SYMBOLIST: BIDIRECTIONAL MAPPING DEVELOPEMENTS

Rama Gottfried
University for Music and Theater
Hamburg, Germany
rama.gottfried@hfmt-hamburg.de

James Tsz-Him Cheung
University for Music and Theater
Hamburg, Germany
tsz.him.cheung@hfmt-hamburg.de

Georg Hajdu
University for Music and The
Hamburg, Germany
georg.hajdu@hfmt-hambu

ABSTRACT

maybe already start with history here, and then continue to symbolist?

SYMBOLIST is an in-development application for experimental notation, with the goal of creating a working environment for developing symbolic notation for multimedia which can be interpreted and performed by electronics. The program aims to provide an open play space, with tools for experimentation, and thinking visually about relationships between representation and interpretation in media performance. In the paper we discuss the evaluation and re-design of the application based on the need for a bidirectional mapping framework for working with symbolic notation and its corresponding data representations.

1. BACKGROUND

The SYMBOLIST project was developed out of an organic
  of compositional notation practice.
  how much of a background?

In this paper we present a case study for the creation of an open system for graphically developing symbolic notation which can function both as professional quality print or online documentation, as well as a computer performable score in electro-acoustic music and other computer aided contexts. Leveraging Adobe Illustrator's graphic design tools and support for the Scalable Vector Graphics (SVG) file format [1], the study shows that SVG, being based on Extensible Markup Language (XML), can be similarly used as a tree-based container for score information. In the study, OpenSoundControl (OSC) [2] serves as middleware used to interpret the SVG representation and finally realize this interpretation in the intended media context (electronic music, spatial audio, sound art, kinetic art, video, etc.). The paper discusses how this interpretive layer is made possible through the separation of visual representation from the act of rendering, and describes details of the current implementation, and outlines future developments for the project.

Initially developed as a method for performing vector graphics, SVG-OSC [3]
  [4] [5]

XML nature of SVG makes it easy to parse and map just like and OSC bundle.

drawing on the OSC research at CNMAT,

just like a piece of paper, SVG could be freely mappable, and *performable* like OSC.

however it can require a lot of specialized code to handle different hierarchy structures and data at different levels in the hierarchy.

the first version of SYMBOLIST was created as a standalone JUCE application Ircam/ZKM [6]

1.1 JUCE Version

explain juce version (include in overview?) MVC [7] focused on creating SVG and outputting OSC version (like SVG-OSC approach)

1.2 Clefs and Bidirectional Mapping

telling story of how we got where we are, conceptual steps
  idea of Clefs
  problem: a lot of mapping is still needed

1.3 Drawsocket

2. SYMBOLIST ELECTRON

using DRAWSOCKET as front end handler [8]
  [9]

a general overview about need for classes (actually this comes later when we get into the the discussion of format i.e. the way the data is represented.

js and DRAWSOCKET makes that easy(-ier)

Figure 1 shows a screenshot from the Electron version.

3. IMPLEMENTATIONS

Electron [1] version (node + chrome w/ special electron IPC)
Max version (node + jweb + Max patch for IPC)

3.1 Application Structure

The main architecture of the platform (figure 2) is organized as a server-client model, comprising of:

