Agile Interface Development using OSC Expressions and Process Migration

John MacCallum
CNMAT - UC Berkeley
Dept. of Music
1750 Arch Street
Berkeley, CA 94709
john@cnmat.berkeley.edu

Adrian Freed
CNMAT - UC Berkeley
Dept. of Music
1750 Arch Street
Berkeley, CA 94709
adrian@cnmat.berkeley.edu

David Wessel
CNMAT - UC Berkeley
Dept. of Music
1750 Arch Street
Berkeley, CA 94709
wessel@cnmat.berkeley.edu

ABSTRACT
This paper introduces "o.expr," an expression language for dynamic, object- and agent-oriented computation of gesture signal processing workflows using OSC bundles. The use of o.expr is shown for a range of gesture processing tasks. Aspects of o.expr, including statelessness and homoiconicity, simplify agile applications development and provide support for heterogeneous computational networks.

Keywords
Gesture Signal Processing, Open Sound Control, Functional Programming, Homoiconicity, Process Migration.

1. INTRODUCTION
We introduce a new tool "o.expr" for functional programming of gesture signal processing algorithms as the main workhorse of the "o." toolkit [5]. o.expr evaluates C-like expressions that contain OSC addresses as variable names. Earlier versions of "o." were embedded in the Max/MSP/Jitter language and relied on this host language to provide the computational heavy lifting using the o.callpatch primitive [5]. o.expr removes this dependency on a particular programming language and enables efficient implementation of gesture signal processing workflows as composable transformations of OSC messages into new OSC messages [16].

"o.expr" contributes the following to gesture signal processing applications:

- increased reliability and legibility by supporting a stateless functional programming style,
- support for a different models of gesture signals via OSC time tags including: band-limited isochronous sampling, Address Event Representations (AER) [4] and compressed sensing,
- dynamic binding of gesture signal processing algorithms to gestural data allowing processing to be delegated to the most efficient node in a computational network via safe, sandboxed process migration, and
- use of self description to minimize stateful registries, discovery protocols and the need for third-party "calls home".

2. PRIOR WORK
Our work is part of a recrudescence of extensive work started in the 1980s on User Interface Management Systems [2, 14]. Attenuation of development of these early experimental systems arose when computers became more affordable and spread from specialized academic laboratories to the office and home. The workstation and personal computer industry normalized device choice and user interfaces, integrating them into proprietary operating system API's. Device communication protocols were also normalized into vendor-controlled protocols such as USB-HID and MIDI. These protocols did not evolve fast enough to accommodate new device technologies and user applications areas such as gaming and interactive installations, growing niches that are now occupied by both vendor- and user-controlled protocols and encodings such as OSC and TUIO.

Most of the specialty languages that were developed as part of the early UIMS's were compiled and much of the work focused on mapping problems between well-established gestures in GUI's and fixed application functions. One interesting experimental language from this period, Squeak [3] managed concurrency in multiple input devices. A more recent representative device-specific language is Proton++ for multitouch gesture mapping [6]. Our work complements these languages by providing the computational machinery to move from sensed input to reliable, calibrated parametric tracking of gesture parameters and feature detection.

Various parameter mapping systems have been proposed for music controller applications [10] based on efficiently encoding a few commonly used mapping strategies. The o.expr language we present takes a more general approach to mapping and makes the functional mappings explicit and accessible to users.

A striking trend in recent NIME projects is an increase in the number of processing components that handle data flowing from sensor to sound output. The following scenario is not unusual: a single chip 9DOF IMU connected to an ARM-based Arduino (Teensy 3), connected via USB to a smartphone that is wirelessly connected to a laptop computer running sound synthesis software. A popular single chip IMU from Invensense actually contains an ARM processor to perform sensor fusion computations. This means that sensor data passes through three ARM processors before sound computations on a final multicore processor. This is typical of a general trend towards rich, complex networks of heterogeneous computation now studied under the rubrics of Cyber Physical systems [7] and material computation [13]. The heterogeneity of these computing elements increases development cost and time because of the number of different development tools that are in play. One early attempt to address this [1] uses a single powerful FPGA development tool for a range of computational structures. Another approach [12] embeds a flexible executive in the sensor/actuator controllers so that development can be unified using OSC messages. These approaches work well within an institution that can impose a particular development discipline. The approach introduced here has greater potential for larger collaborative projects between individuals and institutions by reducing dependences on the capabilities of particular computational nodes.

