o.OM: Structured-Functional Communication between Computer Music Systems using OSC and Odot

Jean Bresson, John Maccallum, Adrian Freed

o.OM: Structured-Functional Communication between Computer Music Systems using OSC and Odot

Jean Bresson[1,2] John MacCallum[1] Adrian Freed[1]

[1]Center for New Music and Audio Technologies, UC Berkeley, USA
[2]UMR STMS – Sorbonne Universités IRCAM-CNRS-UPMC, Paris, France
bresson@ircam.fr, adrian@cnmat.berkeley.edu, john@cnmat.berkeley.edu

Abstract

O.—*odot*—is a portable media programming framework based on the OSC data encoding. It embeds a small expression language which allows writing and executing programs in OSC structures. The integration of programming and declarative functional descriptions within data transfer protocols enables structured and expressive communication in media systems: program snippets can be distributed in OSC messages, which evaluate to further OSC messages in the different communicating software. We present experiments using this framework in the OpenMusic computer-aided composition environment, and illustrate via case studies some advantages of such integrated system.

Categories and Subject Descriptors D.1.1: Software [*Programming Techniques*]: Applicative (Functional) Programming

Keywords Musical domain-specific languages, OSC, Computer-aided composition

1. Introduction

Contemporary practice in computer music involves an increasing amount of communicating software and devices. In this context OpenSoundControl—OSC (Wright 2005)—has become a *lingua franca* for data transfer and control interactions. Applications and projects using it today are countless.

OSC *messages* have a fairly simple syntax and structure. They consist in a variable-length list of values preceded by an URL-like string called an *address*, which usually describes a "target" binding the values. They are generally streamed via UDP between applications or devices, although they may also be used as simple vehicles for structured information, e.g. in media programming environments. A number of features and constructs allow to structure OSC messages, such as:

- *Type-tags* assigned to each value in the messages. A pair (*type-tag* · *value*) is actually called and encoded as an *atom*.
- *Time-tags* which allow the setting of precise and universal-time labels to the messages in order to structure reliable timed control streams (Schmeder and Freed 2008).
- *Bundles*: aggregate data structures containing a set of messages.

FARM'16 September 18-22, 2016, Nara, Japan

ACM 978-1-4503-4432-6/16/09…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2975980.2975985

Despite these different features, however, flat streams of weakly timed messages are the common use, which results in a frequent lack of hierarchy and structure in most media communication frameworks using UDP and OSC.

The *odot* library (Freed et al. 2011) is a software package allowing to deal with OSC-encoded data in media programming environments, using OSC bundles as a base aggregate data structure to enhance internal and external data transmission schemes, program structuring, and expressivity in these environments. In particular, the library embeds a dedicated functional expression language allowing to process and transform bundles dynamically.

This paper is a report on experiments using *odot* in compositional processes implemented in OpenMusic (OM), another functional, visual programming language dedicated to computer-aided music composition (Assayag et al. 1999; Bresson et al. 2011). Embedding the *odot* expression language and programs in this context extends expressivity in communicating with other media environments by sharing a common language.

The paper is structured as follows: Section 2 first gives a brief introduction on OpenMusic and the general context of this work. Section 3 then presents the *odot* library, and Section 4 details different aspects of the *odot* integration in OpenMusic. Section 5 finally gives two examples of applications involving communication with the Max environment.

2. OpenMusic – Compositional Processes

OpenMusic (OM) is a visual programming language dedicated to computer-aided music composition. Based on and written entirely in Lisp/CLOS, it offers all programming features of this language (Bresson et al. 2009) and puts together graphically or textually-designed Lisp programs in advanced musical processes and workflows.

OpenMusic visual programs (also called *patches*) generally involve musical structures (scores, sounds, and other kind of musical material) processed by functional expressions designed as directed acyclic graphs. Output musical objects can be played (when relevant), inspected or modified via graphical editors. In contrast with numerous mainstream music programming systems, the execution paradigm of OM is called "demand-driven": it requires the user (composer/programmer) to evaluate components in these graphs when needed, in order to update values and produce musical structures.[1]

Figure 1 shows a very simple OM visual program, where the break-points of a hand-drawn graphic are mapped and converted to

[1] This execution strategy, also sometimes referred to as "deferred-time", is usually best suited to compositional tasks for it makes computation independent from real-time constraints, and actually makes the temporal dimension of musical structures just another parameter of the programs.

onset and pitch values instantiating chords in a score. The boxes on this visual program represent either functions (*mapcar*, *om-scale*), object constructors (BPF, SCORE), embedded programs/abstractions (the *make-chords* box actually stands for another sub-patch used as a *lambda* function connected to *mapcar*), or simple numerical values. They are patched together through connections determining the functional composition of the program. After programming the basics of this patch, a typical workflow with it would be for instance to edit the points in the editor, evaluate the SCORE at the bottom, modify the pitch values (6000, 8000) at the top, or eventually reprogram parts of it until reaching satisfactory results (this workflow is to be imagined and scaled to the context of more complex patches, involving multiple intermediate states and data structures—see (Agon et al. 2006-2008) for advanced examples of uses and applications of the OpenMusic environment).

