Just a Sounding Object Notation: Sharing Objects for Sonic Interaction Design with JSON and OSC

Marco Tiraboschi
Lab. of Music Informatics (LIM)
University of Milan
Milan, Italy
0000-0001-5761-4837

Stefano Papetti
Inst. for Computer Music and Sound Technology
Zurich University of the Arts
Zurich, Switzerland
0000-0002-7490-3574

Federico Avanzini
Lab. of Music Informatics (LIM)
University of Milan
Milan, Italy
0000-0002-1257-5878

2023 4th International Symposium on the Internet of Sounds | 979-8-3503-8254-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/IEEECONF59510.2023.10335232

***Abstract*—We address the challenges encountered in sharing and communicating complex *sounding objects* in the field of Sonic Interaction Design. To overcome these challenges, we propose the adoption of JSON as a representation syntax and exchange format, accompanied by the integration of an Open Sound Control (OSC) server for enhanced interoperability and centralized control. By leveraging JSON, we provide a standardized method for capturing and sharing comprehensive information about sounding objects, streamlining the transition from prototyping to production software, and enabling efficient communication between analysis and synthesis tools. The integration of an OSC server further enhances real-time interoperability with other software platforms. To demonstrate the efficacy and viability of JSON and OSC as powerful tools for sharing objects in Sonic Interaction Design, this paper presents a practical use case that combines the SAMPLE software for modal analysis with the Sound Design Toolkit for real-time sound synthesis. Notably, SAMPLE generates JSON files that align with our proposed solution, highlighting the seamless compatibility and applicability of these technologies.**

***Index Terms*—sound synthesis, sounding objects, representation, exchange, communication protocol, OSC, JSON**

I. Introduction

Sonic interaction designers frequently find themselves navigating through a high-dimensional space, which is characterized by the large number of parameters present in the software tools they employ. While developing the *Sound Design Toolkit* [1], we realized that the very same high dimensionality that fuels the designer's creativity also poses a challenge due to two primary issues.

First, the lack of an established **exchange format** made it challenging to transition from complex multimedia programming environments, such as Pure Data [2] or Max [3], to production software, such as a VST plugin developed in C++ with JUCE or an XR experience created in C# with Unity. This also slowed down our work as developers, as we had to recreate example prototypes realized in, for instance, Max entirely anew for each distinct implementation of our software. The absence of a shared representation format uncovered the existing gap between sound analysis software, used for parameter estimation and fitting, and synthesis software.

Secondly, real-time interoperability with other software was complicated because it relied on platform-specific utilities. An effective **communication protocol** was missing.

In this paper, we introduce a solution that addresses the aforementioned issues by adopting JSON as a representation syntax and an exchange format for the parameters of virtual *sounding objects*. Additionally, we implemented an OSC [4] server to facilitate seamless interoperability and centralized control. Furthermore, we demonstrate a preliminary example of interoperability, wherein the SAMPLE [5] software for modal analysis already generates JSON files that adhere to the specifications of the Sound Design Toolkit.

II. Background

A. The Sound Design Toolkit

The Sound Design Toolkit (SDT) is a GPLv3 licensed open-source framework, serving as a virtual Foley-box for ecologically founded sound synthesis and design. Its capabilities encompass simulating diverse acoustic phenomena arising from solid interactions (e.g., collision, rubbing, rolling, scraping), liquids (e.g., dripping, streaming water), gasses (e.g., explosions, blowing wind), and machines (e.g., combustion engines, electric motors). Comprising physically informed sound synthesis models, audio processing algorithms, and analysis routines, the SDT primarily targets research and education in Sonic Interaction Design while also finding successful applications in musical contexts.

In particular, the solid resonator models in the SDT are based on the modal synthesis paradigm [6], which allows to define the frequency, gain, and decay time of each of their sinusoidal components. On top of that, the resonators also offer a physical interpretation in terms of vibrating objects, making available their vibration displacement and velocity at various "pickup" points. Such variables are used by the SDT's solid interaction models to compute interaction forces related, for instance, to collision or friction between objects.

Considering the inherent complexity of the sound synthesis techniques adopted, we made the decision to provide designers with a dedicated tool to effortlessly parameterize the SDT's resonator models through an ad hoc modal analysis tool, SAMPLE (see Section III-C).

B. JSON

JSON (JavaScript Object Notation) is a widely used data interchange format known for its simplicity, flexibility, and

ease of integration across various platforms and programming languages. It offers a convenient and efficient way to structure and transmit data between server and client applications.