3. PAPER STRUCTURE

We begin by outlining the features of o.expr's C-like language. We will then present o.expr in the context of several, compact gesture signal-processing examples typical of NIME applications. We conclude with examples of how o.expr expressions themselves can be stored, manipulated, and transmitted as OSC data.

4. O.EXPR SYNTAX AND SEMANTICS

4.1 Introduction

Modules of the “o.” library implement a dynamic object-oriented programming model with specialization provided for by cloning instead of delegation or subclassing. OSC bundles serve as objects and are the only native data structure in “o.”. OSC bundles are general enough to represent aggregate data types similar to the “struct”s of C or objects of Javascript where members of the aggregates are named, typed vectors of primitive data: integers, floating point numbers, times and strings.

Cloning avoids the complication and statefulness of references and garbage collection and, while it is a less commonly used method of implementing dynamic objects than, for example, the delegation style used in Javascript, it has been extensively studied and has a long history of use [8, 9] [15].

Because of the simple semantics of cloning, the basic computational model of each “o.” module is that they emit modified copies of incoming OSC bundles. We will now focus on the most important “o.” module, “o.expr” which provides a rich expression language for specification of computations with OSC messages.

4.2 OSC Addresses as Arguments

Expressions in o.expr define the order of evaluation of nested primitive functions and also name the sources of argument values for these functions using OSC address names. At evaluation time the incoming OSC bundle is copied to the working bundle, and the nested functions are computed using operand values referenced by address name from the working bundle. Assignment functions bind their operands to addresses in the working bundle. The last step of o.expr is to output the working bundle. For example, in the following expression

/foo = /bar + 10

10 will be added to the value (or list of values) addressed as /bar in the working OSC bundle. The result will be bound to the address /foo. If /foo already had a value defined in the working bundle, that value will be replaced by the results of the expression.

4.3 Scalar and Vector Operations

In the case where an address is bound to a list of values (a vector), o.expr will apply a given function to every member of the list. When more than one vector argument is found, the function will be applied to the first *n* items of all lists, where *n* is the length of the shortest list. Finally, in the case of a mixture of vectors and scalars, scalars are promoted to lists containing *n* copies of the scalar value.

Elements of a list may be retrieved using a special double-bracket notation: /foo = /bar[[10]]

In the above example, the 11th element of /bar (counting from 0) will be assigned to /foo. Multiple indexes may be accessed using comma-separated values: /foo[[1, 2, 3]], or by using an address bound to an integer or a list as the parameter: /foo[[/bar]].

4.4 Intrinsic Functions and Constants

Constants are implemented as intrinsic functions with no arguments, i.e., *pi()*, and *e()*.

In addition to the standard arithmetic and logical operators as well as the majority of the functions declared in the C library’s math.h file, o.expr provides useful built-in functions such as *cumsum()* (cumulative sum), *sum(), product(), dot()* (dot product), *sign()* (the sign of its argument), *clip(), scale(), mtof()* (MIDI to Hz), *ftom()* (Hz to MIDI), *reverse(), sort(), nfill()* (create a list of *n* copies of a value), *aseq()* (create an arithmetic sequence), *interleave(), length()* (the number elements bound to an address), *mean(), median(), l2norm(), range()* (the difference of the maximum and minimum of a list), *extrema()* (the minimum and maximum of a list).

Also provided are predicates such as *bound()* that check to see if an address is present in a bundle, *exists()* that returns true if the address is present in the bundle regardless of whether it has data bound to it, and *emptybundle()* that returns true if the bundle contains no messages and false otherwise.

