The Semantic Music Player: A Smart Mobile Player Based on Ontological Structures and Analytical Feature Metadata
THALMANN, FS; Perez Carillo, A; Fazekas,; Wiggins,; Sandler,; 2nd Annual Web Audio Conference

“The final publication is available at https://smartech.gatech.edu/handle/1853/54596”

For additional information about this publication click this link. http://qmro.qmul.ac.uk/xmlui/handle/123456789/11741

Information about this research object was correct at the time of download; we occasionally make corrections to records, please therefore check the published record when citing. For more information contact scholarlycommunications@qmul.ac.uk
The Semantic Music Player: A Smart Mobile Player Based on Ontological Structures and Analytical Feature Metadata

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

ABSTRACT

The Semantic Music Player is a cross-platform web and mobile app built with Ionic and the Web Audio API that explores new ways of playing back music on mobile devices, particularly indeterministic, context-dependent, and interactive ways. It is based on Dynamic Music Objects, a format that represents musical content and structure in an abstract way and makes it modifiable within definable constraints. For each Dynamic Music Object, the Semantic Music Player generates a custom graphical interface and enables appropriate user interface controls and mobile sensors based on its requirements. When the object is played back, the player takes spontaneous decisions based on the given structural information and the analytical data and reacts to sensor and user interface inputs. In this paper, we introduce the player and its underlying concepts and give some examples of the potentially infinite amount of use cases and musical results.

1. INTRODUCTION

The web plays an increasingly significant role in mobile music. While just a decade ago, the primary means of listening to music on the go were dedicated devices with enough memory to store up to ten thousands of songs, today almost a third of the revenue in digital music stems from consumers that are streaming from the web, many of them on their mobile devices [1]. These mobile devices, compared to early MP3 players, are veritable multi-sensory supercomputers with considerable processing power and a multitude of ways of interaction. Currently, the potential that many consumers are constantly online using such powerful devices is only partially taken advantage of to enhance their musical experience. Most of the platforms on which consumers are listening to music are for instance extended with intricate music recommendation, music recognition, and playlist sharing services. However, the music listening experience itself is, apart from a few exceptions1, unchanged. Songs are still most commonly listened to in a fixed and linear way, as immutable finished artworks.

Until recently, one of the primary reasons for such limitations in the mobile music experience might have been the lack of audio development frameworks that offer a standard way of playing back music on the web and across the various available mobile platforms. Today, the Web Audio API2 bears a great potential for platform-independent audio and music development that goes beyond mere streaming and it can now easily be integrated with novel browser-based app development frameworks.

In this paper, we illustrate this potential by introducing the Semantic Music Player (SMP), a cross-platform web and mobile app designed to play back music in spontaneous, unpredictable, context-dependent, and interactive ways. It is entirely built with web technologies and uses a Semantic Web music format we call Dynamic Music Object (Dymo). This format consists of a semantically annotated structural description of linked audio files and a definition of how it should be played back, which enables a potentially infinite number of use cases and musical results. After summarizing the underlying concepts including dynos and their renderings, we introduce the app, its architecture, and the involved technologies. We then discuss a few sample use cases that illustrate some of the potential of the system. Finally, we go into some more technical detail on the semantic information the Dynos can contain and how it can be extracted from the music.

2. DYNAMIC MUSIC OBJECTS

At the core of the Semantic Music Player is a Semantic Web music format we call Dynamic Music Objects [15]. It is a special case of the more general concept of a Digital Music Object (DMO) which unites various representations of musical content to a bundle of music files for research, composition, production, or consumption [6]. Dynamic Music Objects are oriented towards the intermediary or end consumer and are designed to be musically malleable and flexible with precisely defined degrees of freedom and constraints. The parameters for such musical modifications can be deliberately exposed by producers, composers, or distributors in a description based on the so-called Mobile Audio Ontology [15]. Dynos can be dynamic in several ways, including adaptive, interactive, or autonomously dynamic.3

1There is often confusion around these terms. We see adap-
More specifically, dymos consist of:

- a number of linked music files which can be stored locally or in a distributed environment
- a structural definition of the relationships of the music files to each other, annotated musical features and metadata and enabling a number of modifiable musical parameters
- a playback configuration called rendering that describes how various types of controls and features map to parameters

Both the structural definition and the rendering can be represented using for instance RDF (Turtle)\(^4\) or JSON-LD\(^5\) while referring to the Mobile Audio Ontology, an OWL Ontology\(^6\). At the basis of the structural definition is a new implementation of CHARM, an abstract music representation system allowing multiple hierarchies of musical objects or events related by arbitrary logical formulae and abstracted from concrete applications [10, 9]. Each object can have an arbitrary number of sub-objects called parts and a type that determines the relationship of the parts to each other and their parent.\(^7\) Furthermore, an object can have a number of musical attributes called features and parameters. In our definition, features are immutable analytical values extracted from the music and gathered from various other representations, such as for instance the Vamp Ontology or the Audio Feature Ontology \(^7\). Parameters, on the other hand, are modifiable values that describe aspects of the music that can be changed. Finally, in addition to these hierarchical relationships we can describe any kind of graph-based relation between objects in any part of these structures, for instance similarity relations. Using SPARQL\(^8\) or graph query algorithms, the structure can then be queried and navigated on demand by the player.

