# DYNAMIC, INSTANCE-BASED, OBJECT-ORIENTED PROGRAMMING IN MAX/MSP USING OPEN SOUND CONTROL MESSAGE DELEGATION

*Adrian Freed*
CNMAT
Dept. of Music
UC Berkeley, CA 94709
adrian@cnmat.berkeley.edu

*John MacCallum*
CNMAT
Dept. of Music
UC Berkeley, CA 94709
john@cnmat.berkeley.edu

*Andrew Schmeder*
CNMAT
Dept. of Music
UC Berkeley, CA 94709
andy@cnmat.berkeley.edu

## ABSTRACT

A new media programming style is introduced that brings efficient run-time polymorphism, functional and instance-based object-oriented programming to Max/MSP and related visual dataflow languages. Examples are presented to illustrate new, unusual and effective applications of the approach of using OSC messages for object representations and data flow for method delegation.

## 1. INTRODUCTION

Open Sound Control (OSC) was originally designed as a message-passing format to facilitate exchange of control parameters between programs on different computers across a network. Since its release in 1997 [1] OSC has proven to be useful for message exchanges between processes on the same computer system and more recently within processing modules in the same program [2]. This paper shows how OSC messages can provide the missing element (composable, dynamic data types) necessary to add object-oriented programming to dynamic, visual dataflow languages such as Max/MSP and PD.

### 1.1. History

The core data types in Max/MSP (i.e., numbers, strings, and arrays) are typical of languages that were in common use as far back as the 1950's, e.g. Fortran (1957) and Algol (1958).Programs in Max/MSP are expressed using the visual dataflow approach that was first demonstrated in 1966 [3].

CNMAT's limited use (1997) of OSC messages as an aggregate datatype in Max/MSP [4] added what was broadly available in the late 1950's and 1960's in the form of "records" in Cobol (derived from FACT), the "description lists" in IPL, then "property lists" in Lisp and perhaps better known as the "records" of Pascal or "structs" in C.

Regular expressions were designed into OSC originally to facilitate dynamic message dispatch, a core idea in Smalltalk (mid-1970's). Few programmers exploited the potential of using the Max external OSC-route and regular expressions to implement dynamic object-oriented programming in Max–presumably because of the hegemony of statically-typed class-based object-oriented programming invented in Simula (1962) and still active in Objective-C, C++ and Java.

In 2007 the first author developed a suite of Max/MSP patches, called "o." (pronounced "Oh dot"). These were specifically designed to exploit generic programming (pioneered in Ada in 1980) to simplify and teach gesture signal processing in physical computing contexts. This early prototype explored the use of OSC (Sections 2-3 of this paper), ad-hoc polymorphism and delegation (Sections 4 and 6) for dynamic class-less object-orientated programming (OOP) based on his experience developing music programs in prototype-based OOP with the NewtonScript language [5]. With this early library Max programming was possible using many of the valuable new ideas in programming languages from the mid-1980's, e.g. Self [6] and now thoroughly embedded in JavaScript (1995), for example.

In the course of developing an efficient reimplementation of the "o." suite as library in C, the second author created a particularly compact way to express unpacking and reassembling of OSC packets within Max/MSP. Realizing this was a run-time implementation of defunctionalization [7] we added lambda lifting to transform closures (symbolic Max object descriptions) into function objects [8]. The construction of this machinery resulted in the primitives necessary for functional programming in Max/MSP – a key aspect of most programming languages developed in the 1990's. The important practical contribution in "o." implementation is that any Max patch or external can be mapped or applied over pieces of an OSC packet (As explained in Section 5).

We have started to explore applications of the freely available "o." library for gesture signal processing [9] and for other musical applications such as the customizable note editor of Section 9.

We have found the library to be valuable tool to leverage the high degree of composability [17] that emerges when delegation, aggregation and mapping techniques of object-oriented programming are melded to dataflow execution models.

## 2. MISSING TYPES

Max/MSP and PD are among the most popular programming languages for media computing and especially musical applications [10, 11]. An unfortunate legacy of the early success of these programs is their spartan support for data types and the lack of objects and a composable and extensible type system. Although type systems are harder to integrate into dataflow languages than conventional compiled procedural languages [12], theoretical and practical challenges for this have been overcome [13, 14] as evidenced in Ptolomy II.

