dac~
toggle audio on/off

Figure 4 JMSL performance data sent to two MSP patches

Polyphonic voice allocation The routed data is sent to [jmsltomax], which encapsulates two objects, [mspmap] and [VoiceSet]. The former reformats incoming data in a manner compatible with MSP (a different formatter is available for [csound~], another for [rtcmix~]). The [VoiceSet] patch handles polyphonic voice allocation and retrieval (see fig 5). It sends event and parameter data to the appropriate voice number in the [poly~] patch. The voice number is assigned based on the pitch parameter and the event flag. Each event with an event flag of 3 (play) or 1 (on) gets a new voice. The pitch and voice number are then stored in a lookup table, for later reference if needed. Event flagged as 2 (update) or 0 (off) require parameter data to be sent to the voice which is playing the sound associated with the same pitch. The pitch parameter is used to look up which voice is currently playing that pitch. A second lookup table is used to make sure voices that are busy sounding sustaining notes are not interrupted. If all the voices in the [poly~] patch are sustaining, voice stealing is implemented. The event flags 1 (on) and 2 (update) set a voice as busy. The event flags 3 (play) and 0 (off) set the voice as not busy. Parameter data can be passed to busy voices when the event flag is set to 2 (update) and 0 (off), thus allowing for continuous control timbre parameters in currently sounding voices.

Time stamped musical output JMSL's scheduler corrects for timing variations inherent in Java Threads. By scheduling an event a little in the future, vagaries in Thread scheduling or complex calculations can be absorbed so long as they do not take more time than the advance time window (Anderson/Kuivila 1986). Provided that the underlying sound engine supports accurate future time-stamped output, erratic timing and other loads presented by computation or system activity will not interfere with rhythmic accuracy.

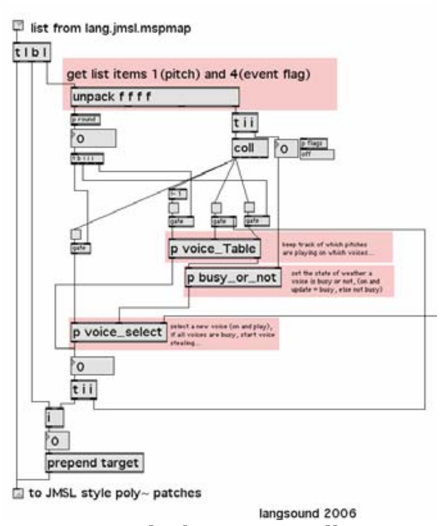


Figure 5. Polyphonic voice allocation and retrieval

Unfortunately, in Max all events happen in the present, so the creation of an object to handle time-stamped future events was required. The [offset] patch addresses this issue (see fig 6). It examines an incoming event's timestamp, compares it to the current Max time, and delays the event for the difference. It cannot handle timestamps arriving out of order, but since JMSL's scheduler assures us that timestamps do arrive in sorted order, this scheme works nicely.

This timing offset patch is placed inside each voice so that timing for every note may be delayed exactly as short or long as needed. Each delay line is cleared before starting the next delay so no old data is accidentally passed through.

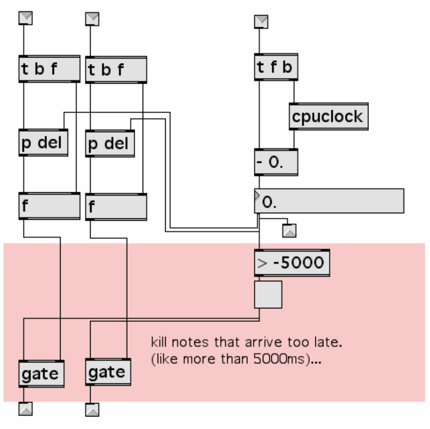


Figure 6 The [offset] patch delays time-stamped events until a specified future time

Substituting your own MSP patch It is straightforward to substitute custom MSP patches into this model. Easiest is to open one of the patches that ships with the demo, and save a copy under a new name (see fig 7). Then open the sub patcher labeled [p synth] and plug in a new circuit as desired. The custom MSP patch will receive at least 5 parameters which should be unpacked: Pitch, amplitude, duration (sec), event type (0..3) and timestamp. Parameters above that can be unpacked and connected as necessary to custom inlets, thus providing full control over the user's own MSP sounds.

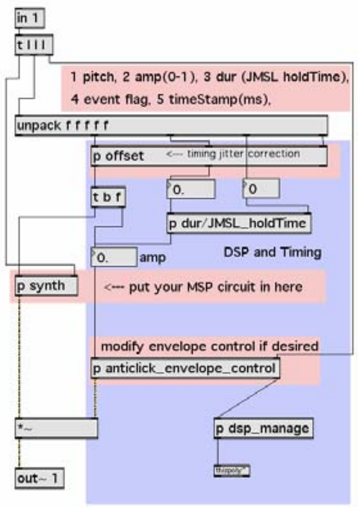

in 1
t l l l
1 pitch, 2 amp(0-1), 3 dur (JMSL holdTime),
4 event flag, 5 timeStamp(ms),
unpack f f f f f
p offset
<--- timing jitter correction
t b f
0.
0
p dur/JMSL_holdTime
0.
amp
DSP and Timing
p synth
<--- put your MSP circuit in here
modify envelope control if desired
p anticlick_envelope_control
*~
p dsp_manage
out~ 1

Figure 7 Support for custom MSP patches.

4 Conclusion

Max's Java API opens the way for bidirectional communication between JMSL and Max/MSP. JMSL can receive and process data generated algorithmically by Max, and to illustrate we presented a tool that transcribes Max-generated melodies to common music notation in JMSL. Conversely, Max users can leverage off the flexible polymorphic hierarchical scheduler and musical abstractions provided by JMSL, and pipe JMSL-generated performance data to Max/MSP, CSound, and RTcmix. Using JMSL and Max together offers composers rich new musical possibilities.

References

Anderson, D. P. and R. Kuivila, (1986) "Accurately Timed Generation of Discrete Musical Events", Computer Music Journal, Vol. 10, No. 3, pp. 48-56.

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. (2004). "Java Music Specification Language, v103 update" *Proceedings of the International Computer Music Conference*. International Computer Music Association, pp. 742-745.

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.

Puckette, M., Zicarelli, D. MAX Development Package. Opcode Systems, Inc., 1991.