JSON is built on two fundamental data structures: objects and arrays. These structures allow for a versatile representation of various data types and hierarchies. An object is defined within curly braces and consists of key-value pairs. Each key is a string enclosed in double quotation marks, followed by a colon, and its associated value. These values can encompass various data type, such as strings, numbers, arrays, or even nested objects. The key-value pairs are comma-separated within the braces. Arrays are enclosed in square brackets and can contain an ordered collection of values. The value inside JSON arrays may also encompass elements of various data types, including objects and nested arrays.

C. Open Sound Control

Open Sound Control (OSC) [4] is an application-layer protocol over either TCP or UDP, developed mainly for communication between computers and multimedia devices, such as digital musical instruments.

By specification, an OSC *client* is any application that sends OSC *packets*, the units of transmission of OSC. Any application that receives OSC packets is an OSC *server*. Every OSC server has a set of OSC *methods*: they are the potential destinations of OSC messages received by the OSC server, and correspond to the available points of control for the application. OSC methods are arranged in a tree structure called OSC *address space*. In this structure, the OSC methods represent the leaves, while the branch nodes are referred to as OSC containers. An OSC address space needs not to be static, but can vary over time. Every OSC method or container is identified by a symbolic name in the form of an ASCII string. The OSC *address* of an OSC method consists in a symbolic name that specifies the full path to the OSC method within the OSC address space. The path starts with the root symbol '/', and it includes the names of all the containers in pre-order, ending with the name of the OSC method. These names are separated by forward slashes. It is noteworthy that the syntax of OSC addresses conforms to the syntax used in URLs.

III. Design and Implementation

A. JSON

As a representation format for the SDT, we opted for JSON, a very common choice across many Web and IoT domains, such as the Internet of Sounds. In contrast, XML has become unpopular mostly due to its verbosity [7]. We integrated the JSON implementation by James McLaughlin [8], [9], which we also contributed to. This implementation fits perfectly our requirements due to its low-footprint and ANSI C compliance.

Any object implemented as a C `struct` in the SDT library is represented as a JSON object. This JSON object includes the SDT object's attributes as keys and their corresponding attribute values as values. For each of these attributes, the corresponding getter and setter function are called by the JSON dump and load functions, respectively.

Some of the SDT objects have multiple values for each attribute, which are represented as JSON arrays. Modal resonators, for example, have two attributes that express the size of other attributes: the number of modes and the number of pickup points. A `SDTResonator` structure holds one modal frequency, one decay time, and one modal weight value for each resonance mode to be simulated. Consequently, modal frequencies, decay times and modal weights are represented as three arrays. Since at each pickup point each mode has its own gain factor, modal gains are represented as a nested array. Listing 1 is an example JSON representation of a resonator.

Since users may want to call the load function at audio rate, every JSON load function has a *safety* flag. When this is turned off, all attributes can be set without restrictions. Otherwise, the JSON loader will avoid setting any attribute that would result in memory allocation or deallocation; instead, a warning will be displayed. Among the attributes requiring special care are the sizes of memory buffers for analysis objects, and the cardinality of array-valued attributes (e.g. the number of modes of a resonator). Functions in the SDT allow to serialize and deserialize JSON from and to either C-strings, to be used on-the-fly, or files, to be stored and exchanged.

In our context, a group of SDT objects is called a *project*. These may be, for instance, the SDT objects loaded in a Pure

```
{
  "nModes": 20,
  "nPickups": 1,
  "activeModes": 20,
  "fragmentSize": 1.0,
  "freqs": [
    8460.87, 8452.44, 2079.75, 7361.63,
    8467.28, 3678.45, 1065.26, 12572.1,
    1063.06, 9533.82, 5762.19, 12596.3,
    10555.4, 17511.9, 5769.25, 16935.2,
    16943.8, 10561.7, 2848.88, 2834.24
  ],
  "decays": [
    0.067, 0.079, 0.558, 0.103,
    0.106, 0.284, 1.100, 0.066,
    0.793, 0.078, 0.128, 0.125,
    0.101, 0.048, 0.130, 0.062,
    0.049, 0.197, 0.047, 0.100
  ],
  "weights": [
    1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0,
    1.0, 1.0, 1.0, 1.0
  ],
  "gains": [
    [
      1.39, 1.20, 0.23, 0.30,
      0.24, 0.08, 0.04, 0.13,
      0.03, 0.09, 0.06, 0.06,
      0.05, 0.07, 0.03, 0.04,
      0.04, 0.01, 0.03, 0.02
    ]
  ]
}
```

Listing 1: Example of a JSON file with parameters for a modal resonator estimated from a glass sound using SAMPLE. This JSON object has four scalar attributes (`nModes`, `nPickups`, `activeModes`, and `fragmentSize`), three array-valued attributes (`freqs`, `decays`, and `weights`) of size `nModes`, and `gains` is a nested array of `nPickups` of size `nModes`.

