Differentiating Values and Properties in OSC

Addressing Classes with Open Sound Control

Timothy Place
Electrotap
6920 N. Mercier Street
Kansas City, Missouri 64118
USA
tim@electrotap.com

Trond Lossius
BEK - Bergen Center for Electronic Arts
C. Sundtsgt. 55
N-5004 Bergen, Norway
lossius@bek.no

Alexander Refsum Jensenius
University of Oslo
PB 1017 Blindern
N-0315 Oslo, Norway
a.r.jensenius@imv.uio.no

ABSTRACT

An approach for creating structured Open Sound Control (OSC) messages by separating the addressing of node *values* and node *properties* is suggested. This includes a method for querying values and properties. As a result, it is possible to address complex nodes inside of more complex tree structures using an OSC namespace. This is particularly useful for creating flexible communication in modular systems. A prototype implementation is presented and discussed.

Keywords

OSC, namespace, Jamoma, standardization

1. INTRODUCTION

Open Sound Control (OSC)[1] has evolved into the de facto standard in the computer music community for communication in and between controllers and sound engines [9]. OSC is a protocol for transmitting messages where the addressing of nodes is based on a "slash" notation similar to URLs. As such, OSC is focused on standardizing the communication of messages, but there is no prescribed standardization of the namespaces or the structure of these namespaces.

The authors are involved in developing OSC namespaces for the Jamoma[2] project. Jamoma is a modular system for developing high-level modules in the Max/MSP/Jitter environment. It uses OSC for internal and external communication [5]. As Jamoma's modular structures grow more complex, we find the bi-dimensional namespace conventions of OSC to be inadequate for addressing our constructs. OSC standardizes the addressing of a node, but it becomes increasingly unclear what to do once we reach the node. The problem is exacerbated when the node itself implements its own OSC namespace, as is the case with Jamoma.

[1]`http://www.opensoundcontrol.org`
[2]`http://www.jamoma.org`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NIME08, Genova, Italy
Copyright 2008 Copyright remains with the author(s).

The OSC 1.0 specification consider nodes only in terms of their methods: "OSC Methods are the potential destinations of OSC messages received by the OSC server and correspond to each of the points of control that the application makes available. 'Invoking' an OSC method is analogous to a procedure call; it means supplying the method with arguments and causing the method's effect to take place."

This paper extends current OSC concepts by considering nodes to represent classes in an object-oriented sense, rather than simple methods. For the purposes of this discussion, we will be considering only nodes that contain one or more methods and/or properties. Properties provide additional information concerning how the node behaves and responds to methods, e.g. by specifying *how* a parameter interpolates to a new value. A node might or might not have a value. If it does possess a value property, that value may be set directly, as it is considered an implicit property of the node. A node may branch out to additional nodes, as in existing OSC practice.

The paper starts by reviewing various approaches to creating more complex communication using OSC. This is followed by a presentation and discussion of our suggested approach, introducing the use of a colon for separating between values and properties. Finally, a prototype implementation built in Jamoma is presented and discussed.

2. COMPLEX STRUCTURES IN OSC

The original idea of OSC is that it is tree-structured into a hierarchy called the *address space* where each of the nodes has a symbolic name and is a potential destination of OSC messages [8]. In contrast to the static schema of MIDI, the *open* nature of OSC means that the address space is defined and created by the "implementors idea of how these features should be organized" [9, p153].

This open approach has made OSC useful in a broad range of applications, and adaptable to situations not foreseen by its developers [7]. However, this lack of standardization in namespace schemas is also likely a major reason that OSC has not gained more widespread use in commercial software applications.

2.1 OSC Namespace Standardization

A growing expanse of projects, including the research on mapping between controllers and sound engines, require the ability to discover and query namespaces using a known and common syntax [4]. Other projects, such as LIBOSCQS[3] attempt to solve the problem of disparate namespaces by providing a query system and service discovery for applications using the OSC protocol[2][6].

