Creating, Visualizing, and Analyzing Dynamic Music Objects in the Browser with the Dymo Designer

THALMANN, FS; FAZEKAS,; WIGGINS,; SANDLER,; Audio Mostly 2016

Creating, Visualizing, and Analyzing Dynamic Music Objects in the Browser with the Dymo Designer

Florian Thalmann, György Fazekas, Geraint A. Wiggins, Mark B. Sandler
Centre for Digital Music
Queen Mary University of London
f.thalmann@qmul.ac.uk

ABSTRACT

Dynamic music is gaining increasing popularity outside of its initial environment, the videogame industry, and is gradually becoming an autonomous medium. Responsible for this is doubtlessly the prevalence of integrated multisensory platforms such as smartphones as well as the omnipresence of the internet as a provider of content on demand. The music format Dynamic Music Objects builds on these assumptions and on recent advances in music information retrieval and semantic web technologies. It is capable of describing a multitude of adaptive, interactive, and immersive musical experiences. This paper introduces the Dymo Designer, a prototypical web app that allows people to create and analyze Dynamic Music Objects in a visual, interactive, and computer-assisted way.

CCS Concepts

•Information systems → Music retrieval; •Human-centered computing → Interaction design process and methods; Mobile devices; •Applied computing → Sound and music computing; •Hardware → Sensor devices and platforms;

Keywords

semantic audio; dynamic music; interaction design; adaptive music; interactive music; music visualization; mobile music; mobile devices; semantic web; music information retrieval; web audio api

1. INTRODUCTION

Having remained within the world of computer games for a long time, dynamic music is gradually emancipating itself from its initial environment [3]. Within the last five years, increasingly many artists have released dynamic musical experiences that include interactive, adaptive, and dynamic aspects. They were primarily released in the form of mobile apps which are often closed-source and custom-made by developer teams in close collaboration with the artists.[1]

Without the help of computer professionals, designing such experiences can be tedious and may require significant training in computer programming and electronic music. In the case of RjDj[2], for instance, the most sophisticated and versatile environment for mobile musical experiences to date, users need to learn Pure Data, a visual programming language for computer music. Other solutions are inspired by digital audio workstations and studio software and typically offer less dynamic possibilities.[3] The most advanced of these solutions, *fmod studio*[4], specializes on adaptive computer game sound and music, presupposes knowledge in audio production, and is arguably limited by a thinking in multitrack recording and loop-based rendering.

In this paper, we introduce the *Dymo Designer*, a web application that enables non-expert users to design dynamic mobile music experiences in a graphical and interactive way, while being assisted by the semantic web and the audio analysis techniques. At its first stage of implementation, the app provides functionality for automatically defining a variety of types of *Dynamic Music Objects (Dymos)* [14]. Dymos consist of a set of audio files, a semantically annotated structural description, and a rendering instruction that describes a scheduling and interaction scheme. While creating dymos so far consisted in manually writing semantic web files, annotating each object with analytical values, and defining complex mapping functions, the purpose of the Dymo Designer is to automate and simplify this process. Users can simply input a number of audio files, which the system analyzes and organizes automatically, depending on the users' needs, and finally generates dymos following given standard templates. The users can also choose to gain more control over certain steps of this process and they can visualize and browse the created dymos and mappings in various ways and modify them interactively. Finally, they can preview how a dynamic music player will render and navigate the ob-

AM '16, October 04 - 06, 2016, Norrköping, Sweden

ISBN 978-1-4503-4822-5/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2986416.2986445

[1] e.g. *Love* by Air (https://www.youtube.com/watch?v=zklBcmqbNsM), created with RjDj (see below), *The National Mall* by Bluebrain (https://itunes.apple.com/gb/app/national-mall-by-bluebrain/id437754072), *Fantom Sensory Music* by Massive Attack (https://itunes.apple.com/us/app/fantom-sensory-music/id1062360670)

[2] http://rjdj.me

[3] For example, Weav (http://weav.io) is currently based on just one variable music parameter, tempo, and ensures that the musical material sounds satisfying at any value. Spotify recently built an equivalent format (https://www.spotify.com/running/).

[4] http://www.fmod.org

jects and how various types of controls available on a mobile device will affect the musical outcome. We will first summarize the concepts introduced in earlier papers, especially Dynamic Music Objects, and then go through the current functionality of the *Dymo Designer* and illustrate it using various examples.

2. DYNAMIC MUSIC AND DYNAMIC MUSIC OBJECTS

Dynamic music is often used as an overarching term for both adaptive and interactive music [3]. *Adaptive music* indirectly adapts to the listener's context and is typically controlled by parameters describing the listener's state, whereas *interactive music* depends on direct user interaction where the listener has noticeable control over musical parameters. However, it is often overlooked that dynamic music can also include *autonomously dynamic* aspects, where certain musical characteristics are malleable and can change from one listening experience to the next, without any interference from the listener. In practice, dynamic music often integrates all three of these types of dynamicity, leading to a highly complex interaction scheme that is typically not entirely comprehensible by the listener, which has been shown to be desirable and which without a doubt keeps the musical experience more interesting in the long run [11, p. 235/255].