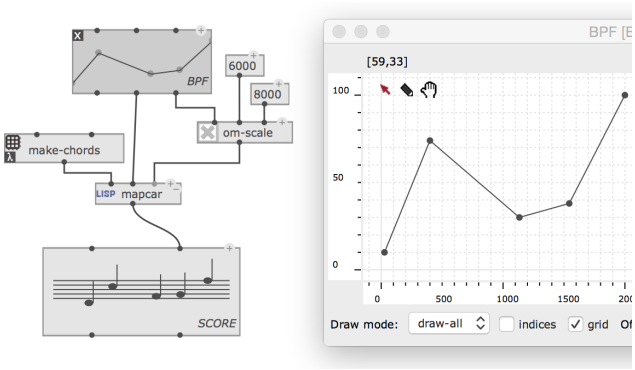

Figure 1. A simple visual program (or *patch*) in OpenMusic. An editor for the BPF object is open on the right.

OpenMusic communicates through OSC with other musical or multimedia environments. Typically such communication takes place with real-time systems like Max (Puckette 1991) or PureData (Puckette 1996). These popular graphical programming environments process data streams or signals in data-flow graphs and are commonly used for the implementation of interactive systems or sound processing units in music performances or media installations. OpenMusic and Max/PureData therefore operate in different paradigms that co-exist in music production (offline/compositional vs. real-time/interactive) (Puckette 2004; Giavitto 2014), and composers frequently use them synergistically.[2] In the remaining of this paper we will study interactions and communication strategies permitted by embedding functional specification and programming within the data transferred between these environments through the use of the *odot* library.

3. Odot

O.—*odot*—is a framework for media/arts programming that uses OSC as its fundamental datatype. It is a superset of OSC providing support for a small number of additional typetags,[3] as well as a functional expression-language interpreter, which makes it possible to write and evaluate expressions to process data contained in OSC bundles. The interpreter evaluates expressions taking an OSC bundle as unique argument and produces a modified copy of this bundle. Expressions themselves can be written and contained in OSC messages, making the bundle a rich context for data aggregation/storage, program writing, and execution.

The syntax and semantics of *odot* expressions are straightforward and inspired by popular scripting and functional languages. They include standard arithmetic operators on scalar and vectors, assignment (=), functional abstraction (*lambda*), higher-order functions such as *map* and *fold*, as well as several primitive operations to manipulate the contents of bundles (such as *assign*, *delete*, etc.). At evaluation time the incoming OSC bundle is copied to a "working" bundle, and the nested functions are computed using operand values referenced by address name from this bundle: the OSC addresses bind data contained in the bundle, and can be used to create new messages in it. For instance: *"/y = /x + 1"* means that a message with address */y* will take a new value computed from the current value of */x* if present in the working bundle, or it will be created and added into it otherwise.

The most mature implementation of *odot* is in the Max media programming framework, where the library can use OSC bundles as structured data containers passed and transformed via functional expressions in the real-time dataflow graphs (Freed et al. 2011). Figure 2 shows an example of a Max patch processing data using *odot* functions and user-defined operations written in the *odot* language.

