L. Turchet and D. Golyshev, "SMIF: A Format for the Offline Exchange of Smart Musical Instruments Configuration and Data"
*J. Audio Eng. Soc.*, vol. 69, no. 12, pp. 946–955, (2021 December).
DOI: https://doi.org/10.17743/jaes.2021.0030



# SMIF: A Format for the Offline Exchange of Smart Musical Instruments Configuration and Data

**LUCA TURCHET ,*** *AES Associate Member,* **AND DMITRY GOLYSHEV**
(luca.turchet@unitn.it) (dmitry.golyshev@studenti.unitn.it)

*University of Trento, Trento, Italy*

Smart musical instruments (SMIs) are an emerging category of musical instruments characterized by sensors, actuators, wireless connectivity, and embedded intelligence. To date, a topic that has received remarkably little attention in SMIs research is that of defining a file format for the offline exchange of content produced by such instruments. To address this gap, in this paper we propose the Smart Musical Instruments Format (SMIF), a file format specific to smart musical instruments. We also provide an implementation of an encoder, decoder, and player for it. Such a format is not completely fitting any current standard but is strongly inspired by the MPEG-A: Interactive Music Application Format (IM AF). In our implementation we integrated IM AF with tracks related to sensors, MIDI, as well as the representation of the instrument's sound engine via the Smart Musical Instruments Ontology. SMIF enables the creation of novel applications for the offline exchange of SMIs configuration and data, some of which are illustrated in the paper.

## 0 INTRODUCTION

In recent years a novel class of musical instruments has emerged as the intersection of digital musical instruments and Internet of Things devices: the so-called "smart musical instruments" (SMIs) [1]. This family of instruments draws upon different lines of existing research including augmented instruments [2], embedded audio [3, 4], embedded acoustic instruments [5], and networked music performance systems [6, 7]. At hardware level they are characterized by sensors, actuators, wireless connectivity, and on-board processing. These features enable SMIs to directly exchange musically relevant information with one another as well as communicate with a plethora of external devices (such as smartphones, wearables, virtual reality headsets, or stage equipment). Examples of existing smart musical instruments developed in industrial contexts are the Smart Guitars by HyVibe or Elk [8], INSTRUMENT 1 by Artiphon, and gTar by Incident, while instances from academic endeavors are the Smart Mandolin [9] and Smart Cajón [10].

SMIs are instances of Musical Things within the "Internet of Musical Things" (IoMusT) paradigm [11], an extension of the Internet of Things [12] to the musical domain. Within this paradigm, SMIs can exchange content with other Musical Things leveraging application and services built on top of the connectivity infrastructure. According to the vision proposed in [1], an SMI is characterized by five core capabilities that define its embedded intelligence: i) knowledge management, i.e., the capability of maintaining knowledge about itself and the environment; ii) reasoning, i.e., the capability of making inferences on the acquired knowledge; iii) learning, i.e., the capability of learning from previous experience; iv) human-smart instrument interaction, i.e., the capability of interacting with the player in ways that extend the bare sound production, such as adaptation and proactivity; v) smart instrument-Musical Things interaction, i.e., the capability of wirelessly exchanging information with a diverse network of Musical Things.

To date, a topic that has received remarkably little attention in SMIs research is that of defining a file format for the exchange of content produced by such instruments, in both offline and online scenarios. In the real-time case, content generated by an SMI can be streamed continuously toward a destination, whereas in the offline case the content can be saved in a format and then shared subsequently. The present study focuses on the offline scenario. In previous work we preliminarily investigated the design of a format specific to SMIs in offline scenarios [13]. We adopted a participatory design methodology consisting of a set of interviews with studio producers familiar with the smart instruments con-

*Corresponding author: Luca Turchet; E-mail: luca.turchet@unitn.it

cept. The purpose of such interviews was that of identifying a set of use cases for a format encoding data generated by SMIs, with the end goal of gathering requirements for its design. Such work identified in the IM AF (MPEG-A: Interactive Music Application Format) [14, 15] a potential candidate to implement a file format for SMIs.

IM AF is a multitrack format that allows users to mix individual tracks for different musical instruments by adjusting their volume and enables the association of synchronized text (e.g., for lyrics). Moreover, it supports a set of presets for the mixes created by the producer as well as a user mixing mode (optionally encompassing interactivity rules). Specifically, an IM AF file consists of: i) multiple audio tracks: they contain the audio signal related to each instrument or voice and may be encoded either in 2D or 3D spatial audio; ii) groups of audio tracks: the definition of the structure of the audio tracks into groups; iii) preset data: a set of pre-defined mixes of the multiple tracks; iv) user mixing data and interactivity rules: this information relates to the interaction of the user with the mixing parameters in the file; v) metadata: this is data related to the music contained in the file and can be both static (e.g., information about the song or album and still pictures) and time-dependent (e.g., the synchronized lyrics).