The developer community has addressed the type limitations of Max/MSP and PD by creating predefined opaque data structures and collections of objects that operate on them, such as Jitter and FTM [15]. The provided types are domain-specific and not composable into aggregate types. Users of these systems are required to learn a large number of primitive operations that only work on the new types. In these systems type parameterization is narrowly confined to being able to set the dimensions of predefined numeric matrix types and neither first-class objects nor the primitives necessary for dynamic object-oriented programming [6] are provided.

Cycling74 has deferred the clean native integration of aggregate data types, offering instead an integration of Java and JavaScript into Max/MSP, thereby covering both class-based and instance-based object-oriented programming. This requires the learning of new programming languages, and addressing their particular scheduling and integration constraints. Lua integration in Max/MSP has also been explored [16].

The solution advanced in this paper is to use Open Sound Control messages and native Max/MSP patches and externals to implement objects for dynamic, instance-based object-oriented programming (sometimes referred to as prototype-based programming). Self [6], ECMAScript [17], JavaScript and NewtonScript [18, 19] are examples of languages using this classless programming style [20] [21] [22].

## 3. OSC MESSAGE FEATURES FOR TYPE REPRESENTATION

The key idea of Open Sound Control is that user-chosen names ("addresses") are bound to data values. This is more than just the old idea of naming variables: because OSC messages move out of one program's context across a network into another's, the name/value pairs carry the meaning of the values to the destination. This has great practical value to the programmer, and helps with documenting complex systems. Bundles, the second important idea in OSC, allow many name/values pairs to be bound in a single, atomic structure. These two ideas are sufficient to allow OSC bundles to represent the property lists of Lisp, and SQL tables for example. The ordering of name/value pairs in OSC bundles is sufficient to represent the ordering requirement of C structures.

Type tags are the third feature of OSC necessary to support type introspection (a necessary feature for polymorphism) and run time type checking in instance-based programming. Also, type tags and predefined endianness allow OSC messages to be mapped unambiguously to C structures, C++, Java classes or JavaScript objects.

## 4. OSC OBJECT CONSTRUCTION AND DISPATCH

### 4.1. Introduction

In prototype-based object-oriented programming, objects are created from scratch (ex nihilo), or by cloning [23] and possibly modifying an existing prototype object. We implement cloning by allocating new memory and duplicating the contents of an OSC message from the prototype object (in the spirit of Kevo [24]). This implementation follows the convention of Max primitive types, is easy to understand, avoids atomicity issues and allows programs to be easily distributed to multiple processors without the cost of managing references [25]. For most OSC messages memory is allocated on the call stack of Max object outlet functions so the cost to allocate and free memory is trivial and known in advance.

### 4.2. Terminology

For the rest of this paper objects built from OSC messages will be referred to as "bundles" and each member of the "o." library will be referred to as a "method". This avoids confusions that might result from the unconventional claim Max/MSP makes on the terms "object," and "message". These are respectively function and array in conventional (and ISO standard) parlance. The spirit of using the terms "bundle" and "method" is to help make this new dynamic object-programming style accessible to both novice and experienced Max/MSP programmers without being burdened with the considerable baggage that comes with the term "object".

Note that OSC methods are signalled with the name prefix "o." as a simple way of establishing a clean name space.

### 4.3. Ex nihilo object creation

The o.message method has the same user interface and input syntax as the Max/MSP message box, providing a constructor for an OSC bundle from a textual description. Just as with the Max message box, Max's list constructor, a simple list selection template operator is invoked using the “$1, $2,…$n” notation to describe how selected values are to be placed in the bundle.

So the classic “hello world” program is implemented as in figure 1.

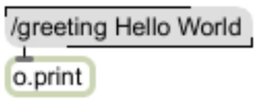


**Figure 1**. Hello World.

Figure 2 better illustrates the dynamic list element substitution familiar to message box users:

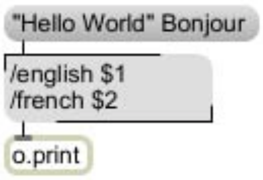


**Figure 2**. Bilingual Hello World

Figure 3 reflects a more interesting media programming scenario

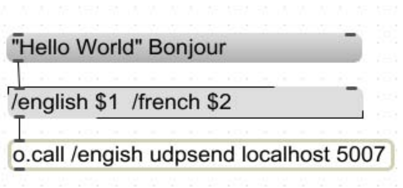


**Figure 3**. Remote Hello World

The last method in the chain results in the text associated with the /english address being sent as a udp packet to the 5007 port on the localhost server. The compact, expressive power illustrated in this example is an important feature of the ``o." library that will be more fully explained in section 5.1 after the requisite scaffolding has been described.