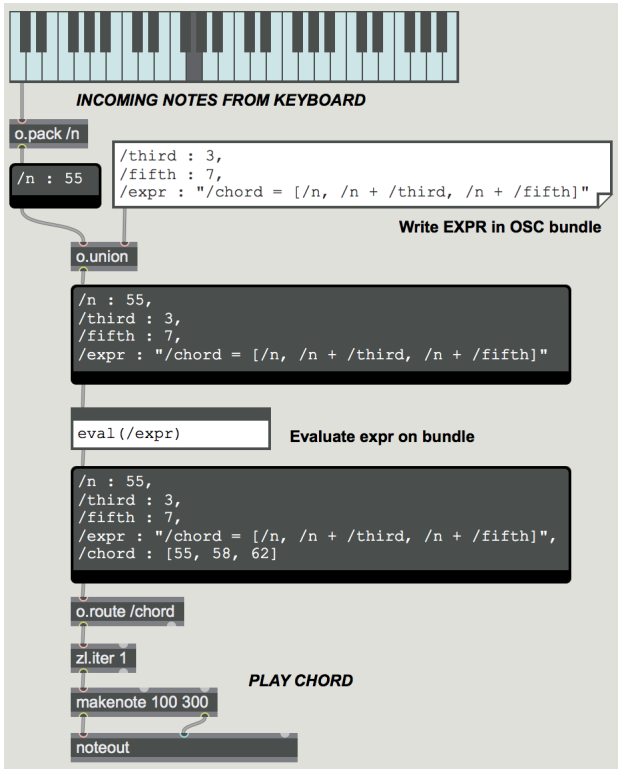

Figure 2. Simple reactive/data-flow patch in Max using *odot*. *o.union* joins bundles containing incoming data (e.g. from a MIDI keyboard) and some code written in the *odot* language. The box *eval(/expr)* evaluates the contents of */expr* as found in the incoming bundle in the context of this same bundle, and returns a new bundle. A *"/chord"* message is generated and added in the new bundle, later converted into MIDI notes by the Max functions at the bottom.

[2] In contrast with the OM "deferred-time" executions considered previously, "real-time" systems have no in-built conception of a time dimension in musical structures and continuously process individual bits of incoming data or audio.

[3] For instance, the support for a *bundle* type-tag allows for OSC messages to contain sub-bundles, implementing nested or hierarchical structures.

4. o.OM

OSC-encoded messages and bundles are used in OpenMusic for transferring data to external software, or to share this data with external linked libraries. Recent applications include for instance the integration of gesture data from augmented drawing devices (Garcia et al. 2014), the control of automatic guided improvisation systems (Nika et al. 2015), or the connection of compositional processes with mobile devices and spatial audio renderers (Garcia et al. 2015).

The work presented in this paper is an experimental implementation of the visual language integrating native support for OSC messages and bundles processing via a foreign function interface to the *odot* library (MacCallum et al. 2015).

4.1 Tools for the Manipulation of OSC Bundles

The basic data structure of our OSC framework is the class OSC-BUNDLE. An OSC-BUNDLE contains a list of OSC messages. Each message is a simple linked list containing an *address* (string) followed by a list of freely typed data. OSC-BUNDLEs also have a "date" attribute, determining their positing in temporal structures.

At a lower level, an OSC-BUNDLE is paired to another structure called an O.BUNDLE. The O.BUNDLE wraps around a pointer to a serialized binary representation of the OSC bundle generated via the *odot* library (including the set of messages and a *time-tag* generated from the date attribute), suitable for processing by this library or for direct transmission via UDP. Conversions between an OSC-BUNDLE and an O.BUNDLE can be implicit and internally carried out by the tools and functions of our framework, or they can be explicit via simple connections in OM visual programs. Figure 3 shows a simple OM visual program producing, encoding and sending out OSC bundles via UDP.

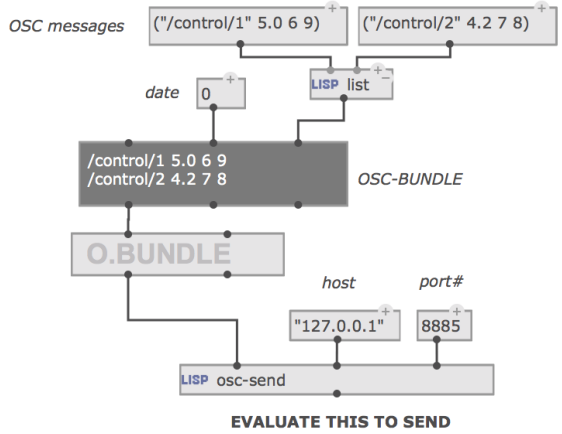

Figure 3. Formatting and sending an OSC bundle in an OM visual program. Note the use of O.BUNDLE to serialize the OSC messages as a binary stream.

Compositional environments like OM can help structuring compound temporal structures including OSC messages and bundles. Figure 4 shows a DATA-STREAM editor: a container and interface allowing to represent and render arbitrary data chunks laid out in a timeline, featuring timing control and programmable mapping of the data to graphical attributes (shape, size, position, color, etc.). This container can embed timed sequences of OSC-BUNDLEs, for instance to monitor (edit, visualize) the streaming of OSC messages to external applications. Every data chunk in it is associated to an action to perform during rendering/playback: in the case of an OSC bundle, this action by default will be a simple call to *osc-send* (as in Figure 3).

