MUSICJSON: A REPRESENTATION FOR THE COMPUTER MUSIC CLOUD

Jesus L. Alvaro
Computer Music Lab
fauno.org, Madrid, Spain
JesusLAlvaro@gmail.com

Beatriz Barros
Dpto. Lenguajes y Ciencias de la Computación
UMA, Málaga, Spain
bbarros@lcc.uma.es

ABSTRACT

New cloud computing ways open a new paradigm for music composition. Our music composing system is now distributed on the Web shaping what we call as *Computer Music Cloud* (CMC). This approach benefits from the technological advantages involved in distributed computing and the possibility of implementing specialized and independent music services which may in turn be part of multiple CMCs. The music representation used in a CMC plays a key role in successful integration. This paper analyses the requirements for efficient music representation for CMC composition: high music representativity, database storage, and textual form. Finally, it focuses on its textual shape, presenting *MusicJSON*, a format for music information interchange among the different services composing a CMC. MusicJSON and database-shaped representation, both based on an experienced sound and complete music representation, offer an innovative proposal for music cloud representation.

1 . INTRODUCTION

Cloud Computing, a new term defined in varied ways [7], involves a new paradigm in which computer infrastructure and software are provided as a service [5]. This services themselves are referred to as *Software as a Service (SaaS)*. Google Apps is a clear example of *SaaS* [8]. Computation infrastructure is also offered as a service (*IaaS*), thus enabling the user to run the customer software.

This new paradigm offers new possibilities for the design of composition systems. Fig. 1 shows the Computer Music Cloud (CMC) approach where the system is distributed across specialized online services [2]. The user interface is now a web application running in a standard browser (1). A storage service is used as an edition memory (2). An intelligent-dedicated service is allocated for music calculation and development (3). Output formats such as MIDI, graphic score and sound file are rendered by independent services exclusively devoted to this task (4). The web application includes user sessions to allow multiple users to use the system. Both public and user libraries (5) are also provided for music objects. Intermediary music elements can be stored in the library and also serialized into a *MusicJSON* format file, as described below.

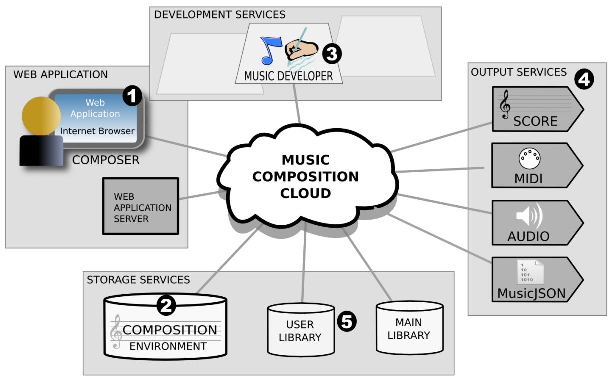


Fig. 1. Basic Structure of a Composition Music Cloud

This CMC approach has several advantages. Some of them come from Cloud Computing, such as scalability, optimization and reuse of available resources. Others come from web applications, such as decentralized information, being able to work from any computer with a standard Internet connection and browser, and the inherited code. Also, the division of the music system into services allows for the design and implementation of independent services with the most appropriate tools, It is apart from the availability of a service for different CMC systems. Since services can be shared, the design of new music systems is facilitated by the joint work of different services controlled by a web application.

The key factor in successful integration is the use of a well-defined music representation for music data interchange. This is also the objective of this paper: presenting a proposal for the interchange of music information among the services which shape a music composition cloud. In the next section, the types of services included in this cloud are analysed as a base to define the requirements to be fulfilled by the selected music representation, tackled in Section 3. Section 4 describes the MusicJSON format as a textual form of the used representation, while Section 5 describes some use examples. The paper ends with some conclusions and some points referred to related work.

2 . MUSIC SERVICES IN THE CMC

In a simple form, Music Web Services are servers receiving a request and performing a task. At the end of a task the resulting objects are returned to the stream or stored in an interchange database. The access to this database is a valuable feature for services since it is a shared workspace where the componets of music composition are represented. The Music Services of the cloud can be classified according to their function. These services are described in the following subsections.

2.1 Input

This group includes the services aimed particularly at incorporating new music elements and translating them from other input formats.