### 4.4. o.build

The o.build method collects values from its inlets and binds them to the given OSC addresses in the spirit of Max's pack object. It outputs the bundle on receipt of the bang message or a value in its first inlet. Default initial values may be provided after each address name in the argument list as shown in Figure 4.

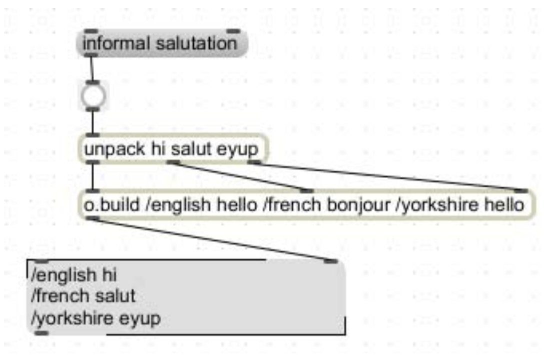


**Figure 4**. Building OSC Bundle associating values to addresses.

Note the use of the second inlet of an o.message box to view the contents of a bundle directly within a patch.

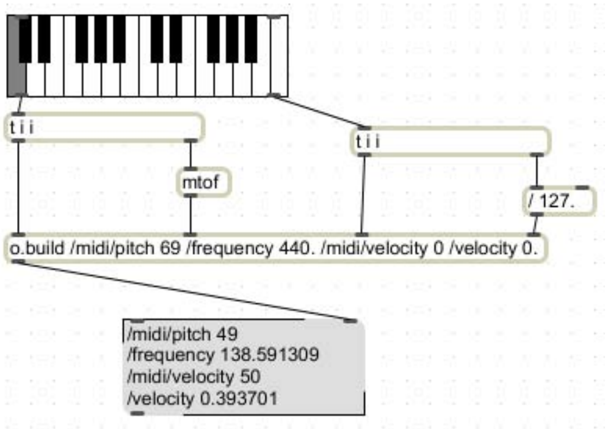


**Figure 5**. Aggregating values into OSC bundles.

Figure 5 shows how o.build can be used to create an interface to the Max keyboard slider that captures both legacy representations of a music keyboard depression as well as more contemporary ones. Note that bundling the data from each keyboard slider outlet better reflects the atomic binding of the two values implied by the gesture that created them than sending them as separate datum at different times out of separate outlets. This common use of the o.build method is analogous to the “named associations” style of Ada function call arguments [26]. Instead of having to direct the right parameters at the right time to the appropriate inlet, parameters are named and bound together into a single bundle. The advantages of this alternative to the positional association style of Max/MSP are well demonstrated in Jamoma [27] which adopts the convention that OSC bundles are sent to the first inlet of Jamoma modules.

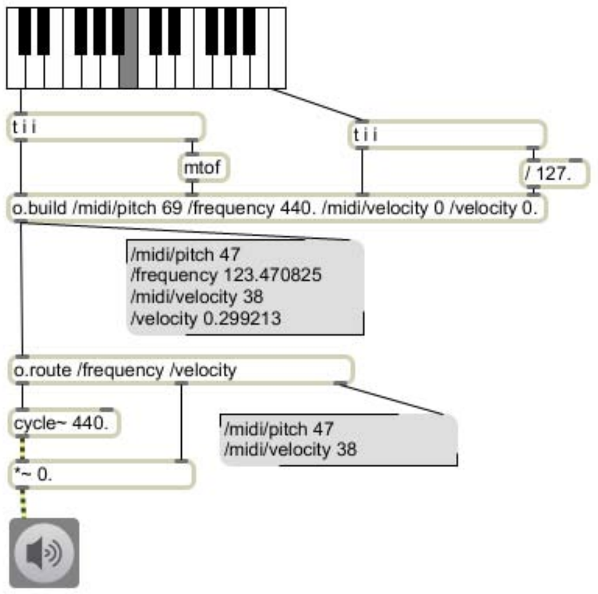


**Figure 6**. OSC bundle from keyboard to synthesizer.

Figure 6 illustrates this bussing scheme and how o.build’s complementary method o.route can be used to bring values out of the OSC bundle into the Max/MSP message world. Notice that the last outlet of o.route outputs a new bundle containing the unmatched elements of the original bundle. The "remainder" bundle can be further processed as the patch evolves illustrating the key to this delegation style of object-oriented inheritance.

It is useful to contrast this approach with static class-based inheritance. The key difference is that in the delegation style new object types are created dynamically by simply adding new address/value pairs to existing objects. Programmers do not need to consult object definitions or API's to understand objects, their derivatives and promises: they simply look at the data in the objects themselves as they are formed and reformed as they move through the patch.