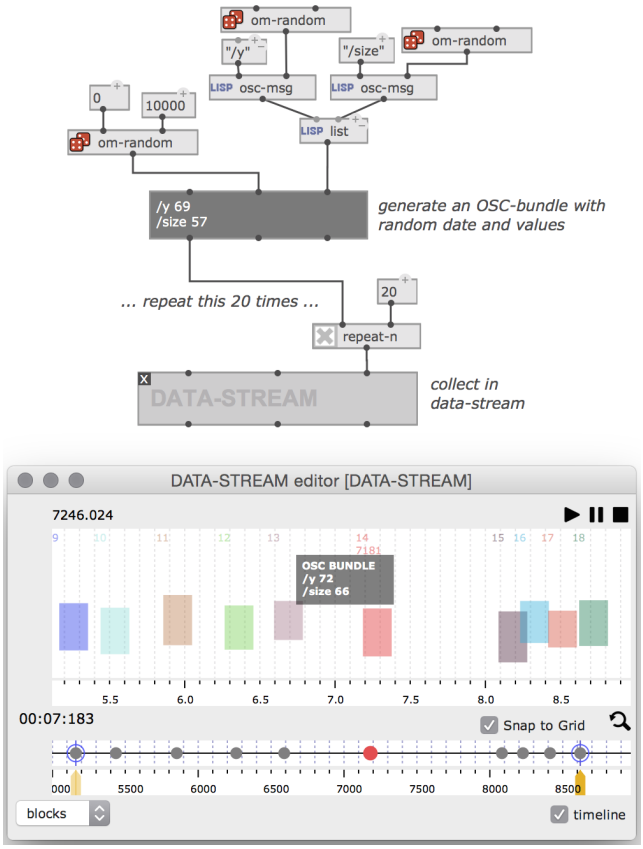

Figure 4. DATA-STREAM: a container and editor/renderer for timed data. In the patch at the top of the figure a sequence of dummy OSC-BUNDLEs are generated and collected in the DATA-STREAM. Visualization parameters for these bundles in the editor (size, vertical alignment, colour) are either determined by searching for specific messages in the bundles (e.g. "/y", "/size"), or just aleatory (e.g. colours).

4.2 Handling o. Expressions in Visual Programs

The *odot* expression language support in OM consists of two main features.

Formatting expressions using visual programs. A minimal set of primitives in OM allows the user to format *odot* expressions by evaluating a Lisp or visual program. The functions *o.map*, *o.lambda*, *o.call*, *o.delete*, etc. produce formatted strings which can be connected to each other in order to build the expressions.

Evaluating expressions. The *odot* language interpreter embedded in the library is used to parse the textual expressions, and to apply corresponding interpreted instructions to OSC bundles.

Figure 5 is an example of a simple *odot* expression built using an OM visual program, and applied to an OSC bundle. The function *o.eval-expr* evaluates the *odot* expression given as its first argument (left inlet) on a bundle given as second argument (right inlet), and returns a new bundle. Note that in this case, the body of the expression itself is in a message of the incoming bundle (*"/myfunction"*), and *o.eval-expr* only evaluates the *address* binding this expression.

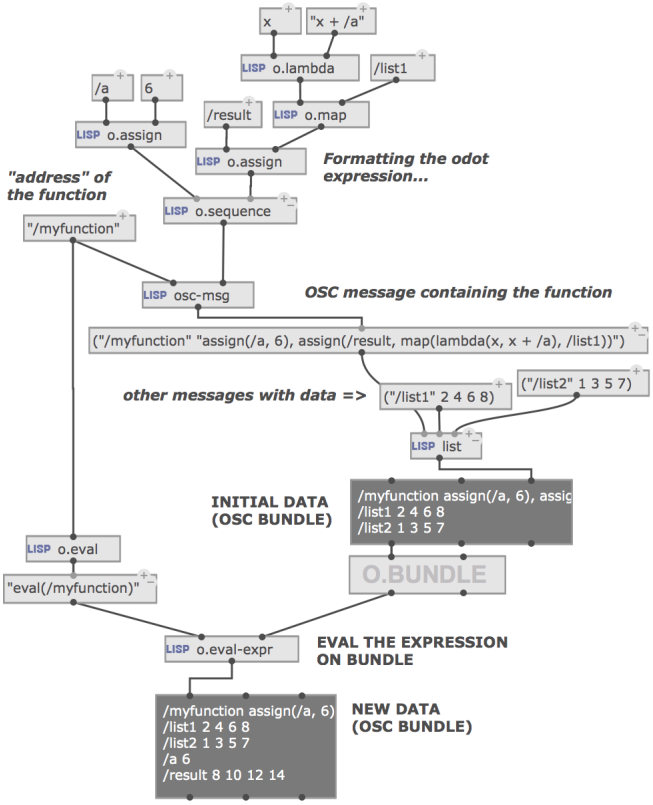

Figure 5. Construction and evaluation of an *odot* expression in an OM visual program.