2.2 Agents

They are those services which are capable of inspecting and modifying music composition, as well as introducing new elements. They include human user interfaces, but they may also consider other intelligent elements taking part in music composition[4]: introducing decisions, suggestions or modifications. In our prototype, we have developed a web application [2] acting as a user interface through the edition of music objects.

2.3 Storage

There are two main types of storage services: libraries and composition environments. Libraries store music objects which shall be used in different compositions. There are general libraries and user libraries. *Main lib* stores shared music elements as global definitions. This content comprises music elements shared by all users as a shared music language. User-related music objects are stored in the *User lib* and include composer-defined music objects which can be reused in several parts or compositions.

Composition environments are the storage services where the piece is progressively composed. This database contains the composition environment (i.e., everything related to the piece currently under composition), and does not only act as a space for information interchange, but also as a real and shared music environment with which several services can interact simultaneously and coordinatedly.

2.4 Development

The services in this group perform *development* processes. As explained in [1], *development* is the process by which higher-abstraction symbolic elements are turned into lower-abstraction ones. High-abstraction symbols are implemented as *meta-events* and represent music objects such as *motives*, *segments,* and other composing abstractions [1]. Algorithmic composition developers and other intelligent services, such as constraint solvers or genetic algorithms, are examples of this type of music development service .

2.5 Output

These services produce output formats as a response to requests from other services. They render formats from the element currently under edition for immediate composer feedback as well as the whole score or audio. The MIDI file for audio playing and standard notation in a graphic format are two clear examples of this. Other output formats are also possible by integrating a suitable translation service.

3 . MUSIC REPRESENTATION FOR THE CMC

To achieve an effective integration of all elements in the cloud for music composition, all services must share the same music representation. This representation must meet three main requirements: satisfactorily represent the basic elements of music composition in a solid hierarchy of classes; present a representation form for storing music objects in a database; and count on a textual form which facilitates the interchange of music objects among the different services. Besides, the cloud's distributed nature must also be taken into account by incorporating the possibility of distribution in the data. All these elements are described in the following subsections.

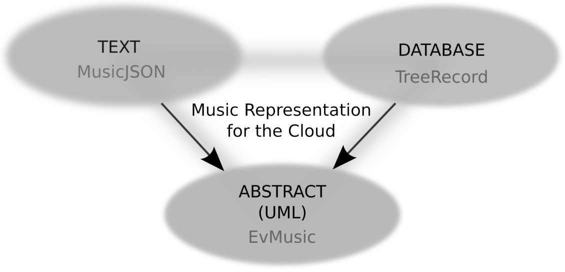


Fig. 2. Abstract, Database and Textual Forms of Music Representation

3.1 EVMusic Representation

The proposal is based on EVMusic representation [1]. It is a robust model with high multi-level representativity, multiple topologies and meticulous, detailed representation of music pitch, as well as full compatibility with traditional music notation. Likewise, its multi-level nature and expandability allow for representing music elements at varied abstraction levels.

EVMusic representation counts on a complete hierarchical organization of classes designed in the platform-independent UML [13], which allows for its use in multiple programming languages. Figure 3 shows a brief extract from the UML representation, showing only some of the classes present in the music fragment in Figure 6, which shall work as a reference for subsequent examples. The present paper is not aimed at contributing a detailed analysis of EVMusic classes. Nevertheless, we shall contribute a brief description of some of the most relevant aspects shown in this figure, particularly the relations.

In a UML Class Diagram [12], the inheritance relation among classes is represented with a hollow pointed arrow. Thus, it can be observed in this figure that a *scoreelement* is an *event*, which in turn is a *treeobj* just like *singlepitch*. Inheritance relations can also be multiple, as it occurs with the note class. Thus, it can be deduced that a note is either a score element with a unique pitch ,or a pitch temporarily placed on the score.