Section 6 includes further discussion of delegation-style inheritance.

## 5. Making OSC methods from Max patches

Instead of creating a large number of new objects to operate on OSC bundles it is possible to reuse existing Max/MSP externals and patches using function-mapping approaches that are analogous to "map" and "apply" from Lisp. The o.call object instantiates a Max patch internally according to its arguments and then routes named messages from incoming OSC packets to the internal Max patch. Finally it gathers the output into an OSC bundle.

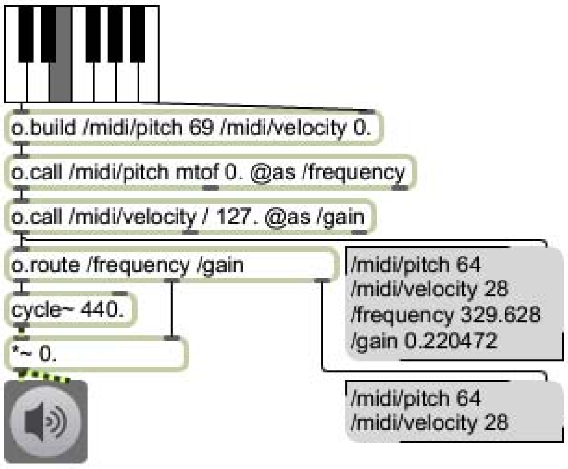
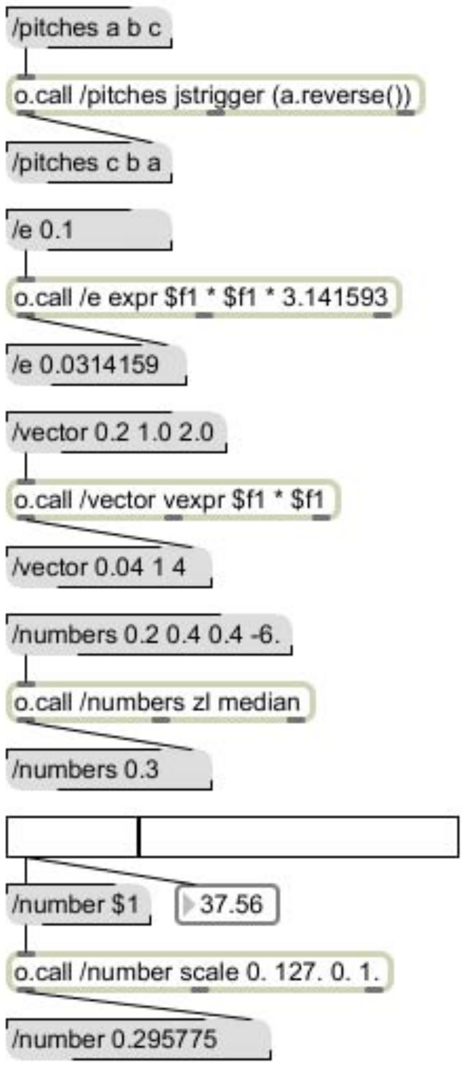


**Figure 7**. Keyboard Synthesizer with o.call data flow.

Figure 7 illustrates o.call implementing a refined version of the keyboard synthesizer patch of Figure 6.

The o.call method uses prefix and suffix operators so it is syntactically closer to Lisp and other functional languages than to C. To clarify subsequent examples we note that the argument list comes first followed by the function description (a Max patch which o.call dynamically instantiates). Finally there may be closing attributes following an @ symbol. These are used to describe what to do with the results of the function call mapping. By default the result is bound to the same name as the first argument pattern. @as is followed by the names to be assigned to new elements that will be added to the incoming bundle. @prepending specifies a prefix to be added to the address of the first argument.

The value to the Max/MSP programmer of this machinery is that only a handful of new "o." methods are needed to bring object-oriented and functional programming into Max/MSP and no changes to the Max/MSP kernel are required. Operations on the values stored in OSC bundles can be done with existing externals and abstractions including all the basic arithmetic and symbolic operations, the zl list objects and JavaScript. Examples are illustrated in Figure 8.