While several comparable formats exist, none of them take advantage of both the Semantic Web and platform-independent web and mobile development. RJD\(^9\) and iXMF\(^10\) 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 is discontinued. Both IM AF \(^12\) and the Audio Definition Model \(^2\) 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 has the significant advantages that one can directly build on other interactive and interactive music as special cases of dynamic music as in \(^5\). However, while adaptive music indirectly adapts to the listener’s context and interactive music allows noticeable direct user interaction, dynamic music also includes music that reorganizes itself, without any external influence.

\(^4\)http://www.w3.org/TR/turtle/
\(^5\)http://json-ld.org
\(^6\)http://www.w3.org/TR/owl2-overview/
\(^7\)The most basic such types are conjunction, disjunction, and sequence, but they can get arbitrarily complex and describe more specific musical entities.
\(^8\)http://www.w3.org/TR/sparql11-query/
\(^9\)http://rjdj.me
\(^10\)http://www.iasig.org/wg/ixwg/

\(^11\)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 hasPart or hasParameter.

\(^12\)http://www.iasig.org/wg/ixwg/

Figure 1: A sample Dynamic Music Object.
all children. By default, every parameter of a parameter in relation to a parent object maps to the parameter of the same name in lower-level parameters. In this way we can define higher-level parameters that change multiple lower-level ones.

Defining Dymos directly as OWL or JSON-LD files can become rather tedious with an increasing number of features, especially for segmentations. We are thus also developing a browser-based application called Dymo Designer which visualizes Dymos in highly flexible ways and provides a simple and intuitive interface for automatically adding features to the structure and defining mappings, which we will encounter in the next section. Figure 2 shows the current version of the Dymo Designer representing an object with several levels of segmentation.

2.2 Mappings, Renderings, and Higher-Level Parameters

So far we have dealt with the structural definition of Dynamic Music Objects. What makes them dynamic is the fact that they have modifiable parameters, a few of which we encountered in the example in the previous section. It is the purpose of so-called renderings to define to what values these parameters are set to initially and how they are controlled or changed during playback. A rendering consists of a set of mappings from any of the available controls and features to any of the defined Dymo parameters. A Dymo can have mappings as well, from any parameters and features to lower-level parameters. In this way we can define higher-level parameters that change multiple lower-level ones. All mappings are currently based on arbitrary JavaScript functions which can have features, controls, and parameters as arguments and an arbitrary function body.

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 3 shows how we can define mappings to achieve automatic mixing between two parallel 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.\footnote{This is also how we can achieve the relative dependencies described in Section 2.1. By default, every parameter of a parent object maps to the parameter of the same name in all children.}

$$\text{Fade} \times \text{index} \rightarrow \text{Amplitude}$$

$$(a, b) \mapsto (1 - b) \times (1 - a) + b \times a$$

For illustration, in JSON-LD we write the following:\footnote{The index feature denotes the index of a part in a conjunction or a sequence.}

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

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:

$$\text{Tempo} \times \text{duration} \rightarrow \text{TimeStretchRatio}$$

$$(a, b) \mapsto a/60 + b$$

Finally, we 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 duration.

\footnote{The dyamos function selects the subset of dymos to be mapped to (here all parts of the main dymo, i.e. on hierarchical level 1)
Figure 4: The GUI dynamically generated for the rendering in Figure 3, shown on an Android device.

![Image](image.png)

$$\text{Fade} \times \text{tempo} \rightarrow \text{Tempo}$$

$$(a, b) \mapsto b[0] + a \ast (b[1] - b[0])$$

Then, we can map any control, such as in our example a slider and a ramp triggered by a button (Figure 3), to the higher-level parameters just defined. Of course, defining such mappings can be tricky for an average user and defining more complex multi-dimensional mappings can quickly become tedious. One of the goals of the Dymo Designer is to facilitate this by offering an intuitive graphical interface to define mappings (see Figure 6).

3. THE SMP USER INTERFACE

The current user interface of the Semantic Music Player is kept as simple as possible: a dropdown menu lets users choose between all Dymos the player has access to and three standard playback buttons can be used to trigger playback. There are also currently two additional buttons that can be used for monitoring and handling sensors. Everything else is generated dynamically once a user selects one of the objects. Depending on the object’s definition the player adds any necessary UI elements such as sliders, toggles, or buttons, labels them as defined, and allocates any required sensor controls. Figure 4 shows the interface generated for the rendering created in Section 2.2, consisting of just the Fade slider and the Transition button.