Both in the music industry and the videogame industry, multiple formats and standards for dynamic music have been introduced in the past, each of them focusing on various aspects and platforms. *RjDj* and *iXMF*[5] are highly versatile but the former is uniquely aimed at mobile app development and does not support any kind of mobile sensor in a platform independent way, whereas the latter was directed at the gaming industry and has been discontinued. Object-based audio models such as *IM AF* [10] and the *Audio Definition Model* [1] encapsulate audio files and XML metadata, the former being largely limited to basic mixing parameters (amplitude and panning) and the latter being specialized in platform-adaptive spatial rendering. None of these formats use linked data to represent the metadata, which offers the significant advantages that one can directly build on other specifications, unambiguously specify the meaning of and the relationships between the concepts used within the format, and draw information from various knowledge bases.

More recently, some of these limitations were addressed with the introduction of the concept of a *Digital Music Object (DMO)*, which unites various representations of musical content in a bundle of music files for research, composition, production, or consumption.[4].[6] *Dynamic Music Objects (Dymos)*, in turn, are a special kind of DMO oriented towards the intermediary or end consumer and designed to be musically malleable and flexible with precisely defined degrees of freedom and constraints. They can be dynamic in all of the three ways discussed above, i.e. adaptive, interactive, or autonomously dynamic.

Dymos currently consist of a number of linked *audio files* which can be stored locally or in a distributed environment. A *structural definition* specifies the relationships between these audio files by defining a number of objects that refer to these files and can be annotated with *musical features and metadata* and contain a number of modifiable *musical parameters*. A playback configuration called *rendering* then describes how various types of controls map to the objects and their parameters. Both the structural definition and the rendering can be specified in OWL/RDF[7] by referring to concepts defined in the *Mobile Audio Ontology* [14] and other musical ontologies. Using SPARQL[8] or graph query algorithms the structure can then be queried and navigated in particular ways by a player for which a prototype is currently being developed, the *Semantic Music Player* [15].

2.1 The Structural Definition

At the basis of the structural definition is a new implementation of CHARM, an abstract music representation system allowing multiple hierarchies (meronomies) of musical objects or events related by arbitrary logical formulae and abstracted from concrete applications [9, 8]. Each object can have an arbitrary number of *parts* and a *type* that determines the relationship of the parts to each other and their parent.[9] It can also have a number of musical attributes which have a hierarchy of types themselves. The two most basic types of attributes are features and parameters. We define *features* as immutable analytical values directly extracted from the audio or gathered from various other representations, such as for instance the Vamp Ontology or the Audio Feature Ontology [5]. *Parameters*, on the other hand, are modifiable values that describe aspects of the music that can be changed. In addition to these hierarchies we can describe any kind of relationship between any of the objects in the structure, for instance *similarity relations* between different segments of a temporal structure.

Figure 1 shows a highly simplified example of a dymo structure representing a simple two-track mix that exposes parameters on various hierarchical levels.[10] A main object named `mix` has two parts which are played simultaneously (conjunction). The `Amplitude` parameter, for instance, is made accessible on several hierarchical levels, in all three objects. Depending on the functional dependencies we define between parameters, their modifications can have different effects. In this case, we define that changing the amplitude of the `mix` object affects the overall amplitude, whereas changing the one of `track1` or `track2` affects these tracks separately. Each object's amplitude is the product of its parent amplitudes. With the `Pan` parameter, in turn, it makes more sense to define the resulting pan as the sum of an object's and its parents' panning values. The example also shows how music files can be linked from dymos. Here, only the objects without parts refer to an audio file. If one of them is played back, we hear the respective file alone, whereas a playback of the `mix` object results in a simulta-

[5]http://www.iasig.org/wg/ixwg/

[6]It was defined in analogy to the Research Object [2] which includes exact specifications of experimental procedures along with the input or resulting data.

[7]http://www.w3.org/TR/owl2-overview/

[8]http://www.w3.org/TR/sparql11-query/

[9]The most basic such types are conjunction, disjunction, and sequence, but these types can get arbitrarily complex and describe more specific musical entities.

[10]The definitions in the figure are formulated in (pseudo-)Turtle, a textual syntax for OWL ontologies (www.w3.org/TR/turtle/). The `a` keyword refers to the `rdf:type` property, which defines instances of OWL classes, written in capitalized words. For instance, `Dymo` is a class and so are all parameters, such as `Amplitude` or `Reverb`. All properties, in turn, are written starting with a lowercase letter, such as `part` or `parameter`.

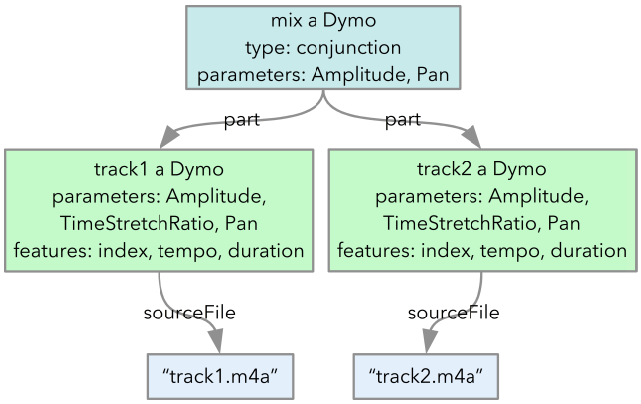

Figure 1: A sample Dynamic Music Object.

neous playback of the two files.

