SYMBOLIST: REDESIGNED FOR DYNAMIC BIDIRECTIONAL MAPPING

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, 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) [1] 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 [2]

[3] [4]

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 [5]

2. JUCE VERSION

explain juce version (include in overview?) MVC [6] focused on creating SVG and outputting OSC version (like SVG-OSC approach)

3. 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

4. SYMBOLIST ELECTRON

using DRAWSOCKET as front end handler [7]

[8]

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.

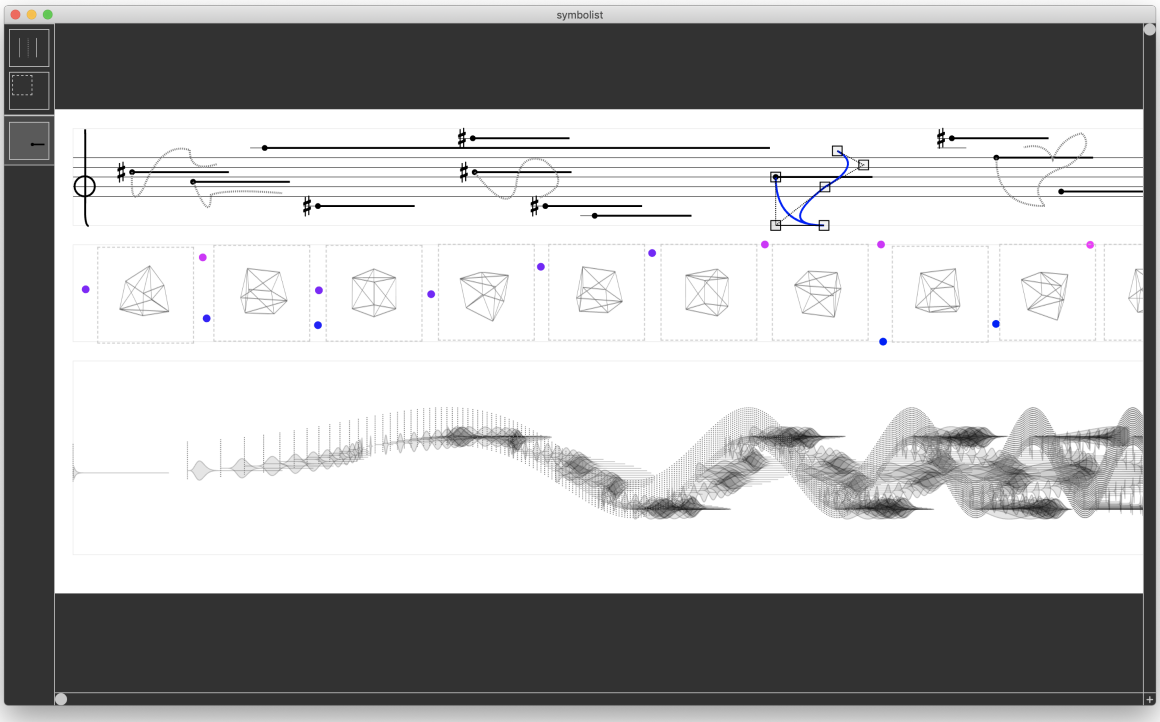

Figure 1. SYMBOLIST screenshot, showing some different types of staves, and editing capabilities.

4.1 Implementations

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

4.2 Contexts

There are three basic representation contexts at the core of SYMBOLIST:

1. **semantic data**, which specifies the various attributes of information about a symbolic object, in terms of the *meaning* to the author. The *semantic data* is the main holder of information in the system, which arranged as a score can function like a database of hierarchal information. For example, a note might contain information about pitch and duration, or a point in space might contain x, y, and z values.
2. **graphic representation**, the visual representation of the semantic data (see figure 2).
3. **performance media**, the performance mechanism which can be used to control different media types using the score data as parameter values.

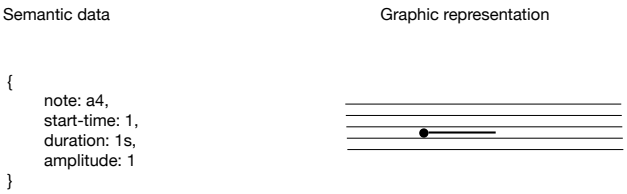

Figure 2. *data* vs *graphic* representation of the same information.