Containment relations are indicated by means of rhombus-shaped arrows and the name of the relation. Thus, it can be observed in the figure that an object of the *aggregate* class includes an attribute known as *pitches*, which is a container of *singlepitch*-like elements. Therefore, a chord (an *nchord*-like object) which inherits the *aggregate* properties is a *scoreelement* which contains several *singlepitches*. Grace notes are also represented with a content relation: they are a sequence of *singlepitches* which ornament a *scoreelement* and are stored in its slot known as *graces*. Finally, the structure of similar objects is also indicated by the content relation. In EVMusic, each event is by default a group of subevents, as clearly reflected in the fractal-like structure of music time. Thus, these subevents are stored in the slot *events* of the main event. The tree structure is represented by means of the *treeobj* class and its *parent* relation. It is extended to all representation elements and shall be commented on in the following section.

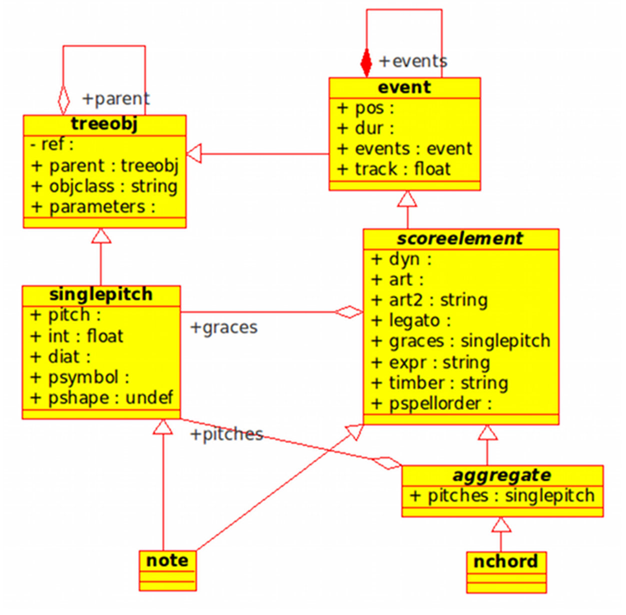


Fig. 3. UML Diagram with some classes of EvMusic Representation

3.2 Database Stored Representation

Database storage allows several services to share the same data and collaborate in the composition process. The information stored in a database is organized in tables of records. The database representation must achieve two objectives: store music instances and store the relations among these forms.

Figure 4 shows a simplified scheme with some database fields. To represent the objects of a particular class, the table of the database incorporates the field *objclass*, which contains the name of the object class. The attributes of each class are usually represented by fields with the name of the attribute. In this figure, these attributes are indicated in the rows of predefined attributes (*AttributeA, AttributeB,...*) Importantly, appropriate representations must enable the incorporation of new elements, even those which have not been defined yet. Some pairs of generic fields have been provided with this purpose. These pairs of generic fields store both the name of the attribute and its value. They are indicated in the central rows of this figure as Expandable Attributes. For instance, if we want to incorporate a new attribute known as "zattrib", the register will contain "zattrib" in the field *attrib1name,* and its value in the field *attrib1value*. Thus, implementing a new storage system is not necessary when a new class, with its new attributes, is defined.

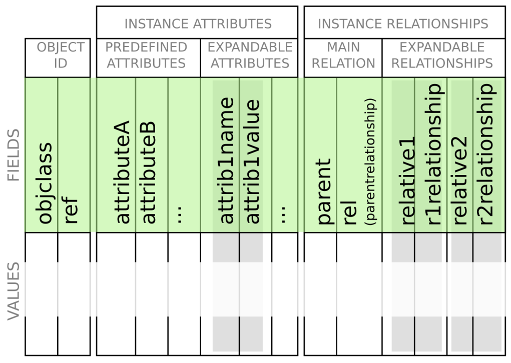
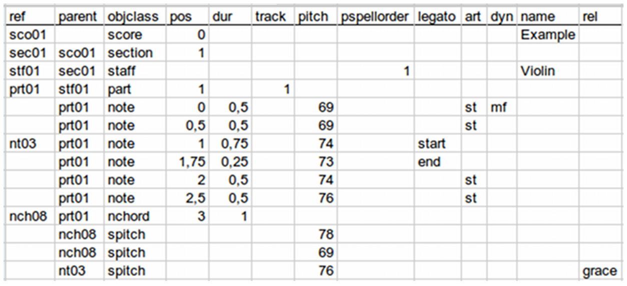