2.2 Mappings, Renderings, and Higher-Level Parameters

Once the structural definition of a dymo is specified we can turn to defining the ways in which a player will modify and navigate the structure. What makes a dymo dynamic is the fact that it has modifiable parameters. It is the purpose of *mappings* to define how these parameters can be modified during playback. Mappings are currently based on arbitrary JavaScript functions which can have features, controls, and parameters as arguments and an arbitrary function body. They also specify which objects are reached, either using another JavaScript function that defines a set of constraints on the dymos' attributes, or by simply referencing the target objects' uris.

We distinguish between two types of mappings, ones that define *higher-level parameters* by mapping from any parameters and features of a dymo to parameters of any of its parts, and ones that define a *rendering* by mapping from any of the available mobile controls to the defined higher- and lower-level parameters of a dymo. The controls currently available in the ontology and the Semantic Music Player include *sensor controls* (accelerometer, compass, geolocation, etc), *UI controls* (sliders, buttons, toggles, etc), and *autonomous controls* (ramps, statistical controls, AI). The former two allow users to influence the music or interact with it, whereas the latter let the player take decisions on its own. In the future, we will also add *contextual controls* which are based on contextual information gathered from the web, such as user preferences, trends, or weather information.

Figure 2 shows an example of how we can use mappings to achieve automatic mixing between two conjunctive tracks. We define a higher-level `Fade` parameter via a function that interpolates between the amplitudes of the two tracks by using the `index` feature of the tracks.[11]

$$Fade \times index \rightarrow Amplitude$$
$$(a, b) \mapsto (1 - b) * (1 - a) + b * a$$

For illustration, in JSON-LD we write the following:[12]

[11]The index feature denotes the index of a part in a conjunction or a sequence.

[12]The `dymos` function selects the subset of dymos to by mapped to (here all parts of the main dymo, i.e. on hierarchical level 1)

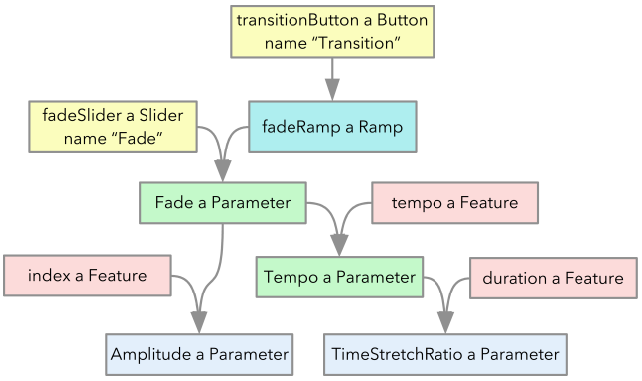

Figure 2: A sample rendering. The mappings are represented as arrows with multiple origins for the domains.

```
{
   "domainDims":[
      {"name":"Fade","@type":"Parameter"},
      {"name":"index","@type":"Feature"}],
   "function":{"args":["a", "b"],
      "body":"return (1-b)*(1-a)+b*a;"},
   "targets":{"args":["d"],
      "body":"return d.getLevel() == 1;"},
   "range":"Amplitude"
}
```

Listing 1: A mapping defining an amplitude cross-fade serialized in JSON-LD.

Further, we smoothen the local tempo of both tracks, by directly mapping the `duration` feature of each bar or beat to the `TimeStretchRatio` parameter. Simultaneously, we define a higher-level `Tempo` parameter which we normalize so that we can express it in beats per minute:

$$Tempo \times duration \rightarrow TimeStretchRatio$$
$$(a, b) \mapsto a/60 * b$$

We then map the `Fade` parameter defined above to the `Tempo` parameter in relation to a `tempo` feature (containing the tempos of both tracks) so that simply by changing the `Fade` parameter we can interpolate both amplitude and tempo:

$$Fade \times tempo \rightarrow Tempo$$
$$(a, b) \mapsto b[0] + a * (b[1] - b[0])$$

Finally, we can create a rendering which simply consists of an identity mapping from any type of mobile control, e.g. a slider, to the `Fade` parameter just defined.

3. THE DYMO DESIGNER

Dymo specification files easily get extensive and complex as soon as we attempt to represent more intricate musical structures, especially ones based on analytical findings, i.e. structured based on features and annotated with features. Furthermore, even for simpler examples such as the ones shown above, defining use cases in Turtle [14] or JSON-LD (above, or in [15]) can be rather tedious or confusing for

users that have no experience with ontologies. We thus not only need a tool that helps us generate these structures automatically based on the analysis of the audio files, but also inspect what was generated, make modifications, and extend the definitions intuitively.

This is why we decided to develop the *Dymo Designer*, a browser-based application that visualizes dymos in flexible ways and provides a simple and intuitive interface for automatically building musical structures and adding features to it, as well as defining all types of mappings and renderings introduced above. All this can be done visually and interactively, for example mapping functions can be drawn onto the screen, either one-dimensional ones as simple graphs, or multi-dimensional ones as areas in multi-dimensional space. Finally, the musical result can be previewed audibly and visually using mock controls.