A list of the addresses present in the bundle can be created with *getaddresses(),* a list of typetags present in a message with *typetags()*, and the number of messages present in the bundle is returned by *getmsgcount()*.

Finally, type casting may be achieved through a number of functions such as *float32(), uint64(), char(), string(), etc.*

4.5 List Construction

Lists may be constructed using the *list()* function, or by placing comma-separated values in single square brackets:

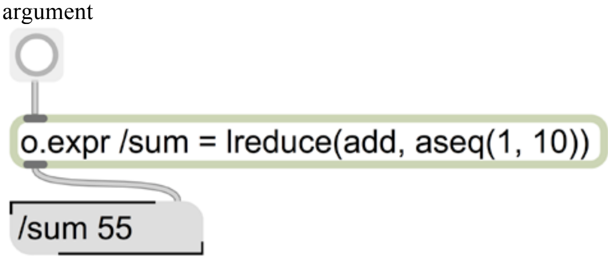
/foo = [1, 2, 3]

When an address is encountered as an element within the square brackets, it will be expanded, e.g., consider the message /foo 1 2 3. The expression /bar = [a b /foo c] will result in the message /bar a b 1 2 3 c.

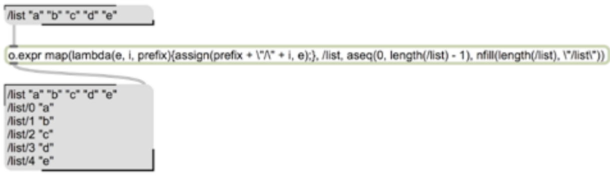
5. Functions, Variables, and Statelessness

5.1 Anonymous + Higher Order Functions

o.expr provides intrinsic higher-order functions such as *apply(), map(), lreduce()* (left-reduce), *rreduce()* (right-reduce), all of which take a function as their first argument. Summing a list of numbers, for example, can be accomplished using *lreduce()* and the functional form of the addition operator as the first argument



In addition to intrinsic functions, these higher-order functions support anonymous (lambda) functions in a style similar to Lisp and Python. In the following example, we take a list of data bound to the address /list and map an anonymous function onto it that will assign each element to a unique address.

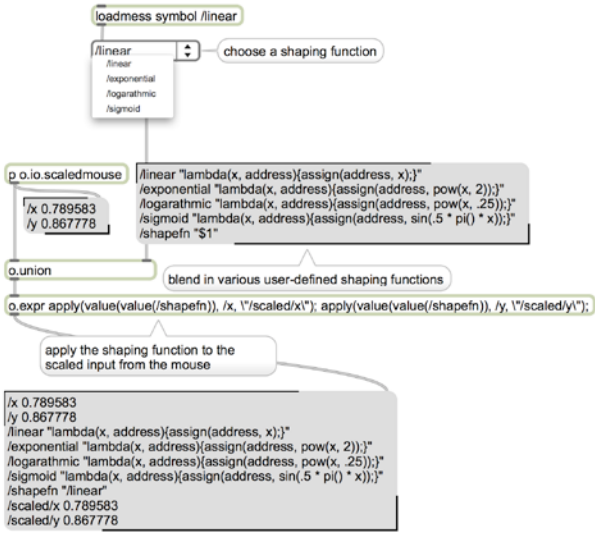


This expression uses the functional form of the assignment operator *assign()* which allows us to specify the assignment target as the result of an expression. The opposite may be accomplished in a similar fashion by calling *getaddresses()* to get the list of addresses present in the bundle and mapping over them.

The *getaddresses()* intrinsic function returns a list of strings and the *values()s* function which is mapped onto that list takes a string and treats it like an address, returning the data bound to it if it exists in the bundle.

5.2 Named Functions

We can create named function definitions simply by binding strings representing anonymous functions to addresses. In the following example, we define four shaping functions as strings bound to descriptive addresses (/linear, /exponential, /logarathmic, and /sigmoid) and assign one of those addresses to an address called /shapefn which represents the function to be applied to our data. We then blend those definitions in to the stream of data coming from o.io.scaledmouse, effectively creating a closure. The chosen function is then applied to the mouse data.