Several projects have undertaken a standardization of OSC messages, and OSC syntax, for different purposes. In actual practice, these various independent efforts at standardizing namespaces incorporate syntactic elements with conflicting meanings as compared to each other. However, there are some commonalities to these efforts and the problems that they try to address.

2.1.1 Querying Nodes

A primary concern in many of these efforts is the ability to query a node for it's value. The Integra project[4] uses a `.get` appended to the node's address Meanwhile JazzMutant's OSC 2.0 Draft Proposal suggests repurposing the reserved `?` to query for the value of a node [3]. We agree that this functionality is needed, and that a standardized way of doing it is essential. However, we propose that users are interested not only in querying the value of the node, but other properties of that node as well.

2.1.2 Specifying Additional Information

The standardization of OSC namespaces for interfacing with VST Plug-ins was suggested in [10], where units may be specified for the value that is being sent to or from a node. In this proposal the units are specified within the namespace. For example, `/low/output` and `/low/dBoutput` are two ways of controlling the same thing (gain) but specified using different units. This is a similar approach as to how we have previously addressed different types of units in Jamoma, e.g. for specifying gain in either MIDI (`/audio/gain/midi`) or dB (`/audio/gain`).
While such an approach may be beneficial in some contexts, we find that a more structured approach could be beneficial in more complex setups. We therefore propose that the units should be specified as a property of that node rather than contaminating the namespace itself.

2.1.3 Augmented Syntax

A review of the myriad of attempts at creating standardized OSC schemas, and standardized means of discovering and querying namespaces, indicates that additional syntax is needed for clarifying function, address, or both when working with a complex OSC system. Integra, JazzMutant, and Jamoma are all examples where additional symbols, such as the colon, have reserved (if different) meanings.

To investigate possible alternatives for sending this structured information, it is useful to observe the fundamentals of other existing methodologies for representing and sending data over a network.

2.2 XML

Extensible Markup Language (XML)[5] is a particularly relevant analogue to OSC. XML defines a means for formatting data, but not the data or anything specific to the dataspace itself [1].

A number of standardized namespaces using XML have gained wide adoption, including Scalable Vector Graphics (SVG)[6], XHTML[7] and SOAP[8]. SOAP is of particular interest because it is designed as a protocol for exchanging structured information.

Using XML, information is encapsulated into elements. These elements form a tree structure analogous to an OSC message, and for the purpose of this discussion we will treat them as such. Using XML elements, one way to represent the preceding audio gain example is thus:

```
<amodule><audio><gain> 0 </gain></audio></amodule>
```

This is clearly more cumbersome to manually type this XML than the equivalent OSC message:

```
/amodule/audio/gain 0
```

It is also more work for the receiving processor to parse, and it uses more bandwidth as well.[9] However, XML elements are able not only to express a value (*content* in xml parlance) between the tags, but also they can provide properties (*attributes*) to the node. For example, we may provide the unit for specifying the gain:

```
<amodule><audio><gain unit='dB'> 0 </gain></audio></amodule>
```

We suggest a model where it is possible to fork an OSC address to access not only the value of the node, but also the properties of that node, much like what is possible in other existing models such as XML[10].

2.3 OSC Nodes as Classes

In the introduction we made reference to the OSC 1.0 Specification, which states an OSC node represents a function call on a server. We propose that OSC nodes may represent more complex classes, and thus require a mechanism to address the members of these classes. A class *member* may be a *property* or a *method*.

In the Figure 1, the OSC namespace from the previous examples is shown across the x-axis. Traversing the OSC namespace is an action making a horizontal traversal across the figure. In the next section, we will suggest a new syntax for traversing this diagram on the y-axis to address the members inside of these nodes.