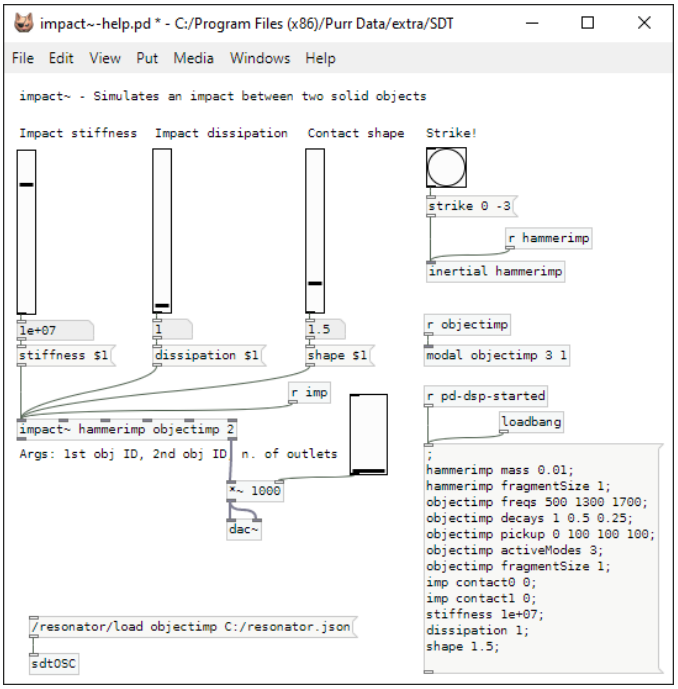

Fig. 1. Example of the Pure Data help patch for `impact~` objects, with the addition of the `sdtOSC` object and an OSC message to load the parameters of a modal resonator from a JSON file.

Data patch, or all the components needed for a VST plugin. A project is represented as a JSON object which holds one key-value pair for each SDT object type. The key identifies the type (e.g. `resonators`, `frictions`, `bubbles`, etc.), and the value is a JSON object itself. Inside this object, there is one key-value pair for each SDT object of that type in the project: the value is the JSON representation of a SDT object, the key is its identifier. Projects thus provide a convenient way to save and load the parameters for all the SDT objects of interest in bulk. For this to be possible, the objects need to be registered into a hash map. In Pure Data and in Max, we automatically register in the appropriate hash map all objects instantiated if their identifier argument is specified.

B. Open Sound Control

We made the decision to incorporate support for the Open Sound Control (OSC) protocol because of its extensive adoption and its ability to provide the flexibility required for our purposes. Avoiding unnecessary reinvention of the wheel, we chose not to include low-level networking functionalities in the SDT. For communication purposes, several projects implemented in Pure Data successfully rely on the networking and OSC libraries by Martin Peach [10] or on the low-level native objects (`netreceive` and `netsend`, `oscparse`, and `oscformat`). On the other hand, Max natively supports network I/O via UDP, simplifying the process for its users. Those opting for more traditional text-based programming languages have the freedom to choose their favorite networking library among the many available.

We added a new function to the SDT (`SDTOSC`), that routes and processes OSC messages. This function is accessible in both the Pure Data (Fig. 1) and the Max (Fig. 2) distribution of the SDT as a new object (`sdtOSC` and `sdt.OSC`, respectively). We designed the address space for the OSC server as follows. The top container is the name of the SDT object type (e.g. `resonator`, `friction`, `bubble`, etc.). For each type container there is one setter method for every attribute. Setter methods take as arguments the identifier of the object to be affected and the value to be set: for example, the message `/myo/threshold mx 0.001` sets to 0.001 the *threshold* attribute value of a myoelastic feature extractor object identified by the key *mx*. The attribute setters for array-valued attributes additionally take the index of the element to be set. All containers also have four methods for JSON input and output:

- `load` loads the object's parameters from a JSON file;
- `loads` loads the object's parameters from a JSON string;
- `save` saves the object's parameters to a JSON file;
- `log` logs the object's parameters as a JSON string.

An additional container exists that does not correspond to any object type: the *project* container is a specialized one that enables the simultaneous management of multiple objects. Its `load` and `loads` methods only take, respectively, the file path and the JSON string to be loaded, while its `save` and `log` methods take as arguments the identifiers of the objects to be included in the project. The `save` method also requires the path of the file to be saved as the first argument.