5.3 Unbound Addresses

Since OSC data is not known at the time the expression in o.expr parsed, it is possible that an address in the expression will not be found at evaluation time, or that the address could be contained in the bundle, but not associated with any data. In the current implementation, the execution of the expression is halted and the input bundle is copied through unaltered. Thanks to the helpful suggestion of Sha Xin Wei, we are exploring the possibility of changing this behavior to propagate the part of the expression that can't be evaluated, i.e. adopting the lazy evaluation model. Such an approach can be found in Mathematica.

We also provide a null-coalescing operator similar to that found in C#: /foo = /bar ?? 10 which means "/foo is assigned the value of /bar if /bar exists, and 10 otherwise."

5.4 Statelessness

An important feature of o.expr is that it is stateless—any state necessary to evaluate an expression, must be contained in the bundle that is sent to the instance of o.expr that will perform the evaluation. No state is retained for use by a computation on subsequent OSC bundles.

This property is valuable for the following reasons:

- The bundle can be sent to any environment that implements o.expr and the results are completely dependent on the contents of the OSC bundle, i.e., there can never be a situation where the user sends a bundle to an instance of o.expr and the results will be unknown because some state of o.expr is unknown.
- One can record a series of OSC bundles, either bundles that would be sent to o.expr, or bundles that have been processed by o.expr, and interpret the contents without having to know what state the object was (or would have been in) when the bundle was processed.
- Regression and unit testing of o.expr functions is simplified because only input/output pair comparison is needed

6. GESTURE SIGNAL PROCESSING EXAMPLES

The following example snippets feature the Max/MSP/Jitter implementation of "o.expr". To follow the narrative the most important thing to know about Max/MSP/Jitter itself is that OSC bundles flow from outlets at the bottom of boxes into the inlets at the top, i.e. the usual arrows found in dataflow diagrams are omitted and messages "fall" from top to bottom on the page.

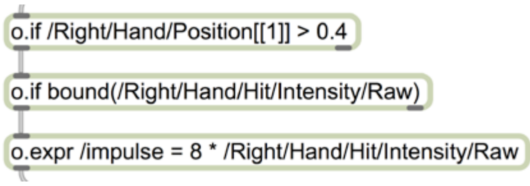
6.1 Definition

Before we start looking at the examples it is useful to consider what we mean by "gesture signal processing". We use the term "signal" in its mathematical sense of "function of time," and "processing" to both capture the notion of computation and evoke related workflows such as Digital Signal Processing or Image Processing.

It is challenging to arrive at consensus on what a gesture is or how to rigorously define the term "gesture." We will sidestep this issue in a productive way by using an operational definition: gesture is the outcome of these signal-processing computations. An important result of this decision and an important contribution of this paper beyond the o.expr tool itself is that we embrace both parametric estimation of continuous gesture (e.g. their prosody) and the recognition and identification and classification of gesture where "signal" is interpreted as a stream of discrete signs, i.e. as semiosis. We will see both senses of gesture represented in the examples.

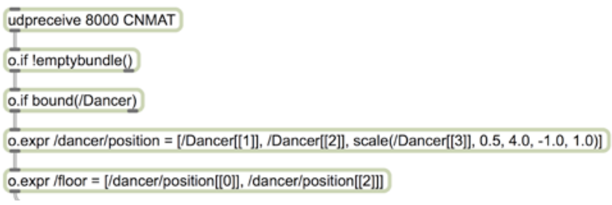
6.2 Feature Filtering

This pipeline processes the OSC bundles that represent the state of a dancer's body as seen from a Kinect and represented by skeletonization machine vision software.



The first step selects OSC bundles that represent the dancer's right hand being above a certain height with respect to their body centroid. The second line selects bundles containing hit events that were identified earlier in the pipeline using feature detection on low velocities. The final step scales an estimate of hit intensity and assigns it to a new name "/impulse" which will inject energy into a resonator bank when received by the OSC-wrapped sound synthesizer later in the chain.