2.4 Introducing the colon separator

In addition to the ASCII symbols already reserved for specific purposes within the OSC protocol [8], we introduce the colon ":" as a separator between the OSC address of a node and the namespace for accessing the members of the node:

```
<node address> <value>
<node address>:<member address> <value>
```

The former message sets the value of the node just as it would using the existing OSC conventions. This is because a property named *value* is considered to be implicitly addressed if there is no specific member address given. The latter form calls or sets a member of the node. The member itself is addressed using a fully-qualified OSC namespace. Again using gain as an example, we can send two messages:

[3]http://liboscqs.sourceforge.net
[4]http://www.integralive.org
[5]http://www.w3.org/XML/
[6]http://www.w3.org/Graphics/SVG/
[7]http://www.w3.org/TR/xhtml1/
[8]http://www.w3.org/TR/soap/
[9]The Efficient XML Interchange (EXI) Format solves many of these concerns with XML, but at the expense of human readability because it is a binary format. http://www.w3.org/TR/2007/WD-exi-20071219/
[10]A more concise option than XML, albeit with less clarity and interoperability than OSC, is JSON (http://www.json.org)

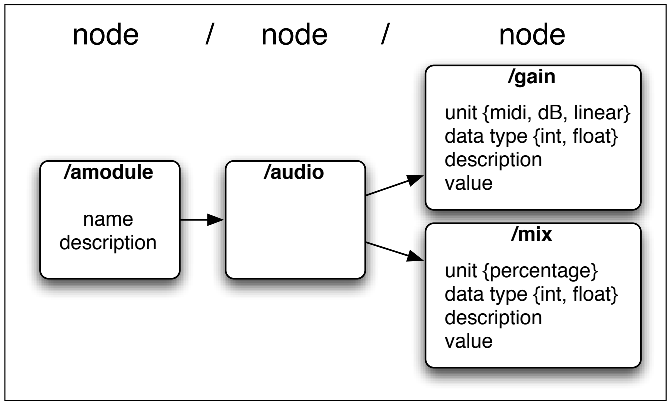

Figure 1: An OSC address tree, with some nodes as classes

one for setting the unit property and one for setting the value.

`/amodule/audio/gain:/unit midi`
`/amodule/audio/gain 120`

Section 3 provides an illustration of the ideas suggested here. In the remainder of the discussion, the address of the node will be omitted for the sake of brevity; e.g. `/computer/module/parameter:/property` will be abbreviated as `:/property`.

2.5 Standardizing Accessors

In addition to using the colon for setting and getting properties of a node in general, we have reserved a limited number of symbols for specific tasks:

- `:/get` returns the value of the node.
- `:/dump` returns the state of the node, which is to say the values of all of the properties including the value itself.
- `:/namespace` returns the namespace implemented at this node.

3. A PROTOTYPE IMPLEMENTATION

The general concepts introduced in the previous section form the basis of the standardized namespace for node properties used in Jamoma. The following serves as an illustration of the concept, demonstrating how it provides the user with extended and structured control of available nodes.

Jamoma distinguishes between the *parameters* and *messages* of a module. Both parameters and messages are addressed as OSC *nodes*. Parameters compose the state of the module, they have a value and can be queried for that value. Messages, on the other hand, are stateless, such as a node that opens a reference file for the module. Apart from the difference in state, both classes of nodes are implemented and behave the same way, and the following discussion is equally valid for both.

3.1 Node type

The type of the node can be specified. Possible types are *none*, *boolean*, *integer*, *float32*, *symbol* and *list*. If one do not want to restrict the type of the node, it can be set to *generic*. The *none* type is only valid for messages. Some of the properties below will only be valid for certain types of nodes. The type property is accessed thus:

```
:/type
:/type/get
```

3.2 Controlling the node itself

In addition to setting the node value directly, it can be set and retrieved as a property. If the node is an integer, float or list type it can also be stepwise increased or decreased. If so the size of the steps is itself a property:

```
:/value
:/value/get
:/inc
:/dec
:/value/stepsize
:/value/stepsize/get
```