Fig. 4. Record Structure of Data Base

The relations among the music objects are also represented in the database. Among all of them, the *containment* or belonging relation is likely to be the most important one. As we have already mentioned, the music objects of the abstract EVMusic representation are usually tree-shaped related. To be stored in a database, these tree structures must be previously converted into records. For this purpose, the three main classes of EV representation are subclassed from a tree node class *treeobj*, shown in Fig. 3. Thus, every object is identified by a unique *reference* and a *parent* attribute. This allows to represent a large tree structure of nested events as a set of records for individual retrieval or update. By default, the slot *parent* always refers to the *containment* relation. However, other relation can be used for this main tree. The field *parentrelationship* (abbreviated as *rel*) was incorporated with this aim. For instance, the last row in Table 1, shows how the grace note indicates the main note *nt03* as *parent*, but the parent relationship in this case is not a temporary structure, but a grace-note structure, so its value in the field *rel* is "*grace*".

The representation of relations has been completed by incorporating new relatives. As it can be observed in Figure 4, the main relation *parent* was added pairs of fields aimed at indicating new relatives. Thus, for instance, a new relation r1 can be incorporated by indicating the reference of the referred object in the field *relative1* and the relation between them in the field *r1relationship*. The incorporation of new relations in the database contributes an important degree of representativity since it opens new creative and representative possibilities, such as, for instance, the opportunity of representing *constraints* among music objects or the definition of some objects according to others. Multiple relations also allow for the coexistence of several organizations of objects, letting , for instance, the same note belong to both a temporary structure (represented by the relation *parent*) and a harmonic structure (represented by an *extended relation*) simultaneously.



ref	parent	objclass	pos	dur	track	pitch	pspellorder	legato	art	dyn	name	rel
sco01		score	0								Example	
sec01	sco01	section	1									
stf01	sec01	staff					1				Violin	
prt01	stf01	part	1		1							
	prt01	note	0	0,5		69			st	mf		
	prt01	note	0,5	0,5		69			st			
nt03	prt01	note	1	0,75		74		start				
	prt01	note	1,75	0,25		73		end				
	prt01	note	2	0,5		74			st			
	prt01	note	2,5	0,5		76			st			
nch08	prt01	nchord	3	1								
	nch08	spitch				78						
	nch08	spitch				69						
	nt03	spitch				76						grace

Table 1. Database Content for a simple example

Table 1 shows the database content for the music example notated in Figure 6. The objclass field indicates the class of every instance. Note the relation with the music example and the following listing code in Section 4.1

The music objects described in the database of a music storage service can belong either to the environment of the music piece currently under composition, or to a general or user library. Library objects are referred to by other objects by putting the prefixes *x.lib* and the reference of the library before the reference of the object, as we shall see in the following examples.

3.3 MusicJSON Textual Representation

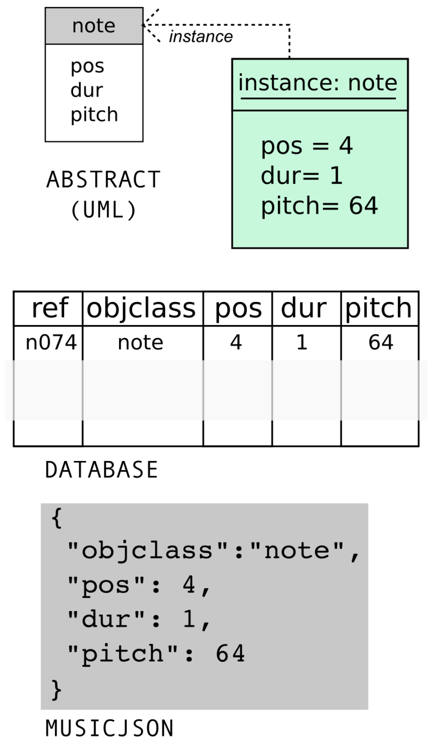
The third specification which must be met by the CMC representation is counting on a textual form which allows music information to be interchanged among services through web streams.