1. The *editor*, a browser-based user interface client which displays the graphic representation of the data, and allows the user to edit and create new data objects through graphic interaction. The *ui_controller* runs in the browser handling interaction, via a library of
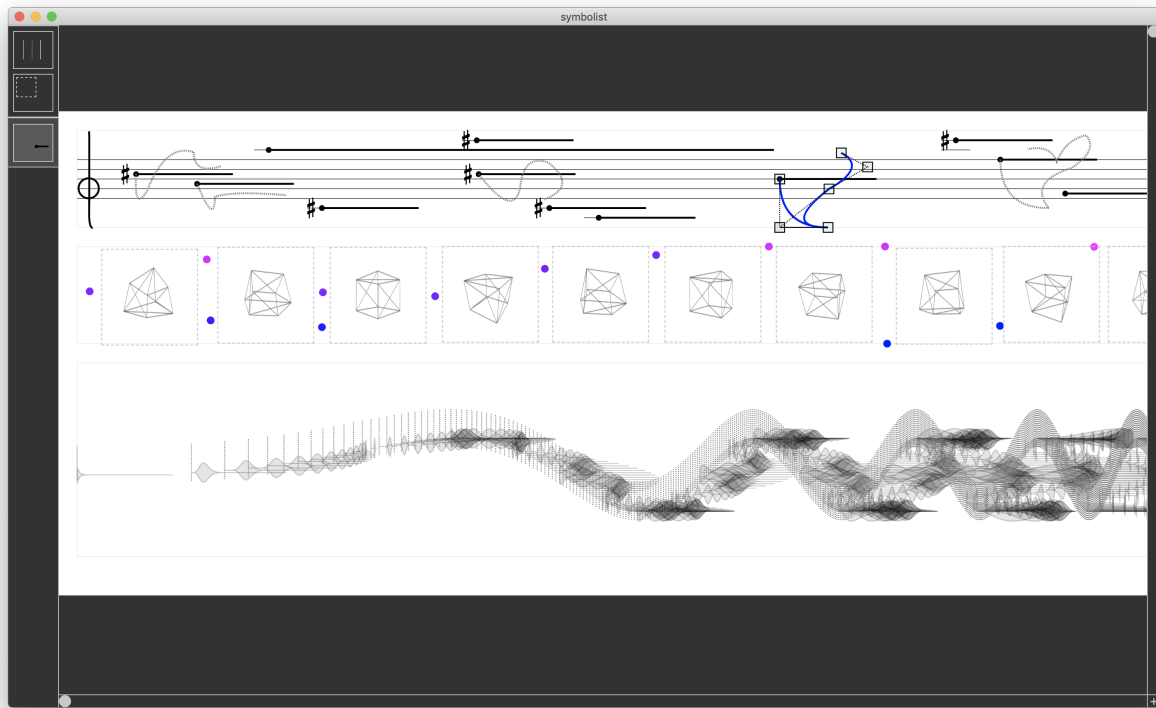
---

[1] https://www.electronjs.org/

Figure 1. SYMBOLIST screenshot, showing some different types of staves, and editing capabilities.
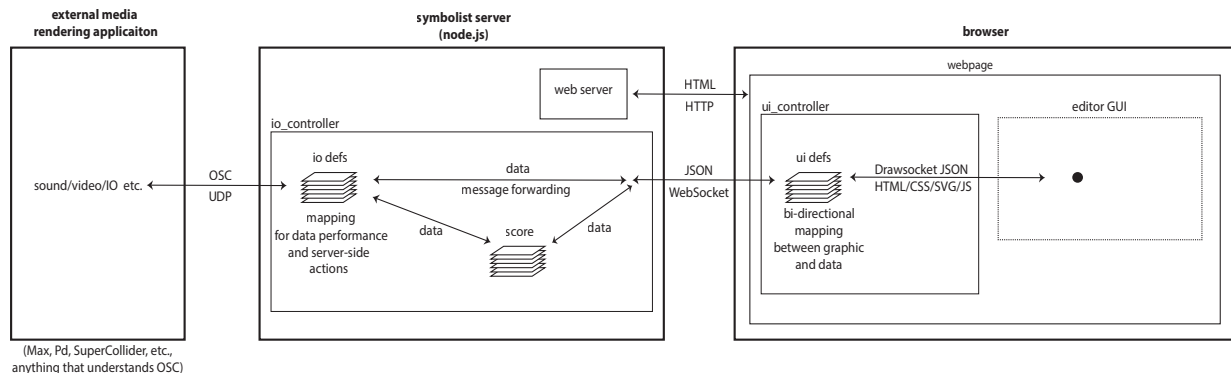

Figure 2. SYMBOLIST architecture.

definition scripts which specify mappings to and from data and graphics formats, as well as other tools and interactions.