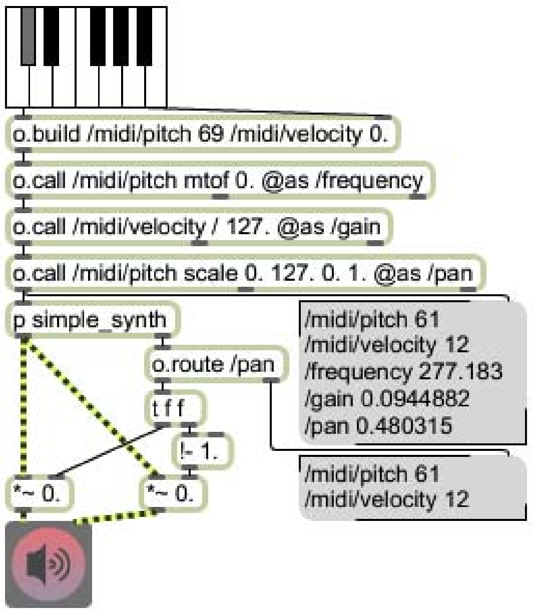


/pitches a b c
o.call /pitches jstrigger (a.reverse())
/pitches c b a
/e 0.1
o.call /e expr $f1 * $f1 * 3.141593
/e 0.0314159
/vector 0.2 1.0 2.0
o.call /vector vexpr $f1 * $f1
/vector 0.04 1 4
/numbers 0.2 0.4 0.4 -6.
o.call /numbers zl median
/numbers 0.3
/number $1
37.56
o.call /number scale 0. 127. 0. 1.
/number 0.295775

**Figure 8**. o.call examples using core Max functions.

## 6. Delegation-style Inheritance

The example in Figure 7 has been augmented in Figure 9 with the feature of pitch-dependent panning to illustrate how delegation can be used to add functionality to programs in a way that promotes reuse (the core benefit of object-oriented programming).



o.build /midi/pitch 69 /midi/velocity 0.
o.call /midi/pitch mtof 0. @as /frequency
o.call /midi/velocity / 127. @as /gain
o.call /midi/pitch scale 0. 127. 0. 1. @as /pan
p simple_synth
o.route /pan
t f f
!- 1.
*~ 0.
*~ 0.
/midi/pitch 61
/midi/velocity 12
/frequency 277.183
/gain 0.0944882
/pan 0.480315
/midi/pitch 61
/midi/velocity 12

**Figure 9**. Delegation-based Inheritance.

The /pan message is computed from the /midi/pitch message and added to the existing bundle. This is analogous to inheritance in class based object-oriented programming. The key to reuse is that neither the original patch assembling the description of the keyboard gesture nor the synthesizer need any changes to support the new /pan parameter. The additional sound functionality is added by using the delegation outlet (conventionally the rightmost) of the synthesizer patch. These new functionalities can of course be encapsulated as required. Note that the /pan route operation delegates its unmatched bundle output inviting future inheritance.

## 7. External Sources and Sinks of OSC Bundles

External sources and sinks of OSC data include the venerable udpsend and udpreceive objects and slipOSC for serial wrapped OSC (typically from USB serial devices).

The new o.io method renders these Max functions obsolete by enumerating and wrapping data from all the core I/O subsystems of OS/X computers as OSC messages. This wrapping function is already partially met with programs such as Osculator (www.osculator.net/) and Glovepie (glovepie.org). Unfortunately there is measurable and potentially troublesome variance in the delay of messages via these programs. The o.io method minimizes these by time tagging the data using the lowest level API to get as close as possible to the actual time the data was acquired. The o.io method already supports core popular protocols HID, UDP, TCP, MIDI, serial and proprietary API's such as the one provided for the built-in laptop motion sensors and multitouch trackpads. The implementation of o.io was carefully designed for extensibility so that new API's and device types can be easily added. Bundles from o.io typically contain the raw data from the API and then one or more overlays of interpreted data according to the device. For example the HID protocol API provides numbered parameter values as the core data stream. The o.io method consults an XML file that describes how to build a symbolic address space from these numbered parameters according to the device and vendor id.

## 8. OSC Bundle Methods

Certain operations on bundles are clumsy to do by breaking them up into Max/MSP native types and reassembling them with o.build. These include merging, unions, intersections and accumulation for which the o.var method is provided. The o.if method is interesting in that it only inspects the contents of a bundle thereby avoid a copy operation as it directs the bundle out of the "true" or "false" outlet according to the evaluation of the conditional expression.