The structure of an IM AF file is derived from the ISO-Base Media File Format standard (ISOBMFF),[1] the most widely deployed standard in the music[2] [14, 15] and media industry (aka .mp4). Another standard that is ISOBMFF-compatible is MPEG-V.[3] This format is conceived to support, among other things, the encoding of sensors data.

This paper investigates the design and implementation of an encoder and decoder for a file format for content related to SMIs, starting from the requirements reported in [13]. The proposed format is called the Smart Musical Instruments Format (SMIF) and has extension *.smi*. The format is based on the IM AF and integrates features from ISOBMFF related to sensors and includes the configuration of an SMI. Such a configuration adheres to the Smart Musical Instruments Ontology [16], which allows one to represent the hardware and software components of an SMI. We clarify that the proposed format does not fit any current standard.

Different ways exist today for storing heterogeneous data for musical performance. For instance, Digital Audio Workstations, such as Ableton Live, and computer music platforms, such as Pure Data [17], can be used to encode a musical performance or project involving audio, MIDI, and automation tracks. However this process on Digital Audio Workstations requires some manual work to set up and cannot be embedded, while on Pure Data it can be embedded but is a time-consuming endeavor. Nevertheless the proposed format is conceived specifically for SMIs (which are based on embedded systems) and has the advantage to improve the experience of setting up the encoding process compared to existing approaches.

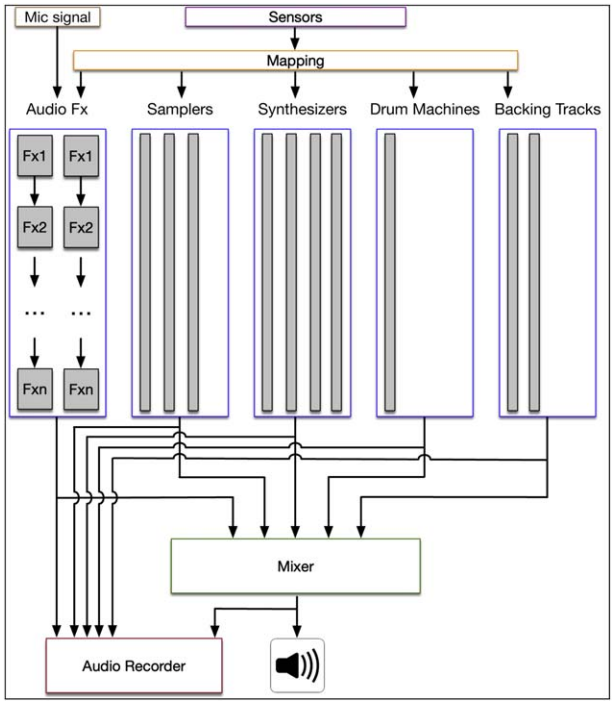


Fig. 1. Block diagram of an example of sound engine running on a smart musical instrument. The gray areas indicate the various sub-components (e.g., separate tracks of effects chains or different parallel instances of synthesizers).

## 1 BACKGROUND

### 1.1 SMI's Sound Engine

The sound engine of an SMI is responsible for the generation of the instrument's digital sounds and may encompass various components (see Fig. 1). For instance, a component can process the sounds detected by a microphone by applying digital audio effects to it, trigger sound samples thanks to a sampler, generate sounds resulting from the control of synthesizers and drum machines, and play back different backing tracks. The parameters of each of these components of the sound engine can be modulated by the sensors present in the sensor interface by means of a set of mapping rules [18]. The sound engine is also responsible for recording the overall sound resulting from the mixing of all such components but can also record the contribution of each component in separate files.

### 1.2 The Smart Musical Instruments Ontology

The Smart Musical Instruments Ontology[4] was conceived to address interoperability issues of heterogeneous SMIs exchanging information between each other [16]. The ontology models the basic concepts of the hardware and software components of an SMI and relates to several existing ontologies. These include the SOSA Ontology for the

[1]ISO/IEC 14496-12:2015 "Information Technology – Coding of Audio-Visual Objects – Part 12: ISO Base Media File Format"

[2]E.g., www.stems-music.com/stems-partners

[3]ISO/IEC 23005-1:2020, "Information Technology – Media Context and Control (MPEG-V) – Part 1: Architecture"

[4]https://w3id.org/smi#

representation of sensors and actuators [19], Audio Effect Ontology dealing with the description of digital audio effects [20], Music Ontology that deals with the description of the music value-chain from production to consumption [21], Studio Ontology for representing the domain of technical workflows occurring in music production [22], and IoMusT Ontology for the representation of Musical Things and IoMusT ecosystems [23].