2. The *symbolist server*, which is comprised of:

   - A node.js (Electron) based web-server which serves the main editor webpage via HTTP, and routes messages between the *ui_controller* and the *io_controller* via WebSockets, and handles operating system commands like reading and writing files.

   - The *io_controller* script runs inside the *symbolist server* and handles input and output from external sources via OSC over a UDP socket. The *io_controller* holds a copy of the score in its semantic format, and loads a parallel library

of user scripts to the *ui_controller* which define the mapping to (and potentially from) other media sources. The *io_controller* can also be used to reformat the score into a format that can be performed by another sequencing tool or program like MaxMSP.

4. GRAPHICAL AUTHORING AND INTERACTION

The graphic user interface of SYMBOLIST is designed around the idea of symbol objects and containers. Graphic objects, or *symbols* are placed in *container* references which define a framing used to interpret the meaning of the *symbol*.

In order to maintain an open and un-opinionated approach to authoring tools, SYMBOLIST tries not to specify how

containers and symbols should look, act, or respond when you interact with them within the application. Rather, the interaction and meanings of the symbols are defined in a library of object *definitions* which create these meanings through mapping semantic data to and from the graphic visualization. Definitions can be shared and loaded to setup different composition environments.

4.1 Interface Components

The SYMBOLIST graphic editor (shown in figure 1) provides a set of basic tools for creating scores using the defined symbols and containers:

- *document view*: the top level view of the application window.

- *menu bar*: the menu bar at the top of the screen or window, which provides access to various application functions.

- *palette*: a set of buttons on in the side bar of the program which display icons of the *symbols* that have been defined for the current selected *container*.

- *tools*: a set of buttons that open tools for custom computer-assisted generation of new symbols, and applying transformations to existing elements (e.g. alignment of multiple objects, or setting distributing objects, etc.)

- *inspector*: a contextual menu for editing the semantic data of an object, which is then mapped to the graphic representation.

4.2 Modes

maybe this should just be a list

4.2.1 Palette and Creation

4.2.2 Selection, Click and Drag

4.2.3 Inspector

4.2.4 Edit

4.3 User Experience

On entering the application, the editor loads a score or initialization file from the default load folder, or you can load a new config file after loading. The config file sets the top-level page setup and palette options.

A typical sequence of creating a score might be as follows:

1. The user opens a workspace, with one or more containers displayed on the screen, for example an empty rectangle which is like a piece of paper.

2. Selecting the "paper" container rectangle, the user then sets the container as the new *context*, by pressing the [s] key (or selecting from the application menu).

3. Once setting the context, the *palette* toolbar is populated with icons of symbols that are defined with the selected container context type.

4. Clicking on one of the symbol icons, puts the interface into *creation / palette mode*, where the mouse interaction is now designed for use with this specific symbol type.

5. Holding the CMD button, creates a preview of the symbol how it will appear when you click, and some text is displayed near the mouse that shows the semantic data associated with the graphic representation.

6. After clicking the symbol is placed in the container.

7. Depending on the symbol type, you may be able to drag the symbol to a new place in the container, and the associated data is updated as a result.

8. Selecting the symbol and hitting the [i] button, brings up the inspector window, where you can edit the data and see the graphics updated in response.

9. Selecting and pressing the [e] button enters *edit mode* which is a modal context where different user interaction could change the values of the symbol in different ways. For example in edit mode you might be able to rotate an object in a certain way, or be able to visualize different connections to the graphic representation to other elements of the score which are not usually highlighted in the score view.

5. DATA REPRESENTATION

At the heart of SYMBOLIST are two parallel forms of information expression: *semantic data* and *graphic representation* (figure 3).