3.1 Architecture and Technologies

The Dymo Designer builds on several npm[13] and Bower[14] packages defined in the context of the Semantic Music Player framework. The central part is the *dymo-core* package, which allows loading and saving Dynamic Music Objects from their JSON-LD or Turtle representations, manages an internal representation in the form of a graph store that can be queried and reasoned upon, as well as an object-oriented bidirectional observer-pattern-based representation of the mappings between controls, features, and parameters. A *dymo-generator* package administers bundling the audio, extracting features, and loading features from their ontological representation (see below). It supports the automatic generation of various standard types of feature-based musical structures. A *music-visualization* package offers a multitude of standard visualizations of graph-based musical structures implemented using D3[15] and AngularJS[16]. The main *Dymo Designer* app draws all other packages together into an AngularJS web front end that defines all user interaction.[17]

3.2 The Main Interface

The graphical user interface of the Dymo Designer prototype consists of a large visualization area, which can represent various aspects of the Dynamic Music Object currently being worked on. Most of the functionality can directly be accessed by interacting with that area and what is being displayed. The area above the view offers UI elements with various more general purposes, such as switching application mode and, depending on the mode, specifying file folders, configuring the view, and so on.

At the current stage, the functionality of the application is divided into four modes or *activities*. In each of these modes, a different set of UI elements appears in the area above the visualization area. In *Dymo Mode*, users can create dymos by importing audio files and either importing feature files or specifying which features to be extracted. Then, they can define how these files are combined into any of the available standard dymo types. They can also directly interact with the view surface and add dymos and `part` relations, represented by nodes and edges, respectively. *Mapping Mode* allows users to define mappings in a similar way, by selecting a set of dymos as targets and then specifying domain and range dimensions from the available controls, features, and parameters. Then, they can either enter an arbitrary javascript mapping function into a text field or draw a function onto the view area. *Visualization Mode* offers UI elements for customizing the view, specifying which object classes or types and which of their relationships are displayed, and how visual characteristics such as size and color are assigned to any of the objects' datatype properties. Finally, *Rendering Mode* brings up mock controls for any UI and sensor controls so that their effects on the music can be tested and experimented with. Optionally, users can also visualize the state of auto controls, e.g. as moving sliders, and similarly any of the modifiable musical parameters present in the dymo structure. The following sections describe the functionality underlying these modes and the associated technologies in greater detail.

3.3 Creating Dymos

In *Dymo Mode*, users can create dymos either manually or automatically by letting the system generate them based on features extracted from the audio. For the manual option, they can draw nodes onto the view surface, assign audio files to them, and define their dependencies, which are visualized as edges. This works particularly well for dymos with a simple structure such as multitrack- or loop-based objects that do not contain any feature information, such as the one depicted in Figure 1. By right-clicking on nodes or edges, users can inspect and modify the objects' parameter values and properties as well as their relations.

The automatic option is better suited for more complex structures with a lot of audio material. One can choose between a number of standard types supported by the *dymo-generator* package based on any given number of organized audio files. The objects generated this way can then be visualized, reorganized, and customized in the same way as manually created ones. Similar methods exist for creating feature-based dymos, the structure and semantic annotation of which is determined by analytical information extracted from the involved audio files. These methods are described in the next sections.

3.3.1 Organizing the Audio and Extracting Features

At the current stage of implementation, users can choose to extract audio features on their own, using the *Sonic Annotator* software[18], or any other feature extraction software that is able to output features as linked data files based on the Audio Feature Ontology (AFO)[19]. The AFO links the features to other musical ontologies telling us about the semantics of the musical content, which is crucial for the Dymo Designer to understand the features' underlying musical concepts. Alternatively, this process can be automated by letting the Node.js application server call its own feature extractor, so that any uploaded audio file is automatically analyzed.

[13] https://www.npmjs.com
[14] http://bower.io
[15] https://d3js.org
[16] https://angularjs.org
[17] The reason for dividing everything into packages is that these packages can also be used by other applications. For instance, the Semantic Music Player merely uses dymo-core to load and play back dymos. On the other hand, the dymo-server, whose purpose it is to generate dymos from a large feature database, only needs the dymo-generator package.

[18] http://www.vamp-plugins.org/sonic-annotator/
[19] http://w3id.org/afo/onto/1.1#

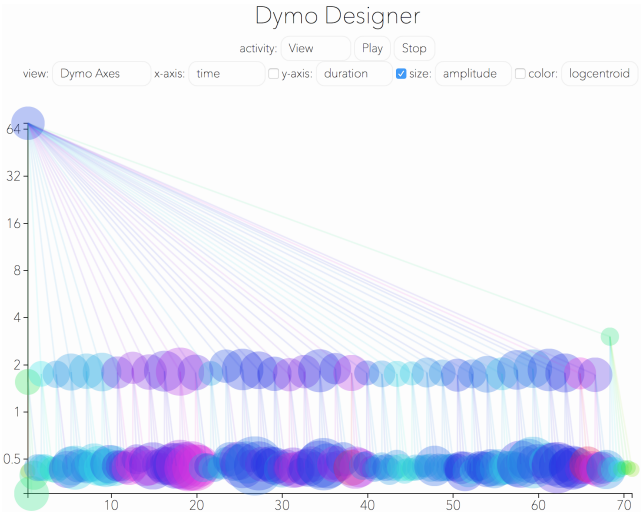

Figure 3: A visualization of a dymo with multiple levels of segmentation (bars and beats) in Dymo Designer. Here, the y-axis represents the duration of each object, size the amplitude, and color the log spectral centroid. The object on the top represents the entire piece with duration 64, its bars have a duration of about 2, and each of their beats 0.5.