In more detail, the requirements for the ontology were the following. The SMI Ontology should be able to: i) represent the concept of SMIs as an instance of Musical Things, including its type, characteristics (including the number and type of inputs and outputs), the structure of its sound engine, and the geographical position; ii) represent the concept of application and service related to SMIs, including its purpose, level of interactivity, type, and user; and iii) describe attributes of the music at a given time, including low and high-level features. A practical use of the ontology in an applicative context has been recently reported in [24] for the case of the creation of a database of SMIs.

## 1.3 Use Cases

The following main use cases for the offline exchange of content generated by an SMI were identified in [13].

### *1.3.1 Advanced Studio Productions*

A format encompassing various kinds of information related to an SMI's affordances would be useful in contexts of studio production as it could enable novel ways to edit a recording generated by a smart instrument. In addition to the conventional recording of the instrument into an audio file, a new format specific to SMIs may encompass information relating to different aspects of the instrument, such as the sensors signals, different audio tracks generated by the sound engine, or structure of the sound engine.

A studio producer could interact with these new levels of information in order to create a modified version of the original recording. For instance, the format could enable:

- The mixing of the various audio tracks related to the different components of the sound engine of the smart instrument (e.g., the instrument signal processed with effects, synthesizers, samplers, and backing tracks);
- The application of effects to the audio tracks corresponding to the various components of the sound engine; or
- The modification of the mappings between sensors values and the parameters of the sound engine (e.g., a sound sample triggered by a sensor in the instrument's sensor interface could be substituted by another sound sample; the sensor associated to a parameter of an effect could be associated to a parameter of another effect).

### *1.3.2 Smart Instrument Configuration*

As the format encompasses all information related to the configuration of an SMI, by decoding the format content, the settings regulating the behavior of a certain SMI could be used to configure an instrument of the same kind (i.e., the format is used as a preset). For instance, a format saved in a certain SMI and loaded into another SMI having the same characteristics could configure:

- The structure of the sound engine: this may include which components are present, such as the chain of the sound effects applied to a microphone or the number, type, brand, and model of the involved digital synthesizers, samplers, or drum machines;
- The mappings between the sensors and sound engine parameters; or
- A set of parameters regulating the behavior of the instrument, such as the sampling rate of the audio tracks and sensor values, beats per minute of the backing tracks, and initial values of all the parameters of the sound engine.

### *1.3.3 Score Information Exploitation*

The file format encompasses the MIDI score of each component of the sound engine, which may include the notes generated by the drum machine and those generated by the synthesizers, as well as the notes generated by the acoustic instrument. The latter could be achieved by means of automatic transcription techniques (see, e.g., [25]) and also include information related to a certain playing technique associated to each note (see, e.g., [10]). The information about the score could be useful for composition purposes, where the content of the recorded file could be extended with additional tracks composed on the basis of the provided score.

### *1.3.4 Learning and Training*

A decoder and player for the format (for both the cases in which it is placed inside the instrument and on an external device such as a laptop) could be used for learning and training purposes. For instance, a player of a certain SMI could load the file format of a particular music piece, mute all (or some of) the interactive tracks (i.e., the track of the recorded instrument and those resulting from the interaction with the sensors), and play over the remaining tracks (e.g., the backing track) to practice the piece in all (or some of) the interactive parts.

### *1.3.5 Mulsemedia Reproductions*

The decoder and player, for instance running on a PC or smartphone, could be used in conjunction with Musical Things providing additional sensory content to the music played. This use case relates to contexts of mulsemedia (i.e., multi-sensory media) applications [26] within the Internet of Musical Things paradigm, where a smart musical instrument is used to control in real time Musical Things aiming at enriching the audience's musical experience with content involving other sensory modalities, such as visual content

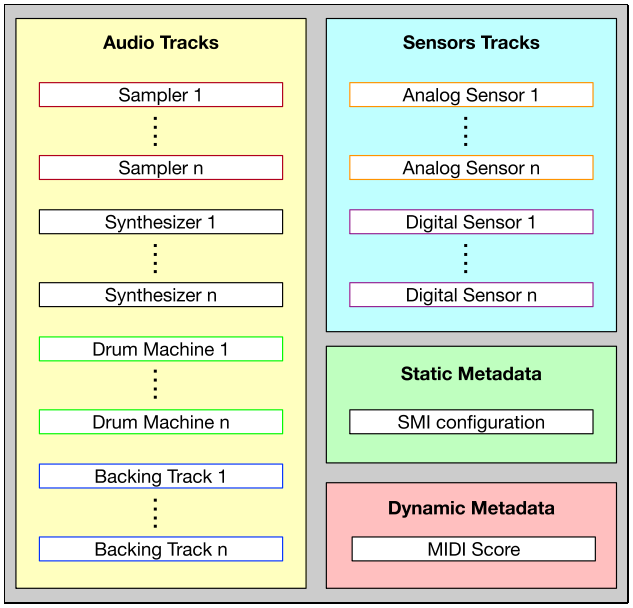


Fig. 2. The Smart Musical Instruments Format structure.