When it comes to design an appropriate textual format for data, basing on formats already widely-used in the Internet seems a rather convenient strategy. Web applications usually use XML and JSON (Java Script Object Notation) [11] for data interchange. Both formats meet the requirements. XML has been successfully used for score representation [15]. However, we opted for JSON, mainly due to the large JSON-compatible tool library available at the time of writing this paper, and the fact that JSON is the interchange format for some of the main Internet web services such as Google or Yahoo. In addition, JSON provides great features such as human readability and dynamic unclosed object support, a very valuable feature inherited from the prototype-based nature of JavaScript [10]. To facilitate communication, JSON also offers JSONP [9] and its corresponding libraries for web applications, which extends interaction flexibility among web services.

As mentioned in [11], *"JSON is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.[...] JSON is built on two structures:*

- *A collection of name/value pairs. In various languages, this is realized as an object, record, structure, dictionary, hash table, keyed list, or associative array*
- *An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence."*

These universal data structures in JSON can be used to describe EvMusic objects and communicate among web music services. *MusicJSON* is the name given to this use. Once the database-shaped representation has been detailed, MusicJSON is easily understandable since they are directly and closely related. MusicJSON can be understood as a serialization of database content. MusicJSON objects are therefore collections of key/value pairs which have been assigned the afore-described attribute *objclass*, so each object always declares its class.



ref	objclass	pos	dur	pitch
n074	note	4	1	64

```
{
 "objclass":"note",
 "pos": 4,
 "dur": 1,
 "pitch": 64
}
```

Fig. 5. A note instance in UML, database and MusicJSON representation

To give a simple example, Figure 5 shows the same object as an instance in a ULM diagram, as an entry in the database, and as a MusicJSON text. Please observe that the textual representation includes the attribute *objclass* with the value "note".

The code in Section 4.1 shows a more illustrative example applied to a brief music fragment. Compared to its standard notation in Figure 6, and Table 1, MusicJSON code is easily understandable. In order to improve the readability, some data semantics are allowed in MusicJSON, like the use of pitch names instead of numbers, as shown in the code.

In MusicJSON not all the attributes of the object are necessary, but only the relevant ones (i.e., those necessary for object definition and abstract-instance construction). If the values corresponding to a necessary property are not indicated, the default values of its corresponding class, or even specific default values defined for a particular group of objects shall be taken. The key *defaultcontent* is provided with this purpose.

Regarding structures, represented in the database as content relations among entries, MusicJSON is not a mere serialization of each database register. It can represent arrays or lists, so structures are presented directly in a deployed form. This tree structure can be observed in the listing code in Section 4.1, which spreads completely the tree *score -> section -> staff -> part -> note*.

Type	Description	MusicJSON use	Previous definition	Extended Reference
External	Use of an external object by its MusicJSON url	{ "objclass":"xref", "type":"url", "url": "http://myobjects.com /mymotive", "subclass":"motive" }		x.url.http://myobjects.com /mymotive
Library	Use of an Object from a Library	{ "objclass":"xref", "type":"lib", "lib": "main", "name": "mota", "subclass":"motive" }	"lib": { "main":"http://mylibs.com /mainlib", ... }	x.lib.main.mota
Variable	Object defined into a variable	{ "objclass":"xref", "type":"def", "name": "ma", "subclass":"motive" }	"def":{ "ma": { "objclass":"motive", "symbol": [0,7,5,4], "slength": "+-+- +--+", }, ... }	x.def.ma
Context	Use of a context object	{ "objclass":"xref", "type":"ctx.mtime", "name": "scale", "subclass":"rpcs" }		x.ctx.mtime.scale

Table 2. Extended references in MusicJSON

Other valuable feature of MusicJSON is related to its distributed nature: *extended references*. Like a hyper-document, MusicJSON allows for the use of external objects either from a library or directly. A new kind of *xref* object was defined with this purpose. The following table shows some types of *extended references* such as external objects by means of URL, elements of an imported library, elements defined in a variable, or contextual elements.

Referred objects can be used in two ways: either defining an xref-type object (as shown by the third column), or using directly the extended reference with the notation of prefixes separated by dots (as shown in the last column). The former section has already shown how library objects are referred to with the prefix *x-lib*. The extended reference for an external object is denoted by the prefix *x.url* followed by the URL of the object. The individual and direct references to a previously-defined object are indicated by the prefix *x.def* followed by the reference previously defined within the same context.