3.3.2 *From Features to Dymos*

The audio and the analytical information obtained as just described can now be used to generate various kinds of dymos, depending on the types of features available. With structural features such as segmentations of various granularities, for instance, we can create a hierarchical dymo, where each segment contains its subsegments as parts. In Section 2.2 we saw how such an object, a dymo segmented into bars which in turn are segmented into beats, can be used in practice. Figure 3 shows how such an object looks like in Dymo Designer.

Other features can then be used to annotate the structure generated this way, whereby a summarization method is used for the duration of each audio segment. For instance, if a loaded feature file contains a high-resolution signal for the entire duration of the analyzed audio file, users can choose to annotate each segment dymo in the hierarchy with either the average, median, or initial value of the feature for the segment's temporal interval. For multidimensional features, such as chromagrams, summarized values are calculated for each component.

The dymo-generator package also provides some functionality to automatically infer similarity relationships from any information available about a given dymo. We can select any set of parts that are annotated with a number of summarized features as just described, for instance all beat dymos of a given dymo. Then, the app creates feature vectors containing a number of specified features common to all selected objects, normalizes them, and calculates their pairwise similarity, e.g. using a cosine distance as suggested in [6], and finally adds similarity relations to the pairs where the resulting value is above a given threshold (see [15] for details). Figure 4 shows a visualization of similarity relations obtained in this way for a dymo divided into irregular temporal sections.

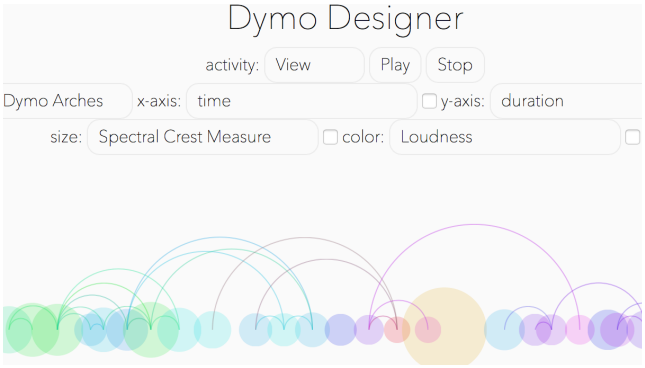

Figure 4: A visualization of similarity relations between segments of a dymo.

3.4 Creating Mappings

After building a dymo as just described, one can switch to *Mapping Mode* to start specifying the relationships between the parameters and features of any dymos in the hierarchy and defining custom higher-level parameters in a similarly interactive way. One can also add any of the controls available in the ontology and define a rendering by adding mappings from these controls to parameters.

3.4.1 *Defining a Mapping Target*

The first step of defining a mapping in Dymo Designer is the selection of a mapping target, the set of dymos affected by the mapping.[20] There are two ways in which users can do this.

Selecting Dymos.

They can simply select a set of dymos by dragging the mouse around them or clicking on them, whereby they can switch back and forth between arbitrary visualizations (see Section 3.5). When defining targets this way, the resulting RDF consists of a `toTarget` triple for each target. In other words, the set of targets is defined explicitly, referencing each target dymo by its uri. This way one can reach an arbitrary set of dymos, independently of any of their properties.

Defining A New Type.

On the other hand, if the set of target dymos can be defined implicitly through rules, for instance all dymos on a certain level of the hierarchy or all the ones that have a datatype property with a value within a certain range, there is a more convenient approach. The users can define a certain set of constraints or rules in the form of a JavaScript function (see Listing 1 above). The Dymo Designer will then add a new OWL class corresponding to these rules (if they can be expressed in OWL), so that the system can infer via reasoning at any time, which dymos belong to the class and which do not.

3.4.2 *Defining Renderings and Interaction Schemes*

Once the target dymos are selected, the users can switch

[20] When defining mappings in RDF, their target can also be a set of controls, which can have parameters as well. Currently, the Dymo Designer does not support the definition of such mappings. However, this functionality will be added in future versions.

to *Mapping Mode*, where they see a visualization of all available parameters common to all dymos they selected as a target, as well as all available controls. They can start defining mappings by right-clicking on the view surface, upon which a node appears, which they can connect to an arbitrary number of controls and one parameter.[21]

Defining Mapping Functions.

So far, the users selected domain and codomain, as well as the target dymos. What now remains to be defined is the mapping function itself. Analogous to selecting the set of target dymos, there are several ways in which the mapping function can be defined. As a first option, the users can write the body of an arbitrary JavaScript function into a text field above the viewing area. In the case of the function defined in the JSON-LD example above (Listing 1), the function body would simply be `return (1-b)*(1-a)+b*a;`. The variables used in the body correspond to the domain dimensions in their added order.