The factorial calculation of figure 10 illustrates o.if in action and how recursion is compactly done with "o." methods.

$$1! = 1, n! = n(n-1)!$$

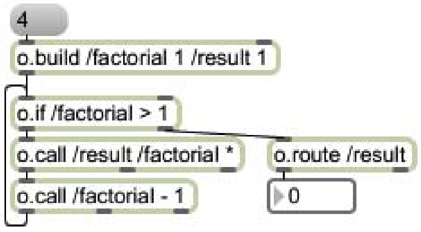


**Figure 10**. Recursion and o.if

The evolving state of this computation is traceable by simply collecting the bundles recursively passed back. The concentration of observable state into bundles turns out to be a very productive programming technique, minimizing bugs that are hard to find because of state hidden within Max externals and patches. OSC time tags can be used in these traces for performance profiling. In the recursive implementation of the factorial function the "o." calls correspond directly to the mathematical definition and all the mathematical operations are explicit. In the iterative version of Figure 11 implicit functionality of the "uzi object" is assumed to be understand as reflected in the different behaviour of each outlet.

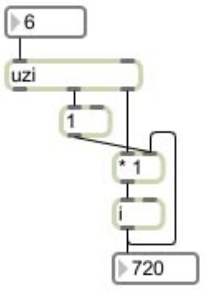


**Figure 11**. Factorial by Iteration

## 9. Music Notation Editor Example

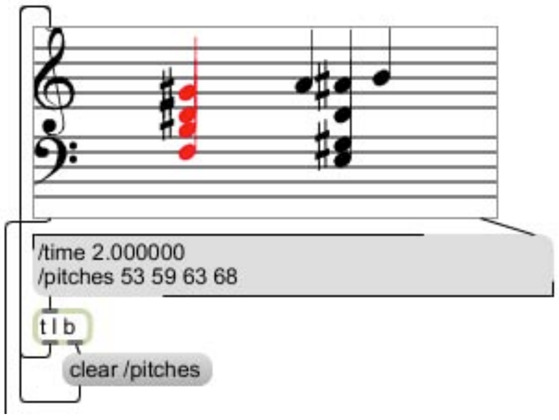
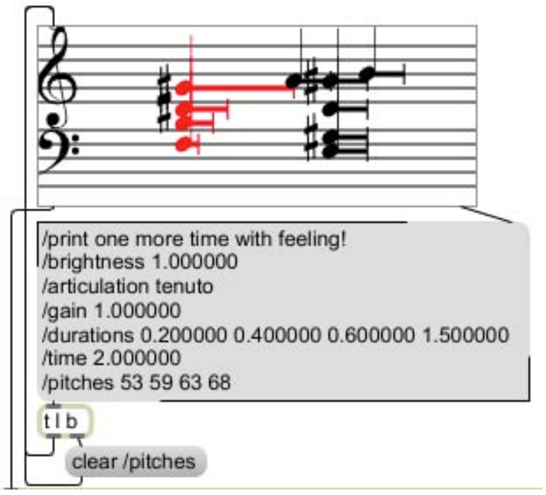


**Figure 12**. o.grandstaff

Figure 12 shows a musical notation editor called o.grandstaff for Max. This example will serve to amplify the earlier points of the paper and place them in a more sophisticated musical context.

With the o.grandstaff visual method notes are entered by pointer and keyboard, or by sending the method OSC bundles containing certain core addresses such as "/pitches" and "/time."

The use of OSC bundles allows for the storage of any additional data alongside the core addresses for which the object has methods to interpret. This extensibility allows the composer to build personal elaborations of notation dynamically.