C. Interoperability Example

Thanks to the addition of JSON and OSC support, the SAMPLE [11] software for modal analysis has been now integrated in the SDT ecosystem. This is a software written in

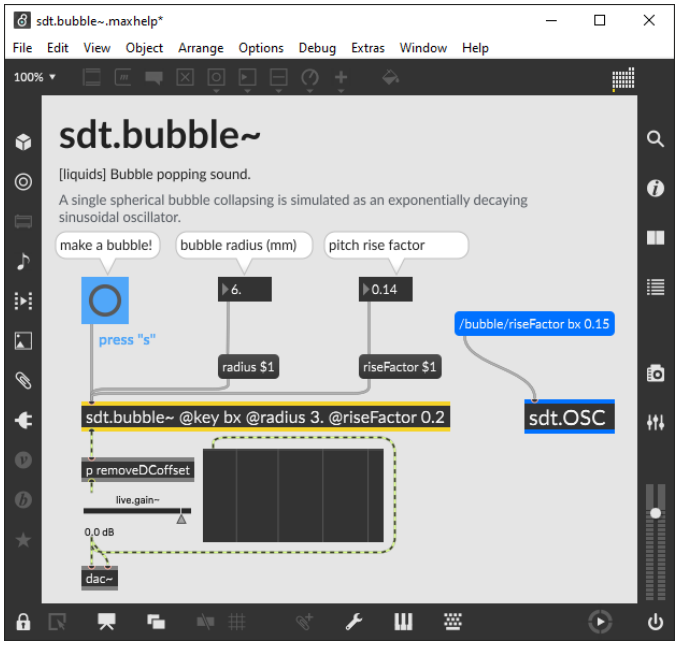

Fig. 2. Example of the Max help patcher for `sdt.bubble~` objects (in yellow), with the addition of the `sdt.OSC` object and an OSC message (both in blue) to set the "rise factor" parameter.

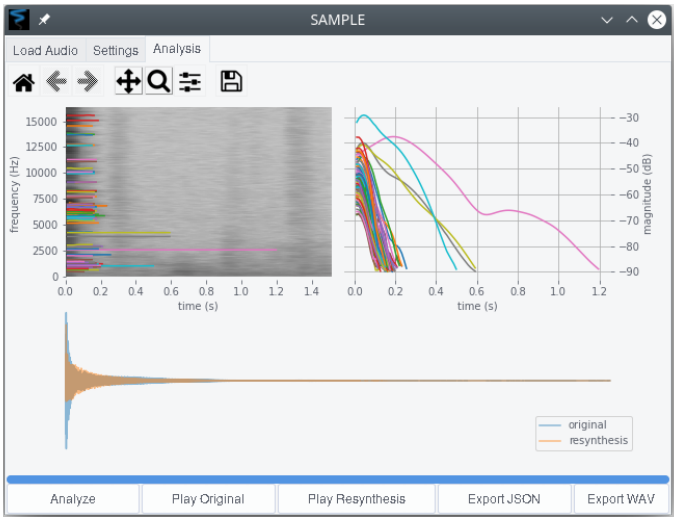

Fig. 3. The SAMPLE GUI. In the *"Analysis"* pane, users can run the algorithm, play an audio resynthesis, and export the estimated modal parameters as a JSON file for the SDT.

Python that implements the SAMPLE algorithm [12], available on open-source repositories such as Zenodo, GitHub and on the Python Package index [5]. SAMPLE can be used either through its Python API or via a Graphical User Interface (Fig. 3). The SAMPLE software can be used to analyze audio samples of modal resonator impulse responses, and its "JSON export" functionality allows users to save the estimated modal parameters to file. One example is displayed in Listing 1: please note that, because of the limitations of the analysis algorithm, currently the software can only estimate the parameters for one pickup point.

IV. Conclusions and Future work

In this paper, we tackled the challenges associated with sharing and communicating complex sounding objects in Sonic Interaction Design, particularly utilizing the Sound Design Toolkit (SDT). To overcome these challenges, we proposed the adoption of JSON as a representation syntax and exchange format, complemented by the integration of an Open Sound Control (OSC) server for enhanced interoperability and centralized control. JSON provides a standardized method for capturing and sharing comprehensive information about sounding objects, streamlining the transition from prototyping to production software, and facilitating efficient communication between analysis and synthesis tools. Furthermore, JSON's human-readable nature [13] allows end-users, including non-developers, to easily edit it according to their specific requirements. The integration of an OSC server further enhances real-time interoperability with various software platforms. Through a practical use case, we have demonstrated the efficacy and viability of JSON and OSC integration by combining the SAMPLE software for modal analysis with the SDT.