The incorporation of xref objects and *extended references* is an important value added to textual representation since it allows for combining varied elements which can be distributed in the Web. Arguably, this makes it a valuable feature for a knowlegde representation for the Cloud. In addition the use of *extended references*, is open to the definition of new references by using new prefixes. For instance, the last row in Table 2 shows a reference to the context of musical time, denoted by the prefix *x.ctx.mtime*. The time context of a music object is very important, as in harmony. For instance, an extended reference of this type enables the definition of an object which depends on the current harmonic context.

4 . MUSICJSON IN EXAMPLES

This section shows some illustrative examples of the use of MusicJSON.

4.1 Music Fragment in Traditional Notation

As previously stated, MusicJSON is compatible with the traditional notation. For the sake of illustration, we include a simple example of traditional notation of a brief music fragment and its corresponding representation in the MusicJSON format. All music objects in the example are represented in the UML diagram of Figure 3. Note MusicJSON's high readability and clear relation to traditional notation. Also note the generic tree structure of a score: *score -> section -> staff -> part -> note*, at the beginning of the code, and the relation to the database content in Table 1 representing the same music example.

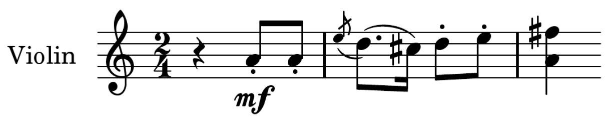


Fig. 6. Score notation of the example

```
{
  "objclass": "score",
  "pos": 0,
  "events": [{
    "objclass": "section",
    "pos": 1,
    "events": [{
      "objclass": "staff",
      "name": "Violin",
      "pspellorder": 1,
      "events": [{
        "objclass": "part",
        "track": 1,
        "pos": 1,
        "events": [{
          "objclass": "note",
          "pos": 0,
          "dur": "0.5",
          "pitch": 69,
          "art": "st"
          "dyn": "mf",
        },
        {
          "objclass": "note",
          "pos": 0.5,
          "dur": "0.5",
          "pitch": 69,
          "art": "st"
        },
        {
          "objclass": "note",
          "pos": 1,
          "dur": "0.75",
          "pitch": "d5",
          "legato": "start",
          "graces": [{
            "objclass": "spitch",
            "pitch": 76
          }]
        },
        {
          "objclass": "note",
          "pos": 1.75,
          "dur": "0.25",
          "pitch": 73,
          "legato": "end"
        },
        {
          "objclass": "note",
          "pos": 2,
          "dur": "0.5",
          "pitch": 74,
          "art": "st"
        },
        {
          "objclass": "note",
          "pos": 2.5,
          "dur": "0.5",
          "pitch": 76,
          "art": "st"
        },
        {
          "objclass": "nchord",
          "pos": 3,
          "dur": "1",
          "pitches": [{
            "objclass": "spitch",
            "pitch": "f#5"
          },
          {
            "objclass": "spitch",
            "pitch": 69
          }
          ]
        }
        ]
      }]
    }]
  }]
}
```

4.2 Storage Service Access

The interchange of musical information between the services of the CMC can be done directly, but it is also possible to exchange it in a shared form through the database. Any exchange of information with the Storage Service is done in MuiscJSON format; not just musical objects, but also the communication protocol. The storage service responds to standard GET requests with a URL that ends with the function to perform, usually a CRUD function (Create, Retrieve, Update, Delete). The request is accompanied by two parameters: a *data* parameter with information in MusicJSON format and a second parameter named *callback*, as the *JSONP* function to be returned. To give an example, Table 3 shows a complete update request to change the duration of note with ref "*note_84*" to a new value of 12.

url	`http://evmusic.fauno.org/storage03/update`
callback	`stcCallback1002`
data	`{"duration":12,"ref":"note_84"}`

Table 3. MusicJSON request parameters

After updating the corresponding data in the storage service, the returned response has the following form:

```
Callback({"message": "Message Content",
"data":[responsed data], "success": true})
```

This means that the returned data are sent back along with a status message confirming the transaction, everything encapsulated in a function with the name of the *callback*, as required by the JSONP data transaction, so the service can accept AJAX requests from other domains. In our example, here is is the response received to the update request above:

```
stcCallback1002({"message": "Updated
record", "data": {"track": "1", "objclass":
"note", "pitch": "39", "start": 6,
"duration": 12, "ref": "note_84", "id":
84}, "success": true}
```

