Dynamic Message-Oriented Middleware with Open Sound Control and Odot

John MacCallum[1], Rama Gottfried[1], Ilya Rostovtsev[1], Jean Bresson[2], and Adrian Freed[1]

[1]Center for New Music and Audio Technologies, Department of Music, University of California, Berkeley,
john@cnmat.berkeley.edu, rama.gottfried@berkeley.edu, ilyaforpresident@berkeley.edu,
adrian@cnmat.berkeley.edu
[2]UMR STMS: IRCAM-CNRS-UPMC, Paris,
bresson@ircam.fr

ABSTRACT

We present recent work on odot, a system that extends Open Sound Control and facilitates the rapid and dynamic construction of Message-Oriented Middleware providing an interoperability layer for communication between applications. Unlike traditional middleware systems, odot, when embedded in a host environment, provides a node where computation can take place, allowing middleware to take shape dynamically as the needs of the system develop.

1. INTRODUCTION

In the course of a complex design project we often encounter the need to use multiple applications to satisfy a diverse set of requirements. The work we present in this paper is designed to facilitate the communication and interoperability of a variety of software environments. Imagine constructing a composition for instrumental ensemble with live electronics in which we manage compositional structures in an offline comositional environment like OpenMusic, [1] build a system for real-time electronics using Max/MSP [2] and a score using our favorite music notation editor or vector graphics software. Sharing data between these applications is typically be done by putting it in a file, either by hand or from within the program, using an *ad hoc* format. In many simple cases, this scenario is innocuous enough, however, these file formats can grow in complexity to the point that they become a full fledged language or encoding, albeit one which is often unspecified and undocumented.

The above scenario could be greatly improved if the different applications involved made use of a common data format and we could dynamically construct *middleware* to provide an interoperability layer between applications. In this paper, we propose that Open Sound Control (OSC) is a suitable encoding, and the odot framework provides a set of tools for constructing such a layer in a dynamic and agile fashion.

We begin with brief descriptions of Message-Oriented Middleware, OSC, and odot. We then discuss our ongoing work to provide support for a variety of host and nested host environments that can be made to communicate by passing OSC between them. Finally, we conclude by discussing a number of examples and future work.

[1] http://repmus.ircam.fr/openmusic/home
[2] https://cycling74.com

2. OPEN SOUND CONTROL AND ODOT

2.1 Open Sound Control (OSC)

Open Sound Control[1][2][3] is a popular encoding in the music and media/arts programming communities for moving data between sensors and actuators, as well as processes and threads on the same computer. It is extensible, has support for a rich set of data types, and is relatively lightweight and easy to implement.

OSC data is binary encoded and consists of *messages* that bind *addresses* to data. Multiple messages may be timestamped and transmitted atomically by collecting them in a *bundle*. Figure 1 illustrates the textual representation of an OSC bundle using a JSON [3] -like syntax. The bundle contains two messages, `/a : 1` and `/b : [1, 2, 3]`, and colons separate the addresses, `/a` and `/b`, from the data, the integer `1`, and the list `[1, 2, 3]`.

```
{
    /a :  1,
    /b :  [1, 2, 3]
}
```

Figure 1. A human-readable representation of an OSC bundle.

2.2 libo

At the foundation of our current work is a C library called libo that implements a superset of the OSC 1.0 specification [4] with a set of extended types (see table 1). We refer to bundles

[3] http://json.org
[4] http://opensoundcontrol.org/spec-1_0

Typetag	Description
h	64 bit big-endian two's complement integer
H	64 bit unsigned big-endian two's complement integer
u	16 bit big-endian two's complement integer
U	16 bit unsigned big-endian two's complement integer
c	8 bit big-endian two's complement integer
C	8 bit unsigned big-endian two's complement integer
I	32 bit unsigned big-endian two's complement integer
d	64 bit ("double") IEEE 754 floating point number
t	OSC timetag (64-bit fixed-point NTP)
B	odot bundle
A	Executable code
T	True (no bytes are allocated in the argument data)
F	False (no bytes are allocated in the argument data)
N	Nil (no bytes are allocated in the argument data)
S	Alternate string type represented as an OSC-string

Table 1. Extended type support provided by libo.

that make use of this particular superset of OSC as *odot bundles*. libo is a small, lightweight, sandboxed environment that provides a set of low-level functions for constructing, querying, and transcoding odot bundles. In this section, we describe those features of libo that facilitate its use as a tool for communicating between diverse applications.