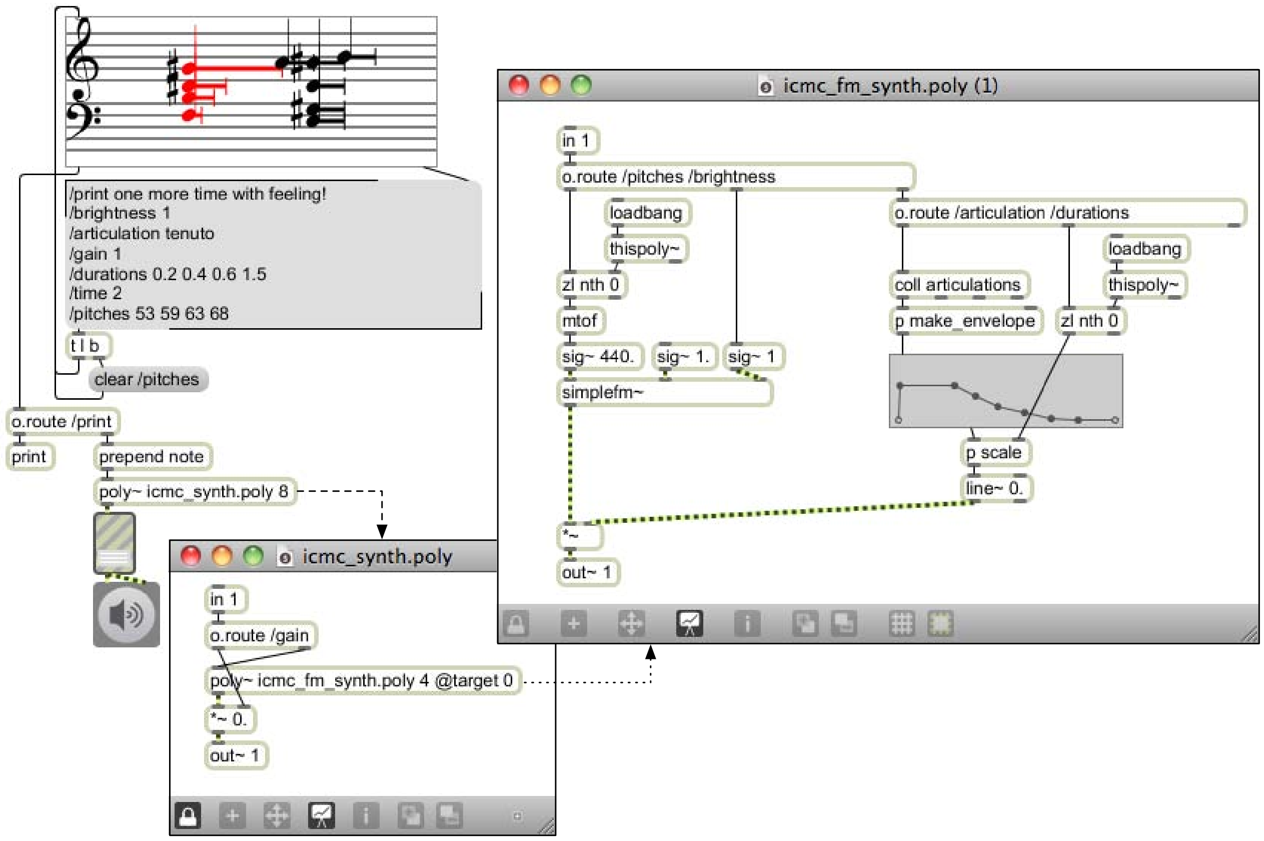
```
/print one more time with feeling!
/brightness 1.000000
/articulation tenuto
/gain 1.000000
/durations 0.200000 0.400000 0.600000 1.500000
/time 2.000000
/pitches 53 59 63 68
```

**Figure 13:** Bundle with extra notation

In figure 13, for example, commands destined for the synthesizer have been added to the bundle along with a message to be printed in the Max window.

Although the synthesizer attached to the patch of Figure 13 may not yet be capable of implementing the high-level description of an envelope such as "/attack tenuto," the message may still be added to the bundle to record the composer's intention. Until the synthesizer is elaborated the extra information will be stored but ignored.

Sound synthesis and processing algorithms often require care in the order in which parameters are updated. Since o.grandstaff outputs OSC bundles, all control



```
/print one more time with feeling!
/brightness 1
/articulation tenuto
/gain 1
/durations 0.2 0.4 0.6 1.5
/time 2
/pitches 53 59 63 68
```

**Figure 14.** o.grandstaff with Synthesizer parameters are delivered atomically to o.route which dispatches them according to the order of its arguments (right to left, following Max programming convention).

The ability to associate the state of the instrument (synthesizer) that will play a note with the pitch to be played is closely related to traditional music notation in which timbral and dynamic information is placed near a given note. In figure 15 the lowest note of the first chord in figure 14 is shown in music notation.

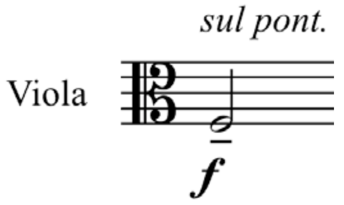


**Figure 15**. Sul pont. notation

The OSC message "/gain" is encoded as the forte dynamic, "/articulation tenuto" appears as a tenuto mark below the notehead, and the "/brightness" message is analogous to the sul ponticello indication.

Note that no intermediate representation (such as MIDI sequencer piano roll) is required or desired in this notation/synthesis system where the two components are coevolved in the course of each composition.

## 10. Conclusion

With the “o.” library OSC messages are more than simply a new aggregate type for Max/MSP. They represent the glue necessary to integrate modern functional and object-oriented programming styles into a visual, dataflow language. Furthermore the time tags, atomicity and ordering semantics of OSC bundles promote productive development necessary for reactive media programming.