Semantic data specifies the various attributes of information about a symbolic object, in terms of the object's meaning to the author. For example, the meaningful attributes of a *note* object might be information about pitch and duration, or a *point* object might contain x, y, and z values corresponding to the point's location in 3D space. In SYMBOLIST *semantic data* is thought of as the main holder of information in the system, which arranged as a score can function like a database of hierarchal information (further explained in section **??** below).

The *graphic representation* of the information is a visual representation of the semantic data, which is open in nature. The aim of SYMBOLIST is to provide an agnostic framework for developing visual, symbolic, or other unknown representations of semantic data for use in multimedia composition practice; and so, one of the main functions of the new version of the software is to facilitate the creation of mapping relationships between different representations of the data (see figure 3).

6. SYMBOLS

In SYMBOLIST terminology a *symbol* is an instance of a symbolic representation of data that connects the semantic, graphic, and possibly other media types of expression together as a multifaceted unit. Each *symbol* is defined as a *class* of object, which specifies the symbol's data structure and UI interaction, and data mapping to different representational contexts.

Semantic data | Graphic representation

```
{
    note: a4,
    start-time: 1,
    duration: 1s,
    amplitude: 1
}
```


Figure 3. *data* vs *graphic* representation of the same information.

6.1 Semantic Data Format

Within the SYMBOLIST application, semantic data is stored as javascript objects, and read/written in JSON format [10], which is transcoded to and from OSC for inter-application communication.

The main attributes used for SYMBOLIST semantic data objects are:

- *id*: a unique identifier name (required).

- *class*: a reference to the definition of the object type in the user-definition library (required).

- *contents*: an array of child objects that a parent container object might hold (required for container symbols).

In addition to the required *id* and *class* attributes, symbol objects may include any number of other *attributes* [2] of the symbol (*pitch*, *amplitude*, etc.). For example a simple semantic object written in JSON might look like:

```
{
    "id" : "foo",
    "class" : "legs",
    "action" : "jump",
    "startTime" : 0.1
}
```

Here we see an object with the *id* "foo," which is of *class* type "legs", that has an attribute *action* associated with it and a start time.

6.2 Containers

Symbols may also contain other symbols. Container symbols function to frame their contents, giving reference and context, like a plot graph frame, which provides a perspective for interpreting a set of data points.

When a symbol contains other symbols, the child symbols are stored as an array in the object's *contents* field. For example an imaginary class *timeline*, which holds two types of leg actions, might look like this:

```
{
    "id" : "bar",
    "class" : "timeline",
    "duration" : 1,
    "contents" : [{
        "id" : "foo−1",
        "class" : "legs",
        "action" : "jump",
        "startTime" : 0.1
    },{
        "id" : "foo−2",
        "class" : "legs",
```

---

[2] The term *attribute* is used here interchangeably with properties, parameters, aspects, etc.

```
        "action" : "sit",
        "startTime" : 0.2
    }]
}
```

In most cases, a symbol's mapping definition will require querying its parent symbol for information, to plot its data relative to the container context, for example offsetting the screen coordinate position based on the parent object position.

6.3 Score, File Format

Using symbols and symbol containers, we can create tree structures which can be used to represent hierarchical grouping; to represent scores, or other types of data structures. At the root of the tree structure is a top-level symbol, which might (but not necessarily) define behavior of its children objects. Since the data elements are stored in js objects, it is easy to import/export SYMBOLIST scores as JSON files.