2.2.1 Platform Support

libo has support for a growing number of hardware architectures and operating systems. Currently, it has been compiled and tested on x86 (32- and 64-bit), and ARM architectures, and runs under Mac OS X, Windows, Linux (tested on Debian and Ubuntu), Apple's iOS.

2.2.2 Language Bindings

libo is written entirely in C and can be bound into any language that supports a Foreign Function Interface (FFI). Currently, we provide bindings to Javascript (Google V8 [5]), Python, and Common Lisp (tested with LispWorks [6] and SBCL [7]), all done with the Simplified Wrapper and Interface Generator (SWIG). [8]

For each supported language, the entirety of libo is exposed providing developers with access to libo's functionality through a familiar interface.

2.2.3 Transcoding

Transcoding between odot bundles and the human-readable representation of the native data structure of a host language or environment is essential for seamless interaction. Currently, libo provides partial support for transcoding to JSON, SVG, and S-Expressions.

2.2.4 Timetags

An important feature of OSC bundles is the inclusion of a slot for a timestamp in the header. In odot, the recommended practice is to bind any timestamps to addresses that describe their semantics. These timestamps are critical to any distributed system in which causal ordering, latency, and/or jitter are important.

2.2.5 Expression Language

libo provides a parser and evaluator for a lightweight yet powerful expression language that uses odot bundles as its fundamental aggregate data type. Since odot bundles can contain code to be evaluated by this mechanism, each site that hosts libo becomes a locus of computation for a language and data structure shared between diverse applications.

2.3 Odot

Odot[4][5] is a framework for writing dynamic programs in a host environment. It is constructed using the low-level building blocks provided by libo. In implementing odot for different host environments, we promote idiomatic design specific to the host, rather than the implementation of a uniform odot interface. A minimal odot implementation provides abstractions for the *bundle*, *message*, and *atom* constructs, access to the parsing, formatting, and evaluation functions, and functions to convert the odot types listed in table 1 to and from those native to the host environment. Beyond this minimal set of features, an odot implementation may also provide any number of additional features idiomatic to the host environment. For example, the o.route module for Max/MSP and PD provides both conversion to native data types and ordered dispatch of messages typical of a dataflow environment. The design of this module is unique to these types of environments, and the same functionality may take on a very different look and feel in other hosts such as Python or JavaScript.

3. MESSAGE-ORIENTED MIDDLEWARE (MOM)

Message-Oriented Middleware (MOM)[6] is a general term referring to infrastructure that enables message passing between parts of a system (figure 2). MOM emerged to support

Figure 2. Message-Oriented Middleware provides infrastructure to facilitate communication between applications.

[5] https://code.google.com/p/v8/
[6] http://lispworks.com
[7] http://sbcl.org
[8] http://swig.org

the ever-growing scale of distributed systems and is largely in response to the shortcomings of Remote Procedure Calls (RPC)[7]. Where RPC is *a*) invasive, requiring deep coupling with the host software, *b*) synchronous/blocking, which subjects the performance of the system to that of the worst-performing node, *c*) statically constructed, requiring great foresight on the part of the RPC implementor, MOM provides a more dynamic and flexible system of message passing that allows different applications to be developed with minimal consideration for interoperability.

In our work described in this paper, a host system need only provide support for user add-ons via an SDK, or simply a scripting language with a Foreign Function Interface (FFI); odot's expression language evaluator provides a node where computation of code contained in the incoming data structure may take place. We refer to our work as *Dynamic* Message-Oriented Middleware to emphasize the fact that odot only provides a computational node, but the code to be executed may be constructed *dynamically* and on the fly in the context of the data structure.

Conceptually, MOM allows different applications to be thought of as *actors* in the Actor Model of Computation[8][9]. In such a system, the sender and receiver act asynchronously and concurrently. Messages need not *arrive* in a specific order, but a mechanism must be provided to ensure that they will *execute* in the correct order. OSC and odot support timetags and ordered collections of messages allowing for the construction of a model of time known as *superdense time*[10]. Superdense time operationalizes the notion of multiple events occurring *simultaneously* but in an explicit, causal order. All events are timetagged with a pair (t, n) where t is the time, and n is an index. Conceptually, two timetags (t, n_1) and (t, n_2) with the same time but different indexes are considered to occur simultaneously, however, their execution will be ordered such that their causal relationship will be maintained, i.e., (t, n_1) will be executed *before* (t, n_2).

4. HOST ENVIRONMENTS

The low-level language bindings discussed in section 2.2.2 are the foundation for odot implementations in a variety of host environments. Figure 3 shows the relationship between libo, the host environments, and odot. On top of libo sits a layer of shims created automatically by SWIG, or, in the case of Max/MSP and PD, by hand and encapsulated in libraries that rely on those environment's SDKs. The shim layer exposes all functions contained in libo to the host environment. A set of language bindings then abstracts away much of that functionality and provides an idiomatic language-specific interface with which we construct an odot implementation. Here, we discuss the details of a number of odot implementations currently in development.

4.1 Max

In the Max/MSP implementation, all object names are preceded by the prefix `'o.'`. Odot bundles and messages

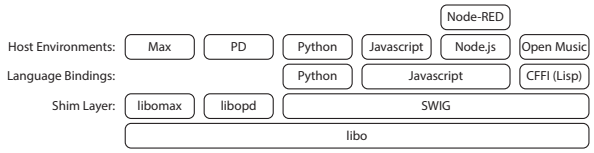

Figure 3. Bindings and host environments.

may be passed between objects by sending the message `FullPacket` followed by two integers specifying the length of the bundle in bytes and the address of the bundle (or message) in memory. The remaining minimal functionality is provided by the following objects:

Bundle FullPacket message
Message FullPacket message
Text Parser `o.compose`
Text Formatter `o.print`, `o.display`
Host to odot Transcoding `o.pack`, `o.collect`
Odot to Host Transcoding `o.route`, `o.atomize`
Interface to the Expression Evaluator `o.expr.codebox`

Below, we provide a number of notes and observations on the implementation.

Implementation Details

Volatility Following the return of the outlet call, the bundle that has been passed is typically freed unless the object has some need to store it for future use.

Bundle Ownership An object receiving a `FullPacket` message from another object is expected to copy it and operate on that copy, i.e., an object must not alter a bundle that it did not create.

Persistent Storage Given the volatile nature of bundles in Max, persistence must be handled by special objects that will make a copy of the bundle passed to them, rather than objects (such as `zl reg`) which copy the Max `FullPacket` message containing a potentially stale memory pointer.

Distributed Computation An important extension to the OSC specification provided by odot is the addition of a data type for representing code in the expression language. This allows code to flow through a network of Max objects and for the decoupling of the what and where of evaluation. For example, one can specify areas in the dataflow graph *where* computation is to take place without specifying *what* will be computed.

Interaction With the Host Environment The odot objects are designed to leverage the host environment where possible. To this end, most odot objects will interpret Max messages that begin with a symbol preceded by a slash (e.g., `/foo 1 2 3`) as an odot message, Further, Max dictionaries containing key value pairs are interpreted as odot bundles.

4.2 PD

Although Max and Pd have aspects of their ancestry in common, they have diverged in a number of important ways, notably, their Graphical User Interfaces (GUIs), and the data types they support. Although flext [9] is an excellent compatibility layer for developing externals that work for Max and Pd, it does not provide a comprehensive set of solutions for our set of issues, and the constraints of its GPL license are prohibitive.

Implementation Details

Pointer Representation In Max, OSC bundles are passed between odot objects using the `FullPacket` message followed by the size of the OSC bundle in bytes and a pointer cast to a long integer. However, the only numeric data type available in Pd is a 32-bit float. To work around this, on 32-bit architectures, we simply reinterpret the bits of the floats, however on 64-bit architectures, we must split the bits of the address into two floats and output a `FullPacket` message with three arguments, rather than two.

Inlet Proxy The Max API provides an inlet proxy that greatly simplifies the handling of object methods across multiple inlets and facilitates the creation of a number of inlets not known at compile time. Since the Pd API does not provide such a utility, we created a proxy class which provides the functionality necessary for our code to run unchanged.

Reserved Characters There are a number of characters that are important in the odot expression language that may not be used anywhere in the GUI such as $\{$, $\}$, and $\backslash$. Every character typed into the GUI is first filtered to remove special characters such as these, which have special meaning to Tcl/Tk, and then passed to the Tcl/Tk evaluator. Further, Pd patches are stored in plain text with a syntax that makes use of certain characters used in the expression language. As a workaround for both of these issues, we simply encode the characters in hexadecimal.

Graphical User Interface Objects The odot library for Max provides a small number of graphical objects for displaying and creating bundles, as well as writing code in the expression language. Since graphical code that paints these objects is unique to the host environment, it was necessary to supply new code for Pd's Tcl/Tk GUI. This was done through the creation of class called `opd_textbox` that handles common interaction and the majority of the object's Tcl/Tk code. When a user clicks on the GUI object, a new text widget is created. To facilitate use of the editing utilities of the text widget (undo, delete, etc.), we switch the UI binding for the mouse and keyboard actions so that these operations affect the text being typed, not the object itself. Once the user finishes editing, the binding is returned to the main Pd, canvas, the text is sent from Tcl/Tk via hex to the C process where it is parsed, reformatted, and sent back to Tcl/Tk for display as plain text within the object canvas. Custom graphics behavior for these objects can be controlled by customizing the `t_widgetbehavior` struct contained in the `opd_textbox` object.

[9] http://grrrr.org/research/software/flext/

4.3 Python

Python is a mature dynamic language with a heterogeneous user-base of various disciplines. SWIG's exposure of libo to Python provides access to the C library calls. To conceal the C interface of the library, we developed a set of Python classes, defining a language-specific interface for working with odot data. Currently, we support creation and manipulation of odot messages and bundles, and provide a dedicated structure for working with time tags.

Odot structures are iterable in Python—individual messages in bundles and individual data members in messages can be accessed using Python-specific iteration syntax. Additionally, we provide efficient transcoding from bundles to Python dictionaries. Since Python is a popular language for server side applications, we also provide transcoding between odot and JavaScript Object Notation (JSON), a lingua-franca data structure of the web.

Our work with Python illuminated our designs for supporting odot as a data structure in textual languages. Since libo was composed specifically in support of visual programming environments, the Python library inspired an additional set of features. Many of these features are applicable to other languages (particularly Java, C++, and JavaScript).

4.4 Javascript, Node.js, and Node-RED

There are many implementations of javascript—Google, Mozilla, Apple and Microsoft all provide competing versions of the dynamic language. Our current focus is to support Google's V8 due to its use in the Node.js server programming environment.

4.4.1 Node.js

Node.js is a platform for building network applications. [10] Currently, Node is a popular platform for server-side software development. At the time of this writing, Node's package management database touts 123,512 total packages and over 42 million daily downloads. [11] These packages cover a rich set of web services and allow contemporary server programmers to integrate new technologies with ease.

Node can be installed on any computer, turning an ordinary laptop into a web node capable of receiving and transmitting network data. Node users write programs in JavaScript. Under the hood, Node unites several technologies. The javascript interpreter is Google V8 (Chrome JavaScript Runtime), written in C++, one of the interpreters targeted by

[10] http://nodejs.org/

[11] https://www.npmjs.com/

SWIG. Input and output is handled with libuv - a C library for non-blocking input and output.

Node contributes an automatic integration of libuv into its environment. Developers work with input and output without worrying about the underlying mechanisms ensuring non-blocking operation (threads). Node provides a set of helpful extensions to JavaScript for server-side applications. For example, Node's Buffer is an efficient solution to dealing with binary data in JavaScript.

C++ developers create addons for Node—dynamically linked shared objects that can expose functions of low-level libraries similar to Max/MSP external objects. [12] Prior to May 2014, we exposed a subset of libo functionality with custom C++ code. As of May 2014, SWIG offers JavaScript as a target, enabling libo integration workflow similar to Python. Once installed, odot functions become available to developers like any other Node module.

4.4.2 Node-RED

Node-RED, a product of IBM Emerging Technology Group, is a graphical programming environment built on Node.js. [13] Node-RED provides an abstraction layer for hardware interactions, web services, and APIs. Node-RED encapsulates various services into graphical objects. The graphical objects enforce a particular standard for sending and interpreting data. Users of Node-RED author complex server-side behaviors by manipulating graphical objects within the web browser editor. Commonly desired functionalities (e.g. obtaining data from an Arduino, interfacing with Twitter APIs) are encapsulated for re-use without writing a single line of code.

4.5 OpenMusic (OM) and Common Lisp

OSC has long been supported in the OpenMusic [11] computer-aided composition environment via the *cl-osc* implementation. [14] In this section, we describe recent work carried out using the Common Lisp bindings to libo. This new strategy improves the efficiency and reliability of the environment with regards to OSC communication, ensuring the support of up-to-date odot features and specs. Nested bundles and the embedded expression language will allow to integrate powerful/expressive data processing in interchange messages and protocols.

OSC messages in OM are represented as simple lists, i.e. with an OSC address followed by a sequence of arguments (float or integer numbers, strings, etc.) The data types of the arguments are therefore implicit and native to the Lisp environment. Accordingly, an OSC bundle is a simple structure or class (*o.bundle*) gathering a list of OSC messages and an optional time-tag. This structure is straightforward to instantiate and process either in Lisp or using graphical programming in the compositional environment.

[12] http://nodejs.org/api/addons.html

[13] http://nodered.org/

[14] https://github.com/hanshuebner/cl-osc

Currently, the intermediate-level functions constituting the Common Lisp odot API in OM permit the following main operations:

Creation of odot bundles from Lisp structures (*o.bundles*). Every message in the *o.bundle* is converted to a C pointer; a serialized bundle is created and returned.

Creation of odot atoms from Lisp data during message processing and conversion. Adequate C constructors are used according to the Common Lisp data types. If a message argument is a nested list itself, the atom is created as a nested bundle.

Decoding odot bundles into *o.bundle* instances, including Lisp-formatted messages.

Reading/setting time-tags in odot bundles : efficient r/w access preventing decoding/encoding in time manipulations.

The *libo* interface is used to generate binary representations from *o.bundles*, stored in foreign-allocated memory. It is also used to read external (serialized) buffers to reconstitute *o.bundles*. These serialized binary representations of OSC bundles are the ones that are sent/received when sending odot data via UDP. They can also be shared or passed as foreign buffers in external applications or libraries (e.g. other C/C++ libraries linked to the Lisp environment).

Due to the cost of foreign memory copy and allocation in Common Lisp, specific strategies must be adopted when such foreign data is manipulated in the OM visual programs.

Depending on the context, we can therefore choose to:

1. make copies in each *odot*-related processing module, which will prevent side effects and keep the system clean and stateless. In this case, as with the Max and PD implementations described above, the memory is also freed explicitly in every module.
2. pass pointers around, taking care of restricting the functional graphs to avoid multiple branching and prevent undesired side effects. In this case we can rely on the Lisp garbage collector in order to get the memory automatically freed when an *o.bundle* is cleaned up.

In addition to facilitating communication between OpenMusic and other external environments, odot is used as a standard framework for the communication with external libraries as a way of integrating advanced audio processing and spatialization control in the compositional environment [12].

5. EXAMPLES

Here, we present two projects that exemplify as a common data structure that facilitates communication between environments.

5.1 Data Visualization With Bokeh

Bokeh is a data visualization package for Python that targets the browser and the HTML 5 canvas for rendering graphics. [15] Python code interacts with Bokeh objects and synchronizes them with a server running in a separate process. This server, in turn, stores the data sent to it in a key-value store such as redis [16] and uses javascript code running in the browser to render the data in an HTML 5 canvas.

In a project for music and dance, the heart rates of the dancers are measured by wearable electrocardiogram (ECG) devices, and transmitted to a computer over Wi-Fi using an X-odot [17] module, and converted to click tracks to control the tempo of the musicians in near real-time. The odot data transmitted by the X-odot module is processed in Max to produce a click-track. The data, containing the results and intermediate data produced by the feature-detection phase, is then forwarded from Max to Python code that unpacks the odot data, adds it to the Bokeh objects, and updates the Bokeh server.

The design of the python code is notable in that it leverages odot's ability to defer computation to the site at which it is needed. For example, given the example described above, imagine that we would like to plot the logarithm of one of the timeseries. Instead of performing that computation in Max/MSP where resources might need to be reserved for other tasks, we embed code into the odot bundle to be executed upon arrival at the Python script. This allows us to make changes dynamically without disturbing either environment, and even allows the two environments to run on separate machines on a network.

5.2 Adobe Illustrator

Another project involves the interpretation of symbolic notation authored in Adobe Illustrator in a variety of rendering contexts [13]. Utilizing the high level graphic design functionality of Illustrator, the system enables the user to simultaneously create high quality scores and visualizations of information and render their scores with other processes via OSC transcoding. Rendering contexts could include be sonic, spatial, kinetic, generative processes, or any process that is able to receive OSC. The benefit of such a system is to allow composers and other artists to find symbolic representation that best suits the performances parameters. For example, representing time varying spatial information in computer music is generally notated with univariate controls, splitting spatial coordinates into multiple single values, whereas in symbolic notation a 2D position can be clearly represented with a 2D plot (within a given resolution).

The project uses Scalable Vector Graphics (SVG) as a intermediate file format, saved by Illustrator and converted into OSC by a Max/Pd external (o.svg). The graphic information is then processed by an interpretation algorithm which selects temporal events and stores them in a table for lookup. The playback is achieved by selecting a time point in the score and outputting the events at this time (within a given threshold). The graphic parameters of the events (start and end positions, radius, color, stroke type, etc.) are mapped to the parameters of the output rendering process (amplitude, frequency, duration, modulation, etc.).

Adobe has recently announced Node.js support within Illustrator, allowing OSC to be sent directly from a script within Illustrator avoiding the need to use the intermediate SVG format. This interaction would facilitate being able to quickly preview short sections of the score while composing, as well as other interapplication possibilities.

6. CONCLUSION AND FUTURE WORK

Media/arts programming often requires the use of a number of software environments during the creation and presentation of a work. We have presented a set of tools that facilitate communication and interoperability between local and distributed software environments by providing a common data structure and computational environment.

In the short term, we plan to complete the implementation of the high level bindings for Javascript and Python upon which many current projects depend. We also plan Java bindings in order to target Processing [18] and Ptolemy.

As stated in Section 4.5, OSC via libo/odot has been considered in recent OM developments for the dynamic communication with external libraries or applications. In this context, several interfaces for the compositional specification, scheduling and playback of control data will be developed, as well as advanced tools for processing odot bundles, including support for the odot expression language.

The two projects discussed in section 5 are under active development and are expected to produce general all-purpose tools for the communication and visualization of odot data.

Finally, we note that this recent work has put pressure on the design of libo, exposing a number of underdeveloped areas of its API. Current and future work involves improving the interface to the library based on what we have learned through this process in order to simplify future binding work and support.

Acknowledgments

This work was supported in part by the TerraSwarm Research Center, [19] 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. The work on OpenMusic and Common Lisp is related to the French National Research Agency project ANR-13-JS02-0004.

[15] http://bokeh.pydata.org/en/latest/
[16] http://redis.io
[17] http://www.x-io.co.uk/products/x-osc/
[18] https://www.processing.org
[19] http://terraswarm.org

7. REFERENCES

[1] M. Wright and A. Freed, "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers," in *Proceedings of the International Computer Music Conference*, 1997.

[2] A. Freed and A. Schmeder, "Features and Future of Open Sound Control version 1.1 for NIME," in *Proceedings of the New Interfaces for Musical Expression Conference*, 2009.

[3] A. Schmeder, A. Freed, and D. Wessel, "Best practices for Open Sound Control," in *Linux Audio Conference*, 2010.

[4] A. Freed, J. MacCallum, and A. Schmeder, "Dynamic, Instance-based, Object-Oriented Programming (OOP) in Max/MSP using Open Sound Control (OSC) Message Delegation," in *Proceedings of the International Computer Music Conference*, 2011.

[5] J. MacCallum, A. Freed, and D. Wessel, "Agile Interface Development using OSC Expressions and Process Migration," in *Proceedings of the New Interfaces for Musical Expression Conference*, 2013.

[6] E. Curry, "Message-Oriented Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed. John Wiley and Sons, 2004, ch. 1, pp. 1–28.

[7] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, February 1984.

[8] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence," in *Proceedings of the International Joint Conferences on Artificial Intelligence*, 1973.

[9] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," in *Journal of Artificial Intelligence*, 1977.

[10] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014, ch. 1.7.2, pp. 20–23.

[11] G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue, "Computer Assisted Composition at Ircam: PatchWork & OpenMusic," *Computer Music Journal*, vol. 23, no. 3, 1999.

[12] J. Bresson, D. Bouche, J. Garcia, T. Carpentier, F. Jacquemard, J. MacCallum, and D. Schwarz, "Projet EFFICACE : Développements et perspectives en composition assistée par ordinateur," in *Journées d'Informatique Musicale*, Montréal, Canada, 2015.

[13] R. Gottfried, "SVG to OSC Transcoding: Towards a Platform for Notational Praxis and Electronic Performance," in *Proceedings of the International Conference on Technologies for Notation and Representation*, 2015.