3.3 Controlling the range

For integer, float and list nodes a range can be specified. This can be useful for setting up autoscaling mappings from one value to another, or for clipping the output range. The clipping property can be *none*, *low*, *high* or *both*. The range properties are accessed thus:

```
:/range/bound
:/range/bound:/get
:/range/clipmode
:/range/clipmode:/get
```

3.4 Filtering of repetitions

It is sometimes useful to filter repetitions to avoid redundant processing. The *boolean* repetitions property is accessed thus:

```
:/repetitions
:/repetitions/get
```

3.5 Ramping to new values

The ability to smoothly move from one value to another is fundamental to any kind of transition and transformation of musical or artistic material. Jamoma offers the possibility of ramping from the current to a new value in a set amount of time. While the OSC message

`/myComputer/myModule/myParameter 1.0`

will set the parameter value to 1.0 immediately, the message

`/myComputer/myModule/myParameter 1.0 ramp 2000`

will cause the value to ramp to 1.0 over 2000 milliseconds. Ramping in Jamoma works with messages and parameters of type integer, float and list.

Jamoma offers vastly extended possibilities in how ramping can be done as compared to Max. In Jamoma the process of ramping is made up from the combination of two components: A driving mechanism cause calculations of new values at desired intervals during the ramp, while a set of functions offers a set of curves for the ramping. Both components are implemented as C++ APIs, and can easily be extended with new ramp or function *units*, expanding the range of possible ramping modes.

3.5.1 The Jamoma RampLib API

The Jamoma RampLib API provides a means by which to create and use *ramp units* in Jamoma. A ramp unit is a self-contained algorithm interpolates from an existing value to a new value over a specified amount of time using one of several timing mechanisms. Currently four such ramp units are implemented:

- *none* - jumps immediately to the new value. Typically used for values where ramping does not make sense.

- *scheduler* - uses the Max internal clock to generate new values at fixed time intervals.
- *queue* - ramping using the Max queue, updating values whenever the processor has free capacity to do so.
- *async* - only calculate new values when requested to do so. This might be used in video processing modules to calculate fresh values immediately before processing the next video image or matrix.

When a new ramp is started, the ramp unit internally uses a normalized ramping value, increasing linearly from 0.0 to 1.0 over the duration of the ramp. Whenever the ramp unit is to provide a new value, it updates the normalized ramping value, and passes it to a Function Unit as described in Section 3.5.2. The normalized value returned is then scaled to the range defined by the start and end values for the ramp, and the parameter is updated in the module.

3.5.2 The Jamoma FunctionLib API

The Jamoma FunctionLib API provides normalized mappings of values $x \in [0, 1]$ to $y \in [0, 1]$ according to functions $y = f(x)$. Currently five functions are implemented: Linear, cosine, lowpass series, power function and hyperbolic tangent. There are plans to expand this further with w.g. exponential functions.

3.5.3 OSC namespace for ramping properties

Ramping properties are addressed using `:ramp/drive` and `:ramp/function` OSC nameclasses. In addition to the ablity of setting or getting current ramp driving mechanism, it might be useful to have the module return a list of all available ramp units. This can be done by means of a `/dump` message:

```
:/ramp/drive
:/ramp/drive/get
:/ramp/drive:/dump
```

Some ramp units might have additional parameters that can be controlled by the user. For instance the user can control how often the *scheduler* ramp unit is to update; the granularity of the ramp. In general a ramp unit parameter `/foo` might be accessed thus:

```
:/ramp/drive:/foo
:/ramp/drive:/foo/get
```

If the current ramp unit do not have a `/foo` parameter, the message will be ignored by the ramp unit. It is possible to query what parameters are available for the current ramp unit with the `/dump` message:

```
:/ramp/drive:/dump
```

The function used for the ramp can be set or queried in much the same way. It is also possible to request information on available function units with the `/dump` message:

```
:/ramp/function
:/ramp/function/get
:/ramp/function:/dump
```