When the application loads, it reads a default initialization file, in the form of a SYMBOLIST score. The current default initialization config file looks like this:

```
{
    "about" : "symbolist will read a json file to configure the
        palette setup, this can be used to dynamically change
        the application layout and tools",
    "id" : "Score",
    "class" : "RootSymbol",
    "tools" : [],
    "palette" : ["SubdivisionTool", "BasicSymbolGL"],
    "contents": {
        "id" : "trio",
        "class" : "SystemContainer",
        "x": 200,
        "y": 100,
        "duration": 20,
        "time": 0,
        "contents" : [{
            "id" : "oboe",
            "class" : "FiveLineStave",
            "height" : 100,
            "lineSpacing" : 10,
            "duration": 20,
            "time": 0,
            "contents" : []
        },
        {
            "id" : "bassoon",
            "class" : "PartStave",
            "height" : 100,
            "time": 0,
            "duration": 20,
            "contents" : []
        },
        {
            "id" : "synth",
            "class" : "PartStave",
            "height" : 200,
            "time": 0,
            "duration": 20,
            "contents" : []
        }]
    }
}
```

The initialization file is literally a score object file, providing the default context for a given authoring situation. In this example, we can see there is a "RootSymbol", which contains a "SystemContainer", which in turn contains two "PartStave" symbols and one "FiveLineStave" symbol (which are all actually containers as well, initialized as empty arrays). The *palette* and *tools* attributes tell the application to provide access to certain tools, and child symbols in the GUI (see also section 4).

7. GRAPHIC DISPLAY FORMAT

The graphic representation of the data in SYMBOLIST uses SVG format, and and follows the same hierarchical structure of the data as found in the semantic data score object. Since the new version of SYMBOLIST uses a browser as a frontend, we are able to take advantage of the many standard tools and web functionalities provided by browsers for display, interaction and data management.

Like the JSON score initialization file above, the main application window is setup using HTML/CSS, and utilizing DRAWSOCKET as a convenience wrapper to provide shorthand methods to create and manipulating browser window elements.

7.1 SVG

The SYMBOLIST format for an SVG *symbol* is a one top group (<g>) elements, with two sub-groups for *display* and *contents*, using HTML/CSS class names. The most simple SVG symbol would be:

```
<g id="foo" class="SymbolClassName symbol">
    <g class='SymbolClassName display'></g>
    <g class='SymbolClassName contents'></g>
</g>
```

Just like the semantic data objects, graphics objects have required *id* and *class* parameters, with an optional *contents* element.

Each symbol grouping element is tagged using class names, following the symbol's unique class name (in this example "SymbolClassName"). Note that the order is important: *the symbol class type must be first*. The *symbol* tag marks the top-level grouping object of the symbol, the *display* element is a group that holds all of this symbol's visual display information, and the *contents* is an group object for holding any potential child elements. Note that for simplicity, all SYMBOLIST graphic elements include the *contents* element as a placeholder.

7.2 HTML

Symbols and containers could also potentially be HTML elements instead of SVG. In the case of HTML you would use <div> tags instead of SVG <g>: html:

```
<div class='SymbolClassName symbol'>
    <div class='SymbolClassName display'></div>
    <div class='SymbolClassName contents'></div>
</div>
```

7.3 dataset-elements

Since SYMBOLIST is constantly mapping to and from semantic data and its graphic representation, we are using the HTML *dataset* feature [3] to store the semantic data inside the top-level *symbol* element (<g> or <div>). The HTML dataset attributes use the prefix "*data-*".[4]

For example, mapping our imaginary "legs" actions above, the corresponding SVG objects would be (skipping the actual display drawing for now):

---

[3] https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dataset
[4] Note that according to the HTML dataset specifications, all names will be converted to lowercase, this can create issues in some cases, so best practice is to use all lowercase for attribute names.

```
<g id="bar" class="timeline symbol" data-duration="1">
  <g class='Timeline display'></g>
  <g class='Timeline contents'>
      <g id="foo-1" class="Legs symbol" data-action="
          jump" data-starttime="0.1">
        <g class='Legs display'></g>
        <g class='Legs contents'></g>
      </g>
      <g id="foo-2" class="Legs symbol" data-action="sit"
          data-starttime="0.2">
        <g class='Legs display'></g>
        <g class='Legs contents'></g>
      </g>
  </g>
</g>
```

7.4 Performing Data

Just as the *graphic representation* can be seen as a visual expression of *semantic data*, the same semantic data can also used as control data in connection with other media forms. For example, a *note* object's pitch, onset, and duration information could be used to trigger a note on a synthesizer, or a sequence of Labanotation [11] could be used to guide the movement of robotic motors, create haptic feedback for live performance [12], and so on.