## 11. Dedication to Max Mathews

We dedicate this paper to the memory of Max Mathews who started us all out on computer languages for music and who mentored and inspired three generations of exciting work.

## 12. Acknowledgements

We gratefully acknowledge the support and encouragement of the McEnerney Endowment, Meyer Sound Laboratories, Pixar/Disney, the Concordia University Faculty of Fine Arts and the Concordia Specialized Individual Program (SIP) for interdisciplinary study.

## 13. Bibliography

[1] Wright, M. and Freed, A. *Open Sound Control: A New Protocol for Communicating with Sound Synthesizers*. International Computer Music Association, City, 1997.
[2] Wright, M., Freed, A., Lee, A., Madden, T. and Momeni, A. *Managing Complexity with Explicit Mapping of Gestures to Sound Control with OSC*. International Computer Music Association, City, 2001.
[3] Sutherland, W. R. *The On-Line Graphical Specification of Computer Procedures*. Massachusetts Institute of Technology., 1966.
[4] Wright, M., Wessel, D. and Freed, A. *New Musical Control Structures from Standard Gestural Controllers*. International Computer Music Association, 1997.
[5] Smith, W. Class-based NewtonScript programming. *PIE Developers*1994).
[6] Ungar, D. and Smith, R. *Self: The power of simplicity*. ACM, City, 1987.
[7] Reynolds, J. Definitional interpreters for higher-order programming languages. *Higher-order and symbolic computation*, 11, 4 1998), 363-397.
[8] Johnsson, T. *Lambda lifting: transforming programs to recursive equations*. Springer, City, 1985.
[9] Freed, A. M., J.; Schmeder, A. Composability for Musical Gesture Signal Processing using new OSC-based Object and Functional Programming Extensions to Max/MSP. In *Proceedings of the NIME 2011* (OSLO, 2011)
[10] Magnusson, T. and Hurtado, E. The Phenomenology of Musical Instruments: A Survey. *eContact!* , 10.42008).
[11] Magnusson, T. and Mendieta, E. *The acoustic, the digital and the body: A survey on musical instruments*. ACM, City, 2007.
[12] Xiong, Y. and Lee, E. *An Extensible Type System for Component-Based Design*. Springer Berlin / Heidelberg, City, 2000.
[13] Lee, E. and Xiong, Y. *System-Level Types for Component-Based Design*. Springer Berlin / Heidelberg, City, 2001.
[14] Lee, E. A. and Xiong, Y. A behavioral type system and its application in Ptolemy II. *Formal Aspects of Computing*, 16, 3 2004), 210-237.
[15] Schnell, N., Borghesi, R., Schwarz, D., Bevilacqua, F. and M¸ller, R. *FTMóComplex data structures for Max*. Citeseer, City, 2005.
[16] Wakefield, G. and Smith, W. *Using lua for audiovisual composition*. City, 2007.
[17] Hansen, L. Evolutionary Programming and Gradual Typing in ECMAScript 4 (Tutorial).
[18] McKeehan, J. and Rhodes, N. *Programming for the Newton: software development with NewtonScript*. Academic Press Professional, Inc. San Diego, CA, USA, 1995.
[19] Smith, W. SELF and the Origins of NewtonScript. *PIE Developers magazine, July*1994).
[20] Smith, R. B. Prototype-based languages (panel): object lessons from class-free programming. In *Proceedings of the Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications* (Portland, Oregon, United States, 1994). ACM.
[21] Noble, J., Taivalsaari, A. and Moore, I. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer, 1999.
[22] Dony, C., Malenfant, J. and Bardou, D. Classifying Prototype-based Programming Languages. *Prototype-based Programming: Concepts, Languages and Applications, (*1998).

[23] Taivalsaari, A. Delegation versus concatenation or cloning is inheritance too. *SIGPLAN OOPS Mess.*, 6, 3 1995), 20-49.
[24] Taivalsaari, A. Kevo, a prototype-based object-oriented language based on concatenation and module operations. *Report LACIR*, 92-02.
[25] Levanoni, Y. and Petrank, E. *An on-the-fly reference counting garbage collector for Java*. ACM, City, 2001.
[26] Taft, S. and Duff, R. *Ada 95 reference manual: language and standard libraries: international standard ISO/IEC 8652: 1995 (E)*. Springer Verlag, 1997.
[27] Place, T. and Lossius, T. *Jamoma: A modular standard for structuring patches in Max*. City, 2006.