Some function units might have additional coefficients influencing the shape of the curve, e.g. the exponent of the *power* function unit can be set. In general a function unit parameter `/bar` might be accessed thus:

```
:/ramp/function:/bar
:/ramp/function:/bar/get
```

If the current function unit do not have a `/bar` parameter, the message will be ignored by the function unit. It is possible to query what coefficients are available for the current function unit with the `/dump` message:

```
:/ramp/function:/dump
```

3.6 Controlling the user interface

In certain applications the CPU overhead of continuously updating the graphical user interface whenever parameter or message values change might become a burden, competing for CPU with e.g. video processing algorithms. If the user does not need continuous visual feedback on updated values of parameters or messages, the GUI for the parameter or message can be frozen, freeing up the processor and GPU for tasks considered more important:

```
:/ui/freeze
:/ui/freeze/get
```

A parameter or message that has its GUI frozen can be forced to update and refresh the displayed value once by means of the message:

```
:/ui/refresh
```

3.7 Description

The description property is a string providing a text description of the parameter. In Jamoma this is used for auto-generating online documentation of the modules. It can also be used for building modules that retrieve the total namespace of all Jamoma modules used, and provide interactive documentation of available parameters. The descriptions property is accessed as:

```
:/description
:/description/get
```

4. DISCUSSION AND FURTHER WORK

4.1 DataspaceLib

In addition to the current RampLib and FunctionLib APIs work has started on implementation of a Datasbape Lib, for the purpose of enabling nodes to be address using several different interchangeable measuring units. For example a gain parameter can be set using MIDI, dB or linear amplitude depending on the context and preferences of the user. The DataspaceLib is also meant to offer mapping between more complex interrelated coordinate systems, so that e.g. cartecian and spherical coordinates can be used interchangeably for description of points in space.

Querying - we propose a different system to the JazzMutant OSC2 draft

Expanding :/dump for namespace discovery

In the figure the slash equals stepping in the right direction, to the next branch of the OSC namespace, while the colon indicates stepping downwards in order to access the inner properties of the current node.

Use analogy: what we are doing is really similar to C++ classes.

The proposal set forward in this paper could be cosidered one step in the direction of a more object oriented approach to Open Sound Control

5. ACKNOWLEDGMENTS

The authors would like to thank all Jamoma developers and users for valuable contributions, and iMAL Center for

Digital Cultures and Technology for organizing a workshop where the issues presented in this paper were discussed.

6. REFERENCES

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (fourth edition). Technical report, W3C, September 2006.

[2] M. Habets. OSCQS - schema for opensound control query. System version 0.0.1, 2005.

[3] Jazzmutant. Extension and enchancement of the OSC protocol. Draft 25 July, 2007.

[4] J. Malloch, S. Sinclair, and M. M. Wanderley. From controller to sound: Tools for collaborative development of digital musical instruments. In *Proceedings of the International Computer Music Conference*, Copenhagen, 2007.

[5] T. Place and T. Lossius. Jamoma: A modular standard for structuring patches in max. In *Proceedings of the International Computer Music Conference*, pages 143–146, New Orleans, LA, 2006.

[6] A. W. Schmeder and M. Wright. A query system for Open Sound Control. Draft Proposal, July 2004.

[7] M. Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.

[8] M. Wright and A. Freed. Open sound control: A new protocol for communicating with sound synthesizers. In *Proceedings of the International Computer Music Conference*, pages 101–104, Thessaloniki, 1997.

[9] M. Wright, A. Freed, and A. Momeni. Opensound control: State of the art 2003. In *Proceedings of NIME-03*, Montreal, 2003.

[10] M. Zbyszynski and A. Freed. Control of VST plug-ins using OSC. In *Proceedings of the International Computer Music Conference*, pages 263–266, Barcelona, 2005.