In the future, usability tests could be conducted to evaluate the impact of the JSON and OSC integration on the sound design workflow speed. Additionally, considering the flexibility of the MIDI 2.0 standard, which allows messages with JSON payload [14], adding support for such protocol could be explored as an alternative to OSC.

Overall, we are hopeful that the compatibility and applicability of JSON and OSC technologies showcased in this study pave the way for enhanced sound design processes and broader possibilities in Sonic Interaction Design.

Acknowledgment

We wish to acknowledge everyone else who was involved in the development and design of the Sound Design Toolkit over the years, including Stefano Baldan, Nicola Bernardini, Gianpaolo Borin, Carlo Drioli, Delphine Devallez, Federico Fontana, Laura Ottaviani, Pietro Polotti, Matthias Rath, and Stefania Serafin. In particular, we extend our deepest gratitude to Davide Rocchesso and Stefano Delle Monache for their leading contributions to the software and their invaluable feedback for this paper.

References

[1] S. Baldan, S. Delle Monache, and D. Rocchesso, "The Sound Design Toolkit," *SoftwareX*, vol. 6, pp. 255–260, 2017. [Online]. Available: https://doi.org/10.1016/j.softx.2017.06.003

[2] M. Puckette, "Pure data," in *Proceedings of the 1997 International Computer Music Conference*. Thessaloniki, Greece: Michigan Publishing, 1997. [Online]. Available: http://quod.lib.umich.edu/i/icmc/bbp2372.1997

[3] ——, "Combining event and signal processing in the MAX graphical programming environment," *Computer Music Journal*, vol. 15, no. 3, pp. 68–77, 1991. [Online]. Available: http://www.jstor.org/stable/3680767

[4] M. Wright and A. Freed, "Open sound control: A new protocol for communicating with sound synthesizers," in *Proceedings of the 1997 International Computer Music Conference*. Thessaloniki, Greece: Michigan Publishing, 1997. [Online]. Available: http://quod.lib.umich.edu/i/icmc/bbp2372.1997

[5] M. Tiraboschi, "SAMPLE – Python package," LIM, University of Milan, 2021. [Online]. Available: https://doi.org/10.5281/zenodo.6536419

[6] J. M. Adrien, "The missing link: Modal synthesis," in *Represent. Music. Signals*, G. De Poli, A. Piccialli, and C. Roads, Eds. Cambridge, MA: MIT Press, 1991, pp. 269–297.

[7] L. Turchet *et al.*, "The internet of sounds: Convergent trends, insights, and future directions," *IEEE Internet of Things Journal*, vol. 10, no. 13, pp. 11 264–11 292, 2023. [Online]. Available: https://doi.org/10.1109/JIOT.2023.3253602

[8] J. McLaughlin *et al.* and M. Tiraboschi, "json-parser – Very low footprint DOM-style JSON parser written in portable ANSI C." [Online]. Available: https://github.com/json-parser/json-parser

[9] J. McLaughlin, J. De Witt, M. Meulemans, and M. Tiraboschi, "json-builder – the serializing counterpart to json-parser." [Online]. Available: https://github.com/json-parser/json-builder

[10] R. Haefeli, "netpd - a collaborative realtime networked music making environment written in pure data," in *Linux Audio Conference 2013*, vol. 1. Institute of Electronic Music and Acoustics. University for Music and Performing Arts Graz, Austria, 2013.

[11] M. Tiraboschi and F. Avanzini, "SAMPLE: a Python package for the spectral analysis of modal sounds," in *Physical bodies – Proceedings of the 23rd Colloquium on Music Informatics*, 2022, pp. 50–55.

[12] M. Tiraboschi, F. Avanzini, and S. Ntalampiras, "Spectral Analysis for Modal Parameters Linear Estimate," in *Proceedings of the 17th Sound and Music Computing Conference*, Torino, Italy, 6 2020, pp. 276–283. [Online]. Available: https://doi.org/10.5281/zenodo.3898795

[13] P. Wehner, C. Piberger, and D. Göhringer, "Using json to manage communication between services in the internet of things," in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2014, pp. 1–4. [Online]. Available: https://doi.org/10.1109/ReCoSoC.2014.6861361

[14] F. Avanzini, V. Faschi, and L. A. Ludovico, "A web-based midi 2.0 monitor," in *Proceedings of the 20th Sound and Music Computing Conference*, Stockholm, Sweden, 2023, pp. 148–153.