Alternatively, users can select from a number of standard function presets (linear, triangle, rectangle, etc), upon which the text field obtains an automatically generated code, which they can then edit. For example, a simple triangle function with domain and codomain in $[0, 1]$ looks as follows:

```
return 0.5-Math.abs(0.5-a);
```

As a third alternative, for simple two-dimensional functions, users can define them in a visual way. From two drop-down lists, they can select the domain and codomain dimension to be displayed on a two-dimensional coordinate system and then simply start drawing any valid function shape (for each x only one y). The Dymo Designer then samples a number of points from the drawn shape and interpolates all function values in between.

Defining Spaces Using Functions.

At a later stage, the Dymo Designer will enable users to draw more complex functional patterns. In the current prototype, one such functionality is already available. One can select two domain dimensions and draw polygons, which will be translated into multidimensional distribution functions, so that the function is zero outside the polygon and 1 in its centroid. Many such functions can then be visualized in the same space, as shown in Figure 5. Such constellations result in an abstract space that can then be navigated using various techniques (see [15] for example use cases).

3.4.3 Adding High-Level Parameters

Such spaces can be kept entirely abstract and thus flexible by not directly defining mappings from controls, but by defining new higher-level parameters for a parent dymo, as described in Section 2.2. This can also be done in Dymo Designer. After selecting the target dymos (Section 3.4.1), the app optionally shows all dymos that are parent dymos of all selected targets. The users can then click on the one they choose to use, and when they switch to *Mapping Mode*, they see that parent dymo's new parameter as well as all of its features (in addition to the target dymo parameters), all of which they can then use to define mappings in exactly the same way as they would for control mappings. These higher-level parameters can then be used in different ways in renderings, e.g. both a slider and a geolocation sensor can be mapped to it.

[21] Currently, the codomain of a mapping can only include one parameter, which directly enables one-to-one and many-to-one mappings. Yet, many-to-many mappings can be emulated by defining a mapping for each codomain dimension.

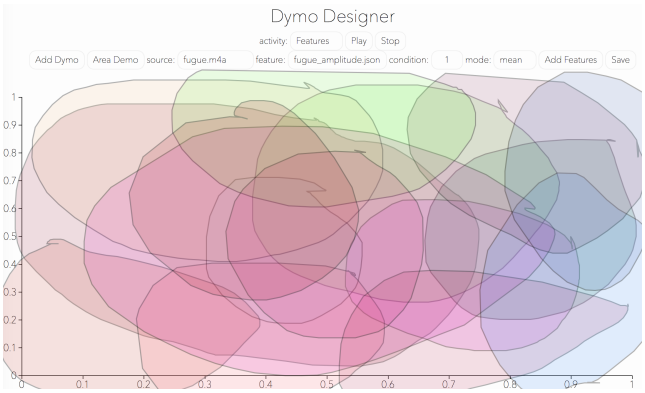

Figure 5: A multidimensional spatial arrangement of polygonal distribution functions. In this case, each polygon is mapped to the amplitude parameter of a dymo.

3.5 Visualizing Dymos and Mappings

We have already encountered some images that show the capabilities of the visualization system of the Dymo Designer. The system is designed to be modular, which means that any visualization code written using D3 that suits the structure of dymos and mappings (or object and graph visualization in general) can be plugged into the system. In the course of the development of the prototype we implemented a few standard visualization methods and bundled them in the *music-visualization* package. In the Dymo Designer users can choose between these preset options as well as define their own custom visualization based on selected OWL object classes and datatype properties.

3.5.1 The Music Visualization Package

The preliminary visualization schemes implemented so far are all based on standard music and graph visualization methods [12, 7, 13]. With all of them, users can assign the characteristics of the visual objects to properties of the objects being visualized, which are typically dymos or parameters. For instance, when visualizing dymos, one might choose to visualize the dymos' durations using the visual objects' `sizes`, their loudness with `opacity`, and their timbral characteristics such as the spectral crest with `color`.

Specifically, the current package includes four presets, all of them using a varying number of coordinate axes, different object types, and different variable visual characteristics.

Axes.

This preset shows any type of object as bubbles (circles) on a two-dimensional coordinate system. Any attribute or numeric datatype property of the visualized objects can be assigned to either of the two axes, as well as the size and color of the bubbles. Any relation or object property can be shown as connecting lines between the bubbles. The hierarchical dymo shown in Figure 3 is visualized in this way, where the x-axis depicts the onset of each segment, the y-

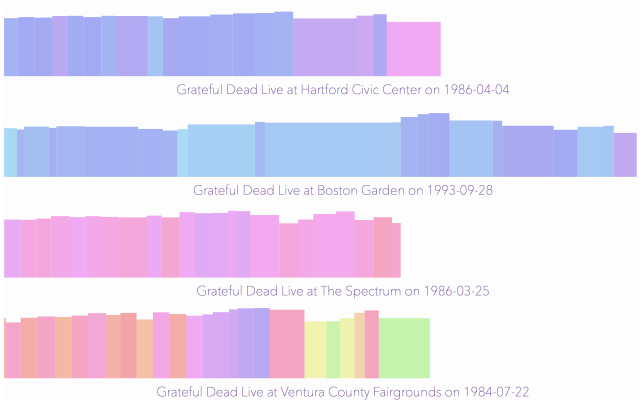

Figure 6: A juxtaposition of various dymos representing segmentations of various recorded live versions of the song *Throwing Stones*. X-axis represents onset, width duration, height loudness, and color timbre.

axis its duration, the size its amplitude, and the color its log spectral centroid. The lines, in turn, show the dymo's `part` relation.