The Semantic Player manages mutual dependencies between various controls. In our example, pressing the transition button triggers the ramp which then gradually moves the Fade parameter. Since the Fade parameter is also mapped to from the slider, the slider starts moving as well. This currently works for any arbitrary invertible algebraic mapping function with one parameter.

4. ARCHITECTURE AND TECHNOLOGY

The Web Audio API is a straightforward choice for developing web-based audio platforms and frameworks considering its increasing versatility and popularity. However, one of the problems that come with it is that not all browsers support the same subset of its functionality which makes results somewhat unpredictable. This is especially the case for browsers on mobile platforms. For the Semantic Music Player we thus decided to use the Web Audio API in conjunction with Ionic\textsuperscript{15} and ngCordova\textsuperscript{16}, two frameworks that simplify the development of browser-based cross-platform mobile apps. This allows us not only to create standalone native apps for any device, but also to include audio functionality in a more controlled way. In particular, Ionic has the advantage over other frameworks that it allows distributing a specific browser as part of the app, in our case Crosswalk\textsuperscript{17}, which is then used regardless of the user’s default browser. We can thus ensure that the audio functionality of the app will be the same on all mobile platforms.\textsuperscript{18}

Figure 5 depicts the architecture of the framework along with the used technologies. The core of the SMP is formed by an object-oriented model written in JavaScript which defines all the classes necessary to load Dymos and represent them internally so that they can be played back and manipulated dynamically.\textsuperscript{19} The controls, which are responsible for the manipulation of Dymos, are depicted in the top left corner of the figure. They include sensor controls, such as the accelerometer, geolocation controls, or the compass, provided via ngCordova, as well as user interface controls, such as sliders, buttons, and toggles, provided by the Ionic/AngularJS frontend. In addition to these, the core of the application also offers autonomous control units, including statistical and random controls, graph navigation controls and, in the near future, units based on artificial intelligence. A last type of controls, to be added at a later stage, are so-called contextual controls, which are based on contextual data queried from the web, including user-specific or public metadata, such as musical preferences and time or weather information.

The player classes, built with the Web Audio API, include a scheduler which decomposes a given Dymo into its parts and creates the appropriate number of sources, each

\textsuperscript{15}http://ionicframework.com
\textsuperscript{16}http://ngcordova.com
\textsuperscript{17}https://crosswalk-project.org
\textsuperscript{18}Using Crosswalk in Ionic and Web Audio API projects was suggested by Devan Sabaratnam at https://www.airpair.com/ionic-framework/posts/using-web-audio-api-for-precision-audio-in-ionic.
\textsuperscript{19}The model can be obtained as a Bower package and is currently used by other applications developed in the context of this project (https://github.com/florianthalmann/dymo-core.)
of them offering different realtime audio controls and effects, spatial positioning, etc. These sources are built on demand and ensure smooth playback being optimized to run even on slower mobile devices, using pre-buffering of audio segments and crossfading to avoid clipping. The way a Dymo is played is determined by a given navigator which traverses the Dymo in a specific way, e.g. based on similarity relations between the Dymo’s parts. The bottom left corner of Figure 5 shows the Dymos themselves, which can either be encapsulated in their own local folder along with their music files, or be distributed in various locations on the web. For the distributed case, we also developed a special audio server application that streams specific chunks of files on demand rather than entire files, which is crucial for some of the audio manipulations the SMP performs, as we will see in the next section.

5. SOME EXAMPLES OF USE CASES

We have already encountered a few simple examples of what can be done with Dynamic Music Objects and the Semantic Music Player. In this section we briefly examine a few further possibilities.

5.1 Spatialization, Spatial Navigation, and Deformation

When working with Dymos containing multiple audio sources, we can not only pan them in order to emulate audio mixing, but we can distribute them in the binaural three-dimensional space offered by the Web Audio API. In this way we can for instance distribute the instruments of a band, each of them represented by a Dymo, around the listener by setting appropriate initial position values. With mappings from compass and geolocation controls to the listener position, we can enable listeners to change their orientation and walk around the space while the music is playing. This can be intensified using additional parameters such as reverb or filtering. In [13] we generalize this and instead of emulating a performance space we create an immersive music-theoretical space by distributing the individual notes of a decomposed audio recording on the well-known chroma helix. In a similar way, we can highlight any analytical aspects of the music, e.g. exaggerate expressive variety by mapping an amplitude feature to the Amplitude parameter.