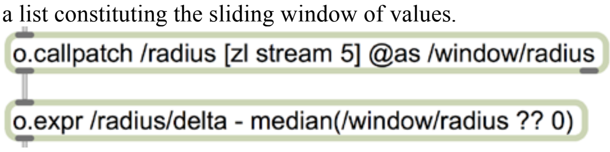
6.3 Coordinate transformation and filtering



The first two predicates check for valid data from the skeletonizer and whether there is a dancer in its field of view. A new parameterization of the space is created that transforms the depth data into a unit interval (-1,1) form to be consistent with the other axes and also finesse the impact of options now available to change the size of this viewport with lenses for the Kinect.

6.4 Managing Statefulness

This component of a novelty detector works by computing the change in current value of a parameter (a radial distance) with the running median of prior values. Since o.expr is stateless, a mechanism of the host language (zl stream) is used to compute a list constituting the sliding window of values.



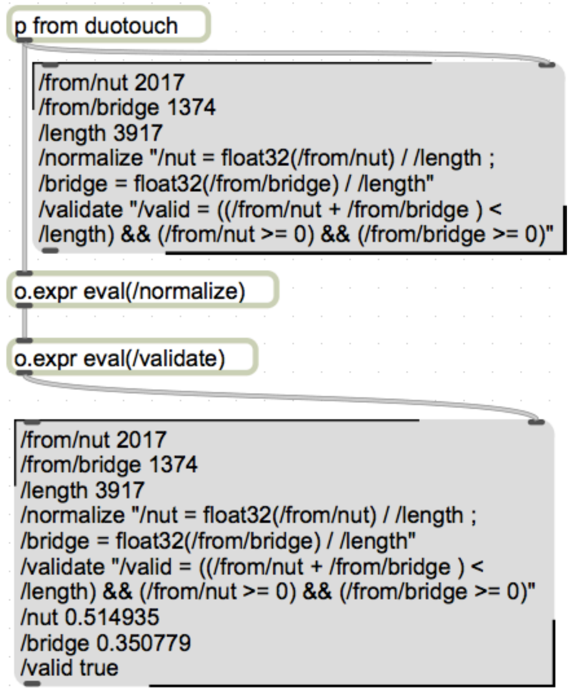
7. DEFERRING AND DELEGATING COMPUTATION

In addition to the usual design constraints of computational performance and algorithm choice, NIME applications involving multiple, specialized processors require attention to the geography and topology of the computations themselves. The "o." dynamic programming model provides an agent-oriented approach to addressing these questions. Of particular value is the separation in location of the description of a computation and its execution. This is evident with the workflow and requirements of calibration. Calibration data is best colocated physically with the sensor so that they move together and devices can be seamlessly moved to other computational nodes. Otherwise elaborate discovery and registry schemes have to be developed to uniquely identify the sensor so that the correct calibration data can be attributed to raw data streams. One advantage however of the latter approach is that it can more easily support new calibration strategies and repurposing developed after the initial deployment of the sensor. This is not an unusual situation and has been observed in commercial applications such as the Wii Remote and academic explorations [11].

Since o.expr expressions themselves can be stored as strings of text or tokenized lists, they can be associated with addresses and stored in OSC bundles. This property, known as homoiconicity, allows "o." objects such as "o.expr" to modify, create and execute o.expr expressions.

Using this property both calibration data and a description of the calibration computations can be stored with the sensor and sent with the measurands to be used later down the computation chain where a sufficiently fast processor will be available. In addition to calibration computations, the sensor subsystem can include predicates for sensor validation and characterization to be computed on demand down stream.