Arcs.

This preset consists of the same kinds of bubbles as in *Axes*, however, depicted along a horizontal axis with any relation shown as arced lines. Figure 4 is based on this preset, where the size of the bubbles show the corresponding object's spectral crest, the color its loudness, and the position on the axis its onset. The lines depict the calculated similarity relationships between the objects. In a future version, a similarity value could be represented by the width of the lines.

Blocks.

This preset consists of a number of rectangles, aligned along a horizontal axis, similar to the *Axes* preset. However, here the objects have two dimension parameters, width and height, replacing the size parameter. Figure 6 shows how the *Rectangles* preset can be used to compare several dymos' temporal proportions and segmentations.[22] The position along the axis depicts a segment's onset, width its duration, height its loudness, and color its timbre (spectral crest).

Graph.

Finally, this preset contains no coordinate axes and organizes the depicted objects based on their relationships using a spring algorithm. Figure 7 shows a dymo's hierarchical structure by visualizing the `part` relations as lines. The same kind of visualization can be created for any graph, e.g. similarity graphs, where we can simply visualize the `similar` relation as edges, as described in the next section. The visual axis parameters have no effect here, but all non-spatial parameters are customizable in the same way as with the other presets.

[22]This screenshot is taken from another app using the *music-visualization* package dedicated to the navigation of the Grateful Dead collection of the Live Music Archive [16].

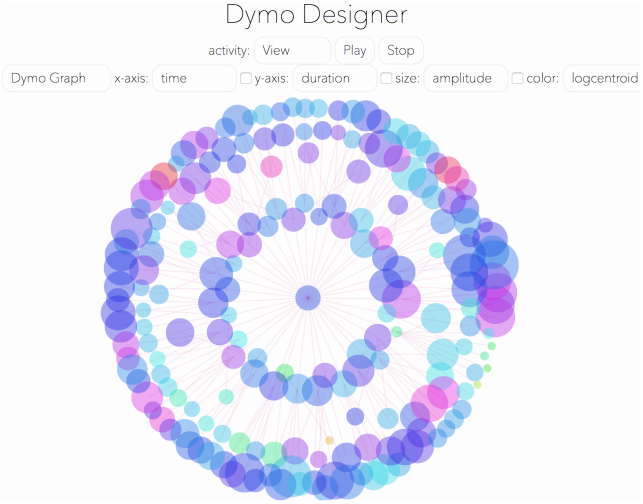

Figure 7: A graph representation of a multilevel dymo representing a bar and beat segmentation, which is still reorganizing its orientation.

3.5.2 *Custom Visualization Mode*

Each of the four application modes described in Section 3.2 has a standard way of visualizing the objects in question: dymos and their relationships in *Dymo Mode*, mappings and parameters in *Mapping Mode*, and again dymos in *Rendering Mode*, usually using the *Graph* visualization scheme. However, users can customize the visualization at any time so that it suits their needs, for instance if they wish to select a particular group of objects to be edited. They can do this by switching to *Visualization Mode*, where they can decide which of the available music visualization schemes they would like to use and which of the depicted objects' properties they which to visualize.

In addition to this, they can also fully customize which objects and relations are shown by setting constraints. Specifically, nodes can be chosen to depict any set of OWL classes and edges any set of OWL object properties. For instance, users could choose to show only objects of type `Amplitude` and show the mappings between them.

3.6 Testing Dymos and their Renderings

Once a dymo and some mappings that include controls are defined, users can start testing the experience they created by playing it back and interacting with it. In *Rendering Mode*, for each control used in the definition of their mappings (see Section 3.4), a mock control appears on the screen, in the form of simple UI controls such as sliders and buttons, depending on the control's nature. Once the dymo starts playing, the mock controls for both UI and sensor controls can be used to simulate the sensor and UI interaction, whereas the mock controls for any auto controls in the mapping hierarchy visualize the respective auto control's state.

By clicking on parts of the dymo hierarchy, users can choose to play back only part of the structure in order to monitor specific aspects of the experience. While the dymo is being played back, any visualization of its dynamic aspects, such as its parameters, change dynamically as well. For instance, if the objects' amplitude parameter is visu-

alized as the size of the corresponding bubble, it changes dynamically as the dymo's amplitude changes. In a similar way, dynamic spatial positions can be visualized using the *Axes* template, where panning could be assigned to the x-axis position, height to the y-axis position, and distance inversely to the size of the visual objects. From *Rendering Mode*, the users can go back to any of the other modes and edit the dymo and rendering until they are satisfied with the result.