4.3 Library Use

Library services can make use of predefined musical objects. To use a library service, this must be declared in *lib* section of the document in a key-value pair. The reference name for that library is assigned as the key, while the URL that returns the library itself, usually encapsulated as JSONP, is assigned as the value.

```
"lib": {
  "main": "http://evmusic.fauno.org/lib/
main/instruments",
  ... ,
}
```

The content of the library returned from the given address is:

```
LibraryCallBack({
"message":"Sent data: 45 entries",
"data":{
"flute":{
    "objclass":"pinstrument",
    "families":[
        "wind", "air",
        "wood", "woodwind"],
    "clef":"treble",
    "transpose":0,
    "stafforder":1.2,
    "constraints":{
        "minpitch":"c4",
        "maxpitch":"c7",
        "usualpitch":"g5",
        "polyphony":1,
        "maxtime":12,
        "maxspeed":90,
        "maxlegatointerval":17
        },
    "techniques":[
        "frullato", "whistle",
        "air", "keysound"]
    },
"oboe":{
    "objclass":"pinstrument",
    ...
    }
...
},
"success":"true"})
```

In this case, the complete library is returned, but it is also possible to ask for only one object from the library, by completing the URL with the *key* that corresponds to that object. Thus, instead of downloading the whole library, we save memory by downloading only the data we need. For instance, in order to access only the flute instrument, we would use the following URL: "`http://evmusic.fauno.org/lib/main/instruments/flute`".

In order to use an object from the library, the *x.lib* prefix followed by the library key must be indicated as reference, as shown in Table 2. The MusicJSON code of the referred object will replace the library call during parsing:

```
{
 "objclass": "scoinstrument",
 "name": "Flauta",
 "value": "x.lib.main.flute"
 }
```

4.4 MusicJSON File

Every EvMusic object, from single notes to complex structures, can be serialized into a MusicJSON text and subsequently transmitted through the Web. In addition, MusicJSON can be used as a music format for local storage of compositions. Next listing code shows a draft example of the proposed description of an EvMusic file.

```
{"objclass":"evmusicfile","ver":"1002",
"content":
    {"lib":{

"instruments":"http://evmusic.fauno.org/lib/
main/instruments",
        "pcstypes":
"http://evmusic.fauno.org/lib/main/
pcstypes",
        "mypcs": "http://evmusic.fauno.org/
lib/jesus/pcstypes",
        "mymotives":
"http://evmusic.fauno.org/lib/jesus/motives"
        },
    "def":{
        "ma": {"objclass":"motive",
            "symbol":[  0,7, 5,4,2,0  ],
            "slength": "+-+- +-++ +---"},
        "flamenco": {"objclass":"pcstype",
            "pcs":[  0,5,7,13 ],},
        },
    "orc":{
        "flauta": {"objclass":"instrument",

"value":"x.lib.instruments.flute",
            "role":"r1"}
        "cello": {"objclass":"instrument",

"value":"x.lib.instruments.cello",
            "role":"r2"}
        },
    "score":{
        "objclass": "score",
        "pos": 0,
        "events": [{
            "objclass": "section",
            "pos": 1,
            "events": [{
              "objclass": "staff",
              "name": "flauta",
              "pspellorder": 1,
              "events": [{
                "objclass": "part",
                "track": 1,
                "pos": 1,
                "events": [{
                  "objclass": "note",
                  "pos": 0,
                  "dur": "0.5",
                  "pitch": 62,
                  "art": "st"
                },
                ...

            ... ]},
            {"objclass":"section","pos": 60,
        ...
    },],}}}
}}
```

The code shows four sections in the content. The second section named *lib* is a dictionary of libraries to be loaded. Both main and user libraries can be addressed. The following section includes local definitions of objects. As an example, a *motive* and a *chord type* are defined. Next section establishes instrumentation assignments by means of the arrangement object *role*. Last section is the score itself, where all events are placed in a tree structure using *parts*. Using MusicJSON as the intermediary communication format enables us to connect several music services forming a cloud composition system.