SYMBOLIST provides several different options for sorting and looking up data (see figure 6), which can serve as a structure for the performance of a "score", or other data formats. Typically, some representation of time is used to indicate an object's moment of action, but in SYMBOLIST the exact nature of the temporal organization is up to to the author (see section 9 for further discussion of performing data).

In addition to the *semantic* and *graphic* contexts of data representation, we can think of the *performance* of the data as a third data context context.

7.5 Mapping

Between each of these representation contexts there is a layer of mapping, with the *semantic data* serving as the primary representation type.

Semantic data to graphic representation mapping (figure 4) is used for the creation of graphic symbols from a stream of input, for example from generative processes, textural authoring, or computer assisted composition systems [13, 14, 15, 16, 17].

Graphic representation to semantic data mapping (figure 5) is used in order to create or edit data based on graphic information. This is the typical "graphical user interface" situation, where the data is accessible through its visual representation.

Semantic data to performance media mapping (figure 6) is the use of the data as a sequence of events that can be played in time (or used to control other processes not necessarily in time).

Note that in SYMBOLIST mapping between *performance media* and *graphic representation* is achieved through first mapping to semantic data. See section 8 for further discussion. Note also that the *ui_controller* receives and outputs messages always in in the *semantic data* format, keeping the concerns of drawing within the browser-side. That said, it is also possible to send DRAWSOCKET messages directly to the UI via the OSC-UDP communication port.
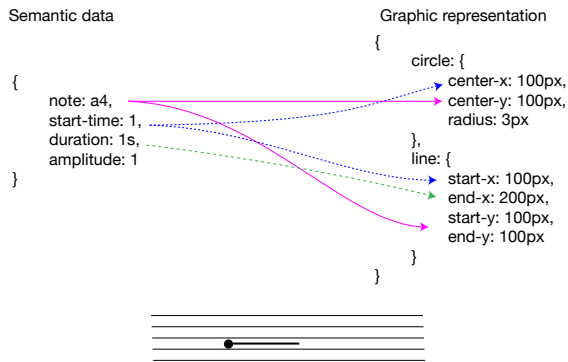
Figure 4. *semantic data* mapped to create a *graphic* representation from input data.
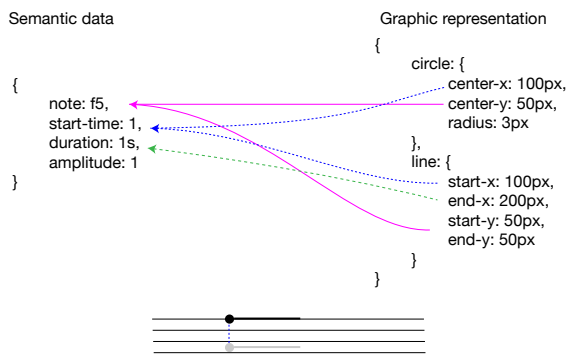

Figure 5. If edited graphically, the updated graphic data is then mapped back to *semantic data* representation.
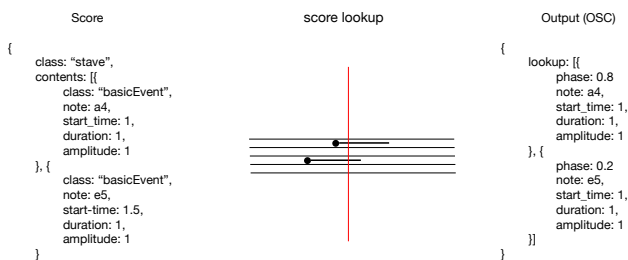

Figure 6. Using the lookup method defined by the symbol class, the *semantic data* can be used to perform external instruments via Open Sound Control.

8. SYMBOL DEFINITIONS

Definition scripts are composed as Javascript modules which are loaded into the program at runtime.

Eventually it is planned to provide a set of tools in the GUI for defining a mapping definition graphically but this is not yet implemented.

There are two types of definition scripts: * 'ui' definitions perform user interactions and mapping between semantic data representation and graphic representation. * 'io' definitions are used to assist in the lookup/playback and mapping of the semantic data to media like sound synthesis, video, etc.

Currently, the system uses the same '.js' file to hold both the 'ui' and 'io' definitions. To aid in development there is a template file that can be used to handle most of the most common actions.

io and ui elements

discuss using DOM elements for hierarchy and js document.querySelector in ui and similar options in io using score vs model

The data objects is stored in a *model* or *score* which is made up of a hierarchical arrangement of objects called *symbols*.

maybe this should be renamed .. the main point here is the content aka symbol, which is represented in different ways in different contexts, ui/io aka semantic and graphic versions.

However, since the *ui_controller* and user definition scripts are all being processed in the browser, scripts are free to use traditional JS approaches to manipulating the browser DOM.

8.1 Creating Symbols via OSC

a mechanism for mapping semantic data received from an external application via OSC to symbolic graphic representation; for example by generating scores through algorithmic processes, via textual authoring, via mapping from gestural controller, etc.

using DRAWSOCKET *key/val* message syntax:

data: adds a data object to the score, and sends to the ui to be mapped to graphical representation. Parameters include:

- *class* (required) the class type of the object to create
- *container* (required) the container symbol class to put the object in (in case there are multiple containers that support the same symbol type)
- *id* (optional) an id to use for the data object, if non is specified a (long) unique string will be generated.
- Other required or optional parameters will depend on the symbol definition.

```
{
    /key : "data",
    /val : {
        /class : "fiveLineStaveEvent",
        /id : "foo"
        /container : "oboe",
        /time : 0.13622,
        /ratio : "7/4",
        /duration : 0.1,
        /amp : 1
    }
}
```

8.2 API Template SymbolBaseClass

Using the template, a basic definition might look like this:

9. IO MESSAGES

SYMBOLIST has built in handlers for a set of messages received via OSC, which can be extended by user scripts, using a key/val syntax, where the *key* specifies the function to call, and the *val* are the parameter values to use for the call.

For example, here is a *lookup* query to find elements that are returned by the parameters *time* in the container with the *id* "trio".

```
{
    /key : "lookup:,
    /val : {
        /time : 0.1,
        /id : "trio"
    }
}
```

The OSC message API supports the following keys:

- *data*: adds a data object to the score, and sends to the ui to be mapped to graphical representation. Parameters include:

  - *class* (required) the class type of the object to create
  - *container* (required) the container symbol class to put the object in (in case there are multiple containers that support the same symbol type)
  - *id* (optional) an id to use for the data object, if non is specified a (long) unique string will be generated.
  - Other required or optional parameters will depend on the symbol definition.

```
{
    /key : "data",
    /val : {
        /class : "fiveLineStaveEvent",
        /id : "foo"
        /container : "oboe",
        /time : 0.13622,
        /ratio : "7/4",
        /duration : 0.1,
        /amp : 1
    }
}
```

- *lookup*: looks up a point in a container, based on a sorting function specified in the definition. For example, this can be used to get all events active at a given time. Parameters:

  - *id*: (required) the *id* of the container to lookup in. Containers will generally iterate all child objects, so for example if you use the *id* of the top level score you should be looking up in to all sub-containers.

- *getFormattedLookup*: optional function that might be defined in an *io* script that outputs an object formatted for a different type of player/render. For example, this function might return a list of */x* and */y* values for use with the *o.lookup* Max object, or create a MIDI file export etc. All parameters included in the *val* object will be sent to the *getFormattedLookup* as a parameters object. Parameters:

  - *id*: (required)

- *call*: calls a function in the one or both of the class definitions. All of the parameters in the *val* object will be passed to the function as an argument. Return values from the *io* controller are with the tag *return/io* and *return/ui* from the *ui*.

  - *class* (required) class of the object to call
  - *method* (required) name of object function to call

```
{
    /key : "call",
    /val : {
        /function : "functionName",
        /class : "className",
        /id : "foo"
        /someValue : 1,
        /anotherValue : 2
    }
}
```

Note that the system will pass the same call to both definitions, so if both have a function of the same name they will both be called. * *drawsocket*: forwards DRAWSOCKET format messages directly to DRAWSOCKET, bypassing the symbolist mapping.

10. CASE STUDY

hello world!

11. CONCLUSIONS

Acknowledgments

12. REFERENCES

[1] World Wide Web Consortium, "Scalable Vector Graphics (SVG) 1.1 (Second Edition)," *W3C Candidate Recommendation*, 2011. [Online]. Available: https://www.w3.org/TR/SVG11/

[2] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[3] 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 (TENOR'15)*, Paris, France, 2015.

[4] 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 (ICMC'15)*, Denton, TX, USA, 2015.

[5] A. Freed, D. DeFilippo, R. Gottfried, J. MacCallum, J. Lubow, D. Razo, and D. Wessel, "o.io: a Unified Communications Framework for Music, Intermedia and Cloud Interaction." in *ICMC*, 2014.

[6] R. Gottfried and J. Bresson, "Symbolist: An open authoring environment for end-user symbolic notation," in *International Conference on Technologies for Music Notation and Representation (TENOR'18)*, 2018.

[7] G. E. Krasner, S. T. Pope *et al.*, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[8] R. Gottfried and G. Hajdu, "Drawsocket: A browser based system for networked score display," in *Proceedings of the International Conference on Technologies for Music Notation and Representation–TENOR 2019*. Melbourne, Australia: Monash University, 2019, pp. 15–25.

[9] G. Hajdu, "Quintet. net: An environment for composing and performing music on the internet," *Leonardo*, vol. 38, no. 1, pp. 23–30, 2005.

[10] D. Crockford, "Json," *ECMA International*, 2012.

[11] A. H. Guest, *Labanotation: The System of Analyzing and Recording Movement*. Routledge, 2005.

[12] T. J. West, A. Bachmayer, S. Bhagwati, J. Berzowska, and M. M. Wanderley, "The design of the body: suit: score, a full-body vibrotactile musical score," in *International Conference on Human-Computer Interaction*. Springer, 2019, pp. 70–89.

[13] J. Bresson, C. Agon, and G. Assayag, "OpenMusic: Visual Programming Environment for Music Composition, Analysis and Research," in *Proceedings of the ACM international conference on Multimedia – Open-Source Software Competition*, Scottsdale, AZ, USA, 2011, pp. 743–746.

[14] N. Didkovsky and G. Hajdu, "MaxScore: Music Notation in Max/MSP," in *Proceedings of the International Computer Music Conference (ICMC'08)*, Belfast, Northern Ireland / UK, 2008.

[15] A. Agostini and D. Ghisi, "A max library for musical notation and computer-aided composition," *Computer Music Journal*, vol. 39, no. 2, pp. 11–27, 2015.

[16] T. Baca, J. W. Oberholtzer, J. Treviño, and V. Adán, "Abjad: An open-source software system for formalized score control," in *2015. Proceedings of the First International Conference on Technologies for Music Notation and Representation-TENOR2015. Paris: Institut de Recherche en Musicologie*, 2015, pp. 162–169.

[17] G. Burloiu, A. Cont, and C. Poncelet, "A visual framework for dynamic mixed music notation," *Journal of New Music Research*, vol. 46, no. 1, pp. 54–73, 2017.