4.3 Mapping

Between each of these representation contexts there is a layer of mapping:

- *semantic data* to *graphic representation* is used for the creation of graphic symbols based on input of semantic data.
- *graphic representation* to *semantic data* is used to edit, or create new data entries, based on graphic information.
- *semantic data* to *performance media* 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).
- mapping between *performance media* and *graphic representation* is achieved through first mapping to semantic data.

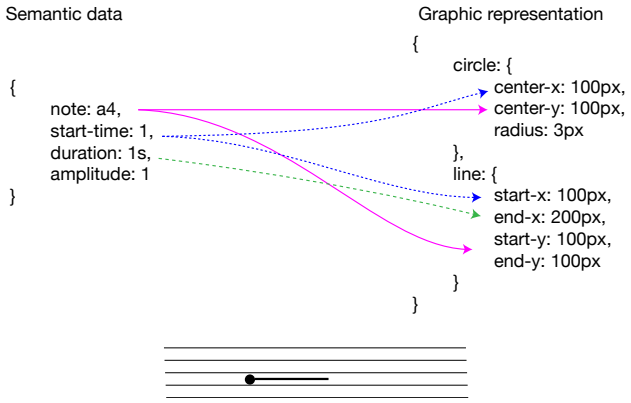

Figure 3. *semantic data* mapped to create a *graphic* representation from input data.

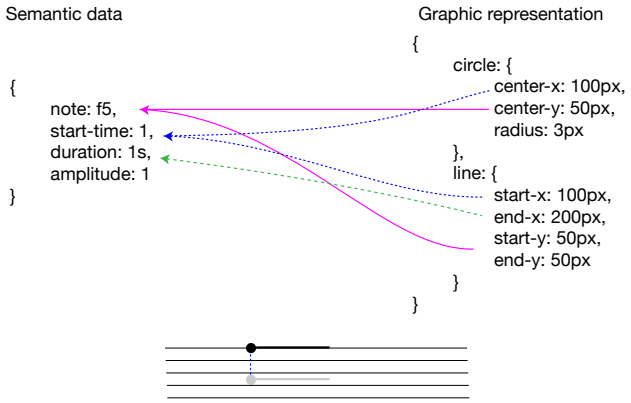

Figure 4. If edited graphically, the updated graphic data is then mapped back to *semantic data* representation.

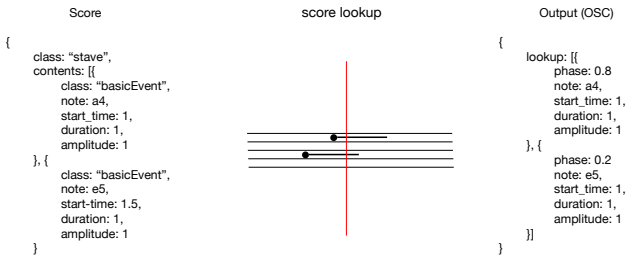

Figure 5. Using the lookup method defined by the symbol class, the *semantic data* can be used to perform external instruments via Open Sound Control.

4.4 Application Structure

The main structure of the platform is currently in three parts:

1. The *editor*, a browser-based graphic user interface which displays the graphic representation of the data, and allows the user to edit and create new data from graphic interaction. The editor loads a library of scripts that define mappings to and from data and graphics formats. The editor receives and outputs data in the semantic format, keeping the concerns of drawing within the browser-side.
2. The *server*, a node.js (or electron) based webserver which routes messages between the *editor* and the *io* system, and handles operating system commands like reading and writing files.
3. The *io-server*, which handles input and output from external sources via OSC. The *io-server* holds a copy of the score in its semantic format, and loads a parallel library of user scripts to the *editor* which define the mapping to (and potentially from) other media sources. The *io-server* might also be used to reformat the score into a format that can be performed by another sequencing tool or program like MaxMSP.

4.4.1 io and ui elements

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

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.

4.5 Symbols

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

Each *symbol* is a data object which holds a set of data parameters. For example a typical event like *symbol* might represent a note event, and contain *pitch*, *time* and *duration* parameters. The details of each symbol's data structure and UI interaction is defined in an object *class*.

CHECK IF CONTAINERS STILL EXIST, edit that out probably everything can be a container IN SCORE, CONTAINER IS USED TO IDENTIFY PARENT ELEMENT (but it's not a class)

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

In most cases, a symbol mapping definition will require querying the parent container symbol for information, to plot the data into the container frame's context.