(e.g., delivered on conventional screens or head-mounted displays) and haptic content (e.g., delivered by musical haptic wearables [27]). The additional content should be perfectly synchronized with the musical content of the file format.

## 2 REQUIREMENTS

### 2.1 Requirements for the Format

The following design requirements for the specification of the file format were identified in [13]. SMIF, see Fig. 2, shall contain:

- **Audio tracks:** these are related to the outputs generated by each of the components of the sound engine as well as the overall output;
- **Sensor tracks:** the values of each sensor (both analog and digital) present in the sensor interface are represented;
- **Static metadata:** these are related to the instrument and its configuration, including i) type, brand, and model of the instrument; ii) information about the sensor interface (including how many sensors, which type); iii) the structure of the sound engine (including which components are present, the chain of effects, and the brand and model of the audio plugins utilized); iv) the sensors-to-sound parameters mapping, including the range of variation of each controlled parameter, type of mapping (e.g., one-to-one, one-to-many, etc.), and type of mapping function (e.g., linear, logarithmic, exponential, ad-hoc, etc.); and v) the sampling rate for audio and sensor tracks; and
- **Dynamic metadata (i.e., time-based):** these are related to i) which subcomponents of each component are active at a given time (e.g., in a section of a musical piece a synthesizer is active, while in another section it is not); ii) the MIDI score of the piece, including additional information related to each note such as the type of gesture that generated it; and iii) the beats per minute of the eventual backing track.

SMIF shall support:

- **Mixing:** the audio tracks can be muted, removed, or substituted, and their individual volume can be adjusted and effects can be applied to them;
- **Change of structure of the sound engine:** each of the components and subcomponents of the sound engine can be modified (e.g., a certain sound effect plugin can be substituted with another one);
- **Change of mapping:** each of the mappings can be modified in all its parts, including the substitution of the controlled parameter and its rage of variation as well as the associated mapping function; and
- **Interaction with other Musical Things:** the reproduced audio content should be synchronized with the multisensory content delivered by connected Musical Things.

### 2.2 Requirements for the Encoder

The encoder is responsible for creating the .smi file. It was primarily conceived to run on the embedded system of the SMI producing the content. A secondary use case for it was the creation of an .smi file from the various tracks and metadata modified on a desktop PC according to the use cases identified in Sec. 1.3. This led to the requirement of an encoder capable of running on different platforms.

Furthermore, an .smi file should be encoded by encompassing various audio formats (e.g., pulse-code modulation [PCM] and mp3) with different sample rates and resolutions (e.g., 44.1 kHz at 16 bits). Sensor tracks should be encoded in a raw format such as PCM, with a much lower sample rate than the audio tracks (e.g., 1 kHz), with resolution of 10 to 16 bits. The metadata related to the instrument configuration should adhere to the RDF data model (e.g., a plain text with Turtle syntax), while metadata related to the MIDI score should be encoded as a polyphonic MIDI file.

### 2.3 Requirements for the Decoder and Player

The decoder was devised to accomplish various purposes. The first was that of running directly on the SMI following the reception of an .smi file on it, e.g., sent from an external device. In this case the decoder is used to extract the instrument configuration metadata as well as the backing tracks in order to feed another program with them, which configures the instrument, i.e., the received .smi file would act as a preset.

The second purpose of the decoder was that of being used together with a player. We considered three approaches all related to a cross-browser and cross-platform web app (in part inspired to the IM AF player reported in [28]):

1. **Client-side (local)**: a simple web page running locally on a desktop PC;

2. **Client-side (with server)**: a web app responsible for the decoding of the .smi file (used in conjunction with a server that sends the .smi file to it); and
3. **Server-side**: a web app used in conjunction with a server, where the server is responsible for the decoding of the .smi file.

All three approaches should be able to read an .smi file and perform the basic functionalities of the format such as mixing (e.g., enabling/disabling an audio track and changing its volume) as well as reading the sensor tracks and forwarding their values to a port of the local machine (another software would listen to such a port and repurpose the sensors values, e.g., to drive synthesizers or control mulsemedia applications). To implement the latter, the Open Sound Control (OSC) protocol was chosen.

## 3 IMPLEMENTATION OF THE ENCODER

The creation of the encoder started from the one described in [15], amending it to accommodate the SMIF features. It has been implemented in the C programming language and does not use external libraries. In this way we can ensure that it can be compiled on any platform