5 . CONCLUSION

This paper puts forward a model of musical representation for the Computer Music Cloud, a new paradigm in which musical computing systems are distributed in several musical services over a computing cloud. This new environment allows to build new systems by putting various services to work together. We present a new architecture which allows to design specialized autonomous musical services that can be implemented separately in different platforms, and may even serve multiple systems simultaneously.

Effective integration of such musical services in the CMC depends on the definition of a music representation that they all share and that will enable efficient exchange of musical information. Musical representation should fulfill three main requirements:1) to have a high and flexible representativity for music composition, 2) to provide a form allowing music objects to be stored as entries of a database; and 3) to count on a text format that facilitates information exchange, as well as the integration of different data sources.

Our new proposal is based on the robust musical representation for EvMusic composition described in UML which has proved effective in actual composition experiments [1,3]. Above this abstract model of classes, the database form representation and the textual representation MusicJSON have been incorporated.

MusicJSON can represent the complete EvMusic class system together with its different topologies and its comprehensive treatment of musical pitch. In addition to being compatible with conventional musical notation, it can represent higher abstraction structures at multiple levels. It is not only a simple format for exchange of musical objects in text form, but it also integrates musical information from different services. It can also handle references to external objects and libraries. Several examples of use have been shown: representing a music fragment, protocol for sharing musical elements between services, use of libraries and the file format. MusicJSON is based on JSON, an increasingly used format on the Internet. Therefore it inherits its expandability and prototyping features, and benefits from its extensive library of available tools and services.

MusicJSON, EvMusic representation and the database musical format have been tested in real CMC prototypes that incorporate different types of music services [2]. Significantly, it is one of the first proposals for music representation in the new paradigm of Musical Composition in the Cloud. This CMC approach also opens multiple possibilities for derivative work. Once you define an efficient shared music representation for the cloud, any music service can be easily incorporated into the new paradigm: services that translate input and output representations, some applications for collaborative composition among multiple users, musical teaching assistants, and even the integration of true intelligent composition agents. It provides a promising environment for the research in Musical Artificial Intelligence (MAI), where specialised agents can cooperate in a music composition environment sharing the same music representation [4]. Likewise, the paradigm shift that involves the CMC, offers new interesting possibilities for web applications acting as user interfaces in the Computer Music Cloud, taking advantage of new technological developments such as the upcoming HTML5 [14]

REFERENCES

1. Alvaro, J.L., Miranda, E.R., Barros, B. "*Music Knowledge Analysis: Towards an Efficient Representation for Composition*", Current Topics in Artificial Intelligence; LNCS 4177, Springer-Verlag, pp. 331-341(2006)
2. Alvaro, J.L. and Barros, B. "*Composing Music in the Cloud*", Proceedings of the International Symposium on Computer Music Modeling and Retrieval, Málaga 2010
3. Alvaro, J.L. : "*Painting Music with Motives, Twelve Years of Symbolic Pitch Composition*". Under Review
4. Alvaro, J.L. : "*Intelligent Music Clouds*". To be Appeared
5. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. "*Above the Clouds: A Berkeley View of Cloud Computing*" {White Paper} http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.
6. ECMAScript Language Specification, http://www.ecma-international.org/publications/standards/Ecma-262.htm
7. Geelan, J: "*Twenty Experts Define Cloud Computing*", Cloud Computing Journal, SYS-CON Media Inc. (2008) http://cloudcomputing.sys-con.com/node/612375/print
8. Google Apps. http://www.google.com/apps/
9. Ippolito, B. "*Remote JSON - JSONP- JSON with Padding*" (2005) http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/
10. JavaScript. http://en.wikipedia.org/wiki/JavaScript.
11. Introducing JSON: http://www.json.org/
12. Martin, R.C. :"*UML Class Diagrams*" Object Mentor articles (1997) http://www.objectmentor.com/resources/articles/umlClassDiagrams.pdf
13. OMG,. Unified Modeling Language: Superstructure. Version 2.1.1, Retrieved from: <http://www.omg.org/uml>, (2007).
14. W3C "*HTML5 A vocabulary and associated APIs for HTML and XHTML*" W3C Editor's Draft http://dev.w3.org/html5/spec/
15. Walter B. Hewlett and Eleanor Selfridge-Field (eds) *"The Virtual Score"*. MIT Press (2001)