Another way of using space with our framework is to navigate playlists or, more generally, temporal musical structure using geolocation mappings. Using a similar mixing mechanism as the one discussed in Section 2.2, but generalized for two-dimensions, we can create landscapes of interlocking musical tracks that can be synchronized using mappings to the Play parameter, but faded in and out at different times. This results in a varying multilayered structure similar to Bluebrain’s The National Mall20 or the ones that can be created with the system described in [11]. Figure 6 shows how the Dymo Designer can be used to draw arbitrary polygonal shapes which can then be transformed into polygon containment functions or interpolation functions directly to be used in mappings [15].

5.2 Variation of Temporal Structure based on Multilevel Similarity

We have briefly seen how a Dymo can represent multilevel segmentation (Section 2.1). When given such an object, the player can vary its temporal structure by changing the order of segments, on various levels at the same time. For instance, we can define a rendering that plays the sections of a piece in backwards order, skips every other bar and switches the first and second beats of each bar. The player can also take spontaneous decisions using one of the statistical controls mentioned in Section 2.2, which can be entertaining. However, for such decisions to make greater musical sense, we can also incorporate additional musical information based on similarity measures. By establishing similarity relations between similar sections, bars or other objects (Figure 2) the player can learn where it can leap to from where. The graph navigator control can take such decisions within given constraints and extend or contract the music in the fashion of the Infinite Music Jukebox.21 In the next section, we will see how we can measure such similarity.

6. EXTRACTING AND PREPARING FEATURES

As we have seen earlier on, Dynamic Music Objects typically include semantic information extracted from the music files they are based on. One way such features can be extracted is using the Sonic Annotator22, a command line tool that outputs feature data in various formats. Sonic Annotator analyses audio files based on any number of given Vamp Plugins, each of them specialized on a small number of related features [4]. Especially useful in connection with this work is that Sonic Annotator can output data as RDF files, thereby linking to appropriate musical ontologies telling us about the data. Other serialization formats include simple CSV or JSON based formats, with an RDF compatible JSON-LD format under development. Such feature files can directly be loaded into the Dymo Designer and are automatically added to the Dymo structure in a configurable way. For instance, if the feature is represented by a high-resolution signal for the duration of the analyzed audio file, each object in the hierarchy is annotated with either the average, median, or initial value of its temporal interval. Some

20Link in Footnote 1.
21http://www.infinitejuke.com
22www.vamp-plugins.org/sonic-annotator/
of the annotations the SMP supports are more complex and need to be prepared beforehand. In the following, we briefly show how this works with the example of similarity.

6.1 Relational Features: Beat Similarity as an Example

The structural definition of Dymos allows relations to be established between objects at any level, for example similarity relations. We can do this by calculating them and constructing a graph representation externally and then importing it into the Dymo Designer. Measuring similarities in recorded music is a difficult task that depends on a great variety of criteria. It is usually approached by numerically comparing psychoacoustic descriptions of the music. For instance, if we wish to measure the similarity of beats within a song, we can extract relevant features related to psychoacoustic characteristics of the sound (i.e., timbre, melody, loudness and rhythm) from the audio signal and average them across each beat interval (beat-synchronous features). From these beat-synchronous feature estimations we can then calculate a self-similarity matrix and represent it as a graph.

However, the Dymo Designer also offers some internal functionality that allows to automatically infer this from the information available in the given Dymo. We can select any set of sub-objects that are annotated with a number of boiled down features as described in the previous section. Then, the app creates feature vectors containing all features common to all selected objects, normalizes them, and calculates their pairwise similarity e.g. using a cosine distance, as suggested in [8], and finally adds similarity relations to the pairs where the resulting value is above a given threshold.

7. CONCLUSION

We have introduced the Semantic Music Player, a prototype of a cross-platform web and mobile application that plays back music in spontaneous, interactive, and context-dependent ways. We have also shown what possibilities are enabled by the underlying abstract music representation system and given a few examples of potential applications that illustrate the versatility of the framework. Finally, we have briefly shown how such representations can be built and how applications with an intuitive user interface may help during this process.

Due to its flexible architecture, the functionality of the Semantic Music Player can be easily transferred to other applications and be deployed in various contexts. For instance, we are currently working with the authors of the Moodplay system [3] to create a web-based version performing automatic mixing within a large music collection of 10000 songs, with mixing principles similar to the ones described in this paper. On the other hand, we are collaborating with artists to create specifically composed native mobile apps to test various dynamic music use cases.

There are many possible extensions to be realized in the near future, some of them mentioned in the paper. For example, the incorporation of contextual controls based on data taken from the web seems a sensible next step, especially since the framework is already built on Semantic Web technologies. Another possible extension could be the incorporation of comparative analysis data from larger music collections, which could lead to more informed and even learned playback decisions.

8. REFERENCES