4. CONCLUSION AND FUTURE WORK

From the brief summary of the current functionality of the Dymo Designer and the examples shown, we can see how the app facilitates the definition of Dynamic Music Objects in comparison to defining them in RDF using JSON-LD or Turtle. RDF suffices for simple definitions, such as the example in Figure 1, which consists of a few dozen lines of code in JSON-LD. However, for feature-based dymos this quickly gets out of control. The dymo represented in Figure 3, for instance, when serialized to JSON-LD, comprises almost 6000 lines of code. This dymo takes less than a second to be generated in Dymo Designer and it can then not only be visualized so that the user can see if it turned out as desired, but also tested, where any part can be listened to and experimented with upon a simple mouse click. During playback, any current parameter values and thus the dynamicity itself of the dymo can be visualized in unlimited ways.

So far, the Dymo Designer has not yet been evaluated with a larger audience, but it has served as valuable prototyping and example generating tool to the project. Many of the examples created to illustrate the capabilities of the Semantic Music Player would have not been possible to be created without the Dymo Designer and the dymo-generator package (see examples cited in [14, 15]). It has also proven helpful in illustrating the concept of Dynamic Music Objects and the various examples to a non-expert audience.

In the near future, the app will be extended with new functionality, some of which was mentioned in the paper, including designing arbitrary contextual controls by directly referencing entities on the web or fetching results from APIs, defining mappings to parameters of controls rather than just dymos, or extending the visualization package with more presets, such as a three-dimensional coordinate system or additional music visualization standards, and with more customizable visual parameters, such as the width of edges or the shapes of objects.

5. ACKNOWLEDGMENTS

This work is supported by EPSRC Grant EP/L019981/1 *"Fusing Audio and Semantic Technologies for Intelligent Music Production and Consumption (FAST-IMPACt)"*. Sandler acknowledges the support of the Royal Society as a recipient of a Wolfson Research Merit Award.

6. REFERENCES

[1] T. 3364. Audio definition model – metadata definition. Technical report, European Broadcasting Union, 2014.

[2] S. Bechhofer, I. Buchan, D. D. Roure, P. Missier, J. Ainsworth, J. Bhagat, P. Couch, D. Cruickshank, M. Delderfield, I. Dunlop, M. Gamble, D. Michaelides, S. Owen, D. Newman, S. Sufi, and C. Goble. Why linked data is not enough for scientists. *Future Generation Computer Systems*, 29(2):599 – 611, 2013. Special section: Recent advances in e-Science.

[3] K. Collins. An introduction to procedural music in video games. *Contemporary Music Review*, 28(1):5–15, 2009.

[4] D. De Roure. Executable music documents. In *Proceedings of the 1st International Workshop on Digital Libraries for Musicology*, pages 91–3, 2014.

[5] G. Fazekas, Y. Raimond, K. Jacobson, and M. Sandler. An overview of semantic web activities in the omras2 project. *Journal of New Music Research*, 39(4):295–311, 2010.

[6] J. Foote and M. Cooper. Visualizing musical structure and rhythm via self-similarity. In *Proceedings of the 2001 International Computer Music Conference*, pages 419–422, 2001.

[7] D. W. Fourney and D. I. Fels. Creating access to music through visualization. In *Science and Technology for Humanity (TIC-STH), 2009 IEEE Toronto International Conference*, volume 939, pages 26–27. IEEE, 2009.

[8] N. Harley. An ontology for abstract, hierarchical music representation. In *Demo at the 16th International Society for Music Information Retrieval Conference (ISMIR 2015)*, Malaga, Spain, 2015.

[9] M. Harris, A. Smaill, and G. Wiggins. Representing music symbolically. In *Proceedings of the IX Colloquio di Informatica Musicale*, Venice, 1991.

[10] G. Herrero, P. Kudumakis, L. J. Tardón, I. Barbancho, and M. Sandler. An html5 interactive (mpeg-a im af) music player. In *Proceedings of the 10th International Symposium on Computer Music Multidisciplinary Research (CMMR), Marseille, France*, pages 15–18, 2013.

[11] A. Hunt and R. Kirk. Mapping strategies for musical performance. In M. Wanderley and M. Battier, editors, *Trends in Gestural Control of Music*. Ircam - Centre Pompidou, Paris, 2000.

[12] E. J. Isaacson. What you see is what you get: on visualizing music. In *Proceedings of the International Conference on Music Information Retrieval (ISMIR)*, pages 389–395, 2005.

[13] F. Thalmann and G. Mazzola. Visualization and transformation in general musical and music-theoretical spaces. In *Proceedings of the Music Encoding Conference 2013*, Mainz, 2013. MEI.

[14] F. Thalmann, A. Perez Carillo, G. Fazekas, G. A. Wiggins, and M. Sandler. The mobile audio ontology: Experiencing dynamic music objects on mobile devices. In *Tenth IEEE International Conference on Semantic Computing*, Laguna Hills, CA, 2016.

[15] F. Thalmann, A. Perez Carillo, G. Fazekas, G. A. Wiggins, and M. Sandler. The semantic music player: A smart mobile player based on ontological structures and analytical feature metadata. In *Web Audio Conference WAC-2016*, Atlanta, GA, 2016.

[16] T. Wilmering, F. Thalmann, and M. B. Sandler. Grateful live: Mixing multiple recordings of a dead performance into an immersive experience. In *Proceedings of the Audio Engineering Society Convention 141*, Los Angeles, CA, 2016.