We illustrate this with an example from a fingerboard controller that uses the duotouch technique to sense the position of touch at two points along a linear resistive strip:



The microcontroller computes the length of the strip dynamically using current-steering networks so that length can be expressed ratiometrically minimizing the usual problem of temperature dependence with resistive sensing. The application computations are more easily done with floating point operations normalized in the unit interval but the microcontroller managing the sensing has no native floating point computational units. The OSC bundle it produces therefor contains a description of the normalization computation that is deferred to the o.expr eval(/normalize) operation in Max/MSP.

The validation parameter expresses a basic range constraint and the impossibility that the sum of the lengths of the touch point from each end of the strip could exceed the total length of the strip.

8. CONCLUSION AND FUTURE WORK

As we move large gesture signal processing applications from Max/MSP/Jitter patches into o.expr we are observing that "o." implementations are simpler more concise and easier to understand, due in part to the lack of hidden state which plagues complex Max patches. This comes from the selfdocumenting nature of OSC messages and the thorough implementation of list processing in o.expr. We have also had positive experiences teaching gesture signal processing with o.expr.

We are actively supporting integration of “o.” into new host programming environments such as PD, Processing and Python. We have recently added new functions to o.expr to support efficient computations with time stamps rendered sufficiently generally to support good time engineering practice as reflected in IEEE1588.

9. ACKNOWLEDGMENTS

We would like to thank Jeff Lubow and Rama Gottfried for their extensive testing and exploration of “o.” This work was supported by Meyer Sound, Pixar/Disney, and the Canada GRAND project and by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

10. REFERENCES

[1] Avizienis, R., Freed, A., Suzuki, T. and Wessel, D. Scalable Connectivity Processor for Computer Music Performance Systems. *International Computer Music Conference*. 523-526, 2000.

[2] Buxton, W., Lamb, M.R., Sherman, D. and Smith, K.C. Towards a comprehensive user interface management system. *SIGGRAPH Comput. Graph.*, *17* (3). 35-42, 1983.

[3] Cardelli, L. and Pike, R., Squeak: a language for communicating with mice. in, (1985), ACM, 199-204.

[4] Culurciello, E., Etienne-Cummings, R. and Boahen, K. Arbitrated address-event representation digital image sensor. *Electronics Letters*, *37* (24). 1443-1445, 2001.

[5] Freed, A., MacCallum, J. and Schmeder, A. Dynamic, Instance-based, Object-Oriented Programming in Max/MSP using Open Sound Control Message Delegation *ICMC 2011*, ICMA, 2011.

[6] Kin, K., Hartmann, B., DeRose, T. and Agrawala, M. Proton++: A Customizable Declarative Multitouch Framework *UIST 2012*, 2012.

[7] Lee, E.A., Cyber physical systems: Design challenges. in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, (2008), IEEE, 363-369.

[8] McKeehan, J. and Rhodes, N. *Programming for the Newton: software development with NewtonScript*. Academic Press Professional, Inc. San Diego, CA, USA, 1995.

[9] Noble, J., Taivalsaari, A. and Moore, I. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer, 1999.

[10] O'Sullivan, L., Furlong, D. and Boland, F. Introducing CrossMapper: Another Tool for Mapping Musical Control Parameters *NIME 2012*, 2012.

[11] Schmeder, A. and Freed, A. Support Vector Machine Learning for Gesture Signal Estimation with a Piezo-Resistive Fabric Touch Surface *NIME*, Sydney, 2010.

[12] Schmeder, A. and Freed, A. uOSC: The Open Sound Control Reference Platform for Embedded Devices *NIME*, Genova, Italy, 2008.

[13] Sha, X.W., Freed, A. and Naveb, N., Sound Design as Computational Matter. in *SigCHI*, (Paris, 2013).

[14] Sibert, J.L., Hurley, W.D. and Bleser, T.W. An object-oriented user interface management system. *SIGGRAPH Comput. Graph.*, *20* (4). 259-268, 1986.

[15] Taivalsaari, A. Delegation versus concatenation or cloning is inheritance too. *SIGPLAN OOPS Mess.*, *6* (3). 20-49, 1995.

[16] Wright, M. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, *10* (3). 193-200, 2005.