All symbols are stored in container symbols. ¿¿ this is true but out of date, any symbol can have ”contents” ¡¡

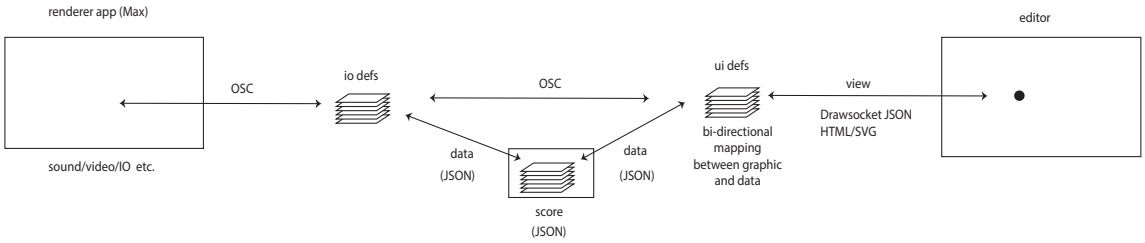

Figure 6. SYMBOLIST architecture.

4.6 Representation

Data is stored in JSON format.

The main object data attributes are:

- *id*: a unique identifier name.
- *class*: a reference to the definition of the object type in the user-definition library.
- *contents*: an array of other objects that a container object holds.

For example a simple *symbol* object 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.

Symbols can also be containers of other symbols, 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”,
        ”action” : ”sit”,
        ”startTime” : 0.2
    }]
}
```

4.7 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.

JSON files

Symbolist files are composed in the same way that the data model is stored. The top level symbol will be created in the ‘top-svg‘ symbol which is pre-defined in the ‘index.html‘ file.

An example initialization config file might look 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”,
    ”tools” : [],
    ”palette” : [”SubdivisionTool”, ”
        BasicSymbolGL”],
    ”class” : ”RootSymbol”,
    ”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” : []
        }]
    }
}
```

4.8 Graphic Display Format

The graphic representation of symbols is in SVG format, which is laid out in the ‘index.html‘ file. The DRAWSOCKET SVG/HTML/CSS wrapper is being used for convenience, to provide

a shorthand method of creating and manipulating browser window elements. 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.

Since 'symbolist' is constantly mapping to and from semantic data and its graphic representation, we are using the HTML *dataset* feature to store the semantic data inside the SVG object.

Typically, the 'id' attribute is used to quickly identify graphic objects, for the sake of clarity this is being left out of the examples below.

4.9 SVG Symbol Representation

The SVG SYMBOLIST format for a *symbol* is a one top group (<g>) elements, with two subgroups:

```
<g class="symbolClassName symbol" data-time
    ="0.1" data-duration="1">
  <g class='symbolClassName display'></g>
  <g class='symbolClassName contents'></g
      >
</g>
```

Each symbol grouping element is tagged using CSS class names, following the symbol's unique class name (in this example "symbolClassName"):

- *symbol* marks the top-level grouping object of the symbol
  - *display* a group that holds the visual display of the container symbol, and
  - *contents* which contains other symbols.

Note that the order is important: *the symbol class type must be first*.

The semantic data is also stored in the top-level symbol <g> element, using the HTML dataset feature, marked with the prefix 'data-'.

For example:

```
<g class='containerClassName symbol container'
    data-time='0.1' data-duration='1'>
  <g class='containerClassName display'>
    <line .... />
  </g>
  <g class='containerClassName contents'>
    <g class"symbolClassName symbol" data
      -time="0.1" data-duration="1">
      <circle .... />
    </g>
  </g>
</g>
```

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

```
<div class='containerClassName symbol container'>
  <div class='containerClassName display'></div>
  <div class='containerClassName contents'></div>
</div>
```

4.10 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.

4.11 Library Definitions API

MENTION CALL METHOD SYSTEM

discuss Template SymbolBaseClass system

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.

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

4.12 Editor

editor section is maybe not that necessary if we are mainly addressing the backend design? maybe could be too much information... not sure

but, the palette symbol system is interesting maybe, to discuss how you can "enter into" a symbol, and then the possible sub-elements populate the palette.

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.

See below for more information about the API for creating symbol definitions.

4.12.1 Interface Components

The SYMBOLIST graphic editor 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*: (not yet implemented in the current version) a set of interactive tools that provide ways of creating 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.

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” or ”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. CASE STUDY

hello world!

6. CONCLUSIONS

Acknowledgments

7. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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