5. Examples of Structured-Functional Communication with Real-time Media Systems

The usefulness of an expression language (*odot*) inside an other—already domain-specific—functional language (OM) might not appear straightforward at first sight. It is actually interesting that this simple expression language (as well that the format of the structures it operates on) be shared by communicating applications, so that programs can be arbitrarily written and evaluated at the different points of the communication within different environments and languages. In this section we present two examples of such applications connecting the OM and Max environments. We believe however that this approach can be generalised to other communicating frameworks.

5.1 Control of Spatial Audio Processing

This first example is based on a scenario previously studied in OM and computer-aided composition projects, consisting of generating spatial scene descriptions in OM, that are streamed to real-time systems in order to control spatial audio rendering (Bresson and Schumacher 2011). This use case often requires the encoding and streaming of significant amounts of structured data (each virtual sound source can be described by a large number of multi-dimensional spatial and audio descriptors). Functional specification can optimize the encoding of such spatial scene descriptions, especially in the case where for instance the positions (or some other attributes) of the sound sources are functionally related.

Let us take the example of a group of sound sources centred around a given, mobile point in space. The number of sources can be variable (and arbitrarily big) but the whole scene can be described by a small and constant-size set of messages including this number of sources, the central position of the group, and a functional specification of the distribution of the sources around this position. We may for instance represent a circular distribution around the center with the following *odot* bundle:

```
/x : -0.25,
/y : 1.96,
/spread : 1,
/n : 15,
/gensource : "lambda([i],
                     assign(\"/source/\"+string(i)+\"/xyz\",
                            [/x+(/spread*cos(/n*i)),
                             /y+(/spread*sin(/n*i)),
                             0])
                     )",
/cloudfun : "map(/gensource, aseq(1, /n))"
```

Figure 6 shows an OM patch producing this kind of bundle. Actually in this example, three different distribution functions are provided, among which the receiver might be able to choose in order to "unfold" the group of sources and get their actual positions. This bundle is part of a lambda function (created by *make-action*) that is attached to a 3D-curve as a "rendering action". Figure 7 shows the Max patch receiving the bundles produced by this action, and decoding them by choosing among the three available functions, then transmitting the unfolded data to the *spat* audio renderer and graphical interface (Carpentier et al. 2015).

This architecture allows the receiver to dynamically change the number of sources, the source distribution function, or any other parameter prior to rendering. The example therefore illustrates some notable features of the scripting language embedded in inter-application communication protocols, such as:

- The reduced and *constant* size of the streamed bundles, allowing to better control and deal with the bandwidth of transmission.[4]
- The possibility to write some interpretive instructions for the streamed data in an environment, leaving some choice and freedom to the receiver with regard to this interpretation.

5.2 Data Stream Visualization and Authoring

This second example goes the opposite way. Some data captured from an incoming data stream (e.g. from real-time image processing) is processed, sent out as OSC bundles and received in OM to be stored, visualized and edited (and eventually played back again).

In this case functional specifications written in the *odot* expression language are used to complement the OSC bundles with graphics-related elements computed frame-by-frame from the initial data. This allows the authoring of the mapping functions to be undertaken in any of the environments at hand and applied at any stage of the processing.

In Figure 8 incoming OSC bundles trigger reactive computations of the OM visual program (Bresson 2014). Each incoming bundle is transformed trough the application of a mapping function written in *odot*, which adds "/nshapes", "/y", "/h" and a set of */nshapes* messages bound to addresses of the form "/shape/[*i*]/xyr", each specifying the position and radius of the shape #*i*. The processed bundle is then added to the DATA-STREAM structure at the bottom of the visual program, which will use these graphical attributes to display a custom representation of the data (see Figure 9).

[4] The conceptual compactness and space efficiencies afforded by embedded programming languages have been demonstrated in different domain-specific contexts before with PostScript for graphical rendering (Adobe Systems Inc. 1985) with Aegis (Blewett et al. 1985), NeWS (Gosling et al. 2014) and Java, for client DOM interactivity and Lua (Ierusalimschy et al. 2007) for internet server appliance configuration.

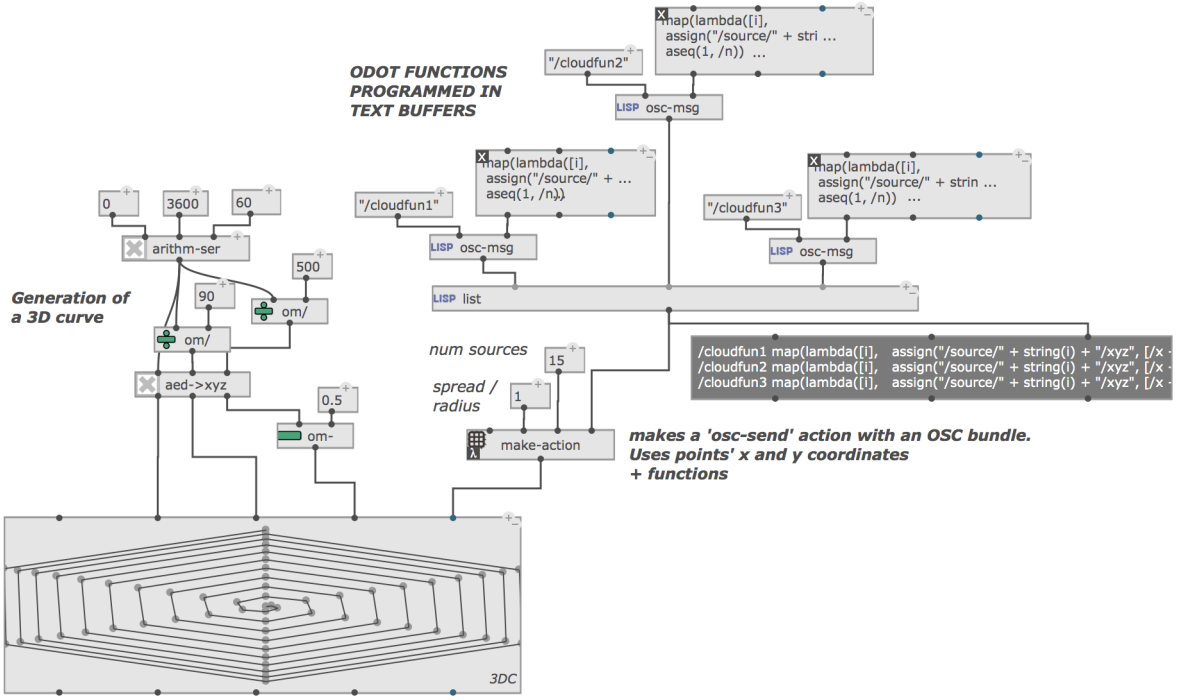

Figure 6. Formatting *odot* expressions and bundles in the rendering action for a 3D-curve in OM. The box *make-action* generates a lambda-expression containing an *osc-send* call with the produced bundle from each successive point in the curve. On the right, one such bundle is partially displayed for illustrative purpose (the bundles are actually created on-the-fly by the lambda-expression at rendering time).

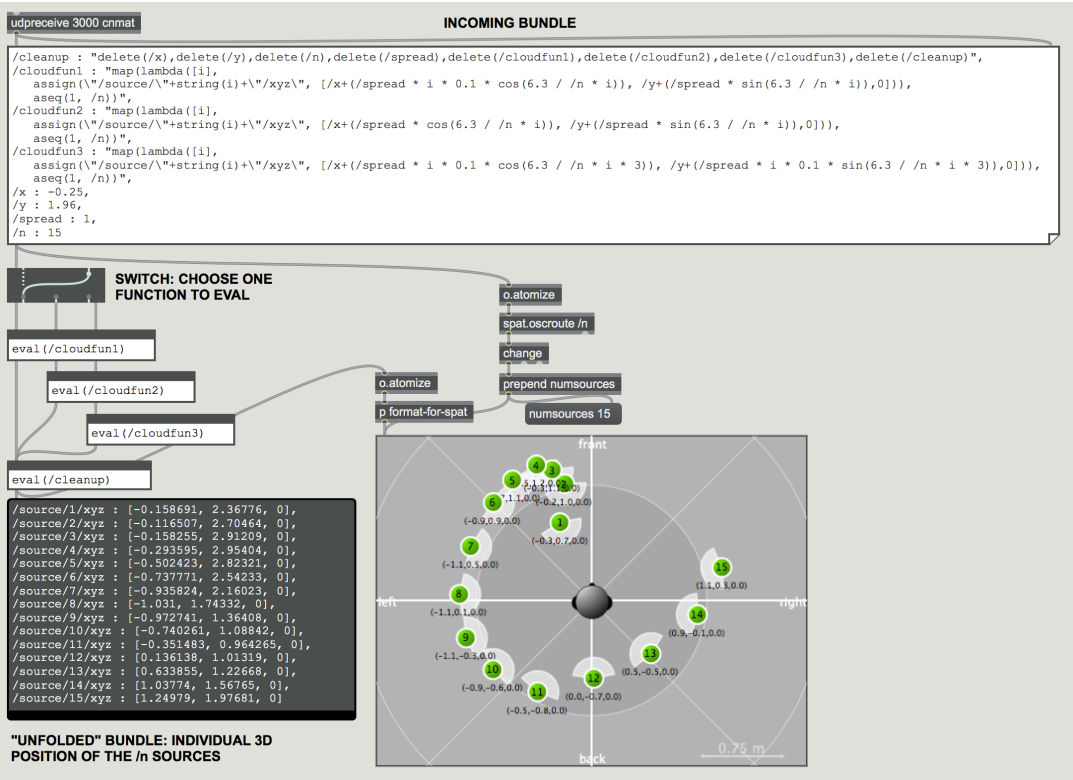

Figure 7. Receiving OSC bundles from the 3D-curve playback (Figure 6). The selection and evaluation of one of the proposed *odot* functions generate a set of sources distributed following a specific rule around a central position (*/x*, */y*).

Additional features brought forward by this example are:

- The versatility of the representation, freely controlled and adapted in either the sender or receiver part of the system using the same language (one could also imagine a system of switch between alternative representations, as presented in the previous example).
- The possibility to control some aspects of the visualization from a remote environment (where the original data comes from, e.g. Max), without programming it (in this case, in Lisp) in the authoring environment itself.

6. Conclusion

We have presented some applications of the *odot* framework and expression language in the OpenMusic computer-aided composition and visual programming environment. *Odot* expressions operate directly on OSC-encoded representations of the data (the most widespread format used in computer music) and can be embedded in this format. In addition to structuring the communication between environments by gathering full control statements instead of exchanging sparse control messages, OSC communication is therefore enhanced by computational aspects and functional programming expressivity.

We have shown in the presented use cases how declarative functional descriptions can optimize the communication by reducing/containing the size of the transferred data, but also how they can permit foreign environments to share program specification and executions through a common language—an approach easily applicable to other host frameworks and languages.

Acknowledgments

This project was carried out with funding and support from the Fulbright Franco-American Commission and the French National Research Agency (ANR) project with reference ANR-13-JS02-0004. The authors would like to thank Rama Gottfried, Ilya Rostovtsev and Jeff Lubow for their helpful support.

References

Adobe Systems Inc. *Postscript Language Reference Manual* (1st Ed.), Addison-Wesley Longman Publishing Co., 1985.

C. Agon, G. Assayag, and J. Bresson, editors. *The OM Composer's Book 1 & 2*. Editions Delatour / IRCAM Centre Pompidou, 2006-2008.

G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue. Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic. *Computer Music Journal*, 23(3), 1999.

C. Blewett, A. Freed, J. Langer, R. Mascitti, C. Rodine, and W. Weber, *The Aegis System*. AT&T Bell Labs Internal Memorandum, 1985.

D. Bouche and J. Bresson. Articulation dynamique de structures temporelles pour l'informatique musicale. In *Modélisation des Systèmes Réactifs (MSR)*, Nancy, France, 2015.

J. Bresson. Reactive visual programs for computer-aided music composition. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Melbourne, Australia, 2014.

J. Bresson, C. Agon, and G. Assayag. Visual Lisp/CLOS Programming in OpenMusic. *Higher-Order and Symbolic Computation*, 22(1), 2009.

J. Bresson and M. Schumacher. Representation and Interchange of Sound Spatialization Data for Compositional Applications. In *Proceedings of the International Computer Music Conference*, Huddersfield, UK, 2011.

J. Bresson, C. Agon, and G. Assayag. OpenMusic – Visual Programming Environment for Music Composition, Analysis and Research. In *Proceedings of ACM MultiMedia (OpenSource Software Competition)*, Scottsdale, USA, 2011.

J. Bresson and J.-L. Giavitto. A Reactive Extension of the OpenMusic Visual Programming Language. *Journal of Visual Languages and Computing*, 25(4):363–375, 2014.

T. Carpentier, M. Noisternig, and O. Warusfel. Twenty Years of Ircam Spat: Looking Back, Looking Forward. In *Proceedings of the International Computer Music Conference*, Denton, USA, 2015.

A. Freed, J. MacCallum, and A. Schmeder. A Dynamic, Instance-Based, Object-Oriented Programming in Max/MSP using Open Sound Control Message Delegation. In *Proceedings of the International Computer Music Conference*, Huddersfield, UK, 2011.

J. Garcia, P. Leroux, and J. Bresson. pOM: Linking Pen Gestures to Computer-Aided Composition Processes. In *Proceedings of the 40th International Computer Music Conference (ICMC) joint with the 11th Sound & Music Computing conference (SMC)*, Athenes, Greece, 2014.

J. Garcia, J. Bresson, and T. Carpentier. Towards Interactive Authoring Tools for Composing Spatialization. In *Proceedings of the IEEE 10th Symposium on 3D User Interfaces*, Arles, France, 2015.

J.-L. Giavitto. Du temps écrit au temps produit en informatique musicale. In H. Vinet, editor, *Produire le temps*, Hermann, 2014.

J. Gosling, D. S.H. Rosenthal, and M. J. Arden, *The NeWS Book: An Introduction to the Network/Extensible Window System*. Springer Science & Business Media, 1989.

R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, New York, NY, USA, 2007.

J. MacCallum, R. Gottfried, I. Rostovtsev, J. Bresson, and A. Freed. Dynamic Message-Oriented Middleware with Open Sound Control and Odot. In *Proceedings of the International Computer Music Conference*, Denton, USA, 2015.

J. Nika, D. Bouche, J. Bresson, M. Chemillier, and G. Assayag. Guided improvisation as dynamic calls to an offline model. In *Proceedings of the Sound and Music Computing conference (SMC)*, Maynooth, Ireland, 2015.

M. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3), 1991.

M. Puckette. Pure Data : another integrated computer music environment. In *Second Intercollege Computer Music Concerts*, Tachikawa, Japan, 1996.

M. Puckette. A divide between 'compositional' and 'performative' aspects of Pd. In *First International Pd Convention*, Graz, Austria, 2004.

A. Schmeder and A. Freed. Implementation and Applications of Open Sound Control Timestamps. In *Proceedings of the International Computer Music Conference*, Belfast, 2008.

M. Wright. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10(3), 2005.

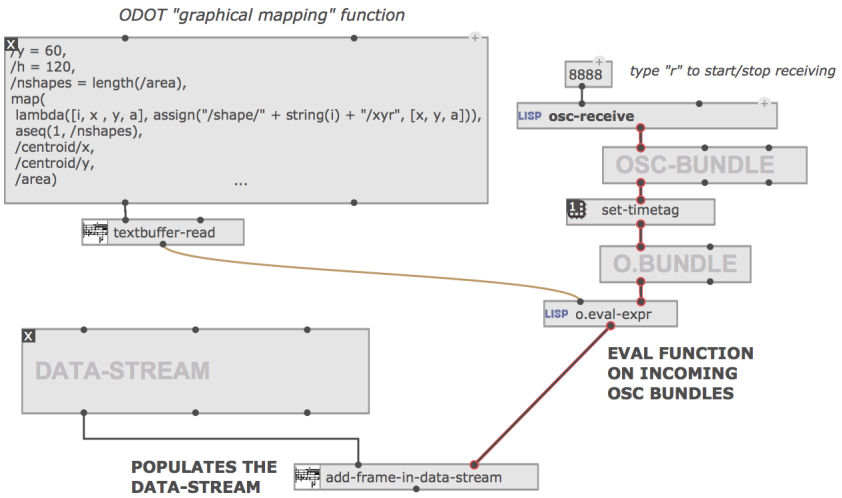

Figure 8. Receiving contour recognition data as OSC bundles in OM, on-the-fly application of a graphical mapping (*odot* function) and storage in a DATA-STREAM.

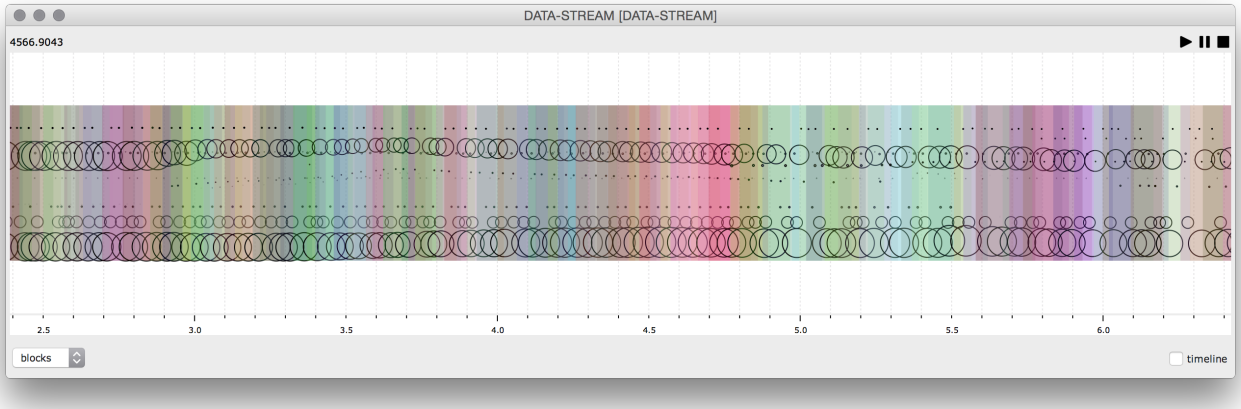

Figure 9. Visualization of the OSC bundles received and processed in Figure 8 in the DATA-STREAM editor. The black circles, their position and sizes are determined by the mapping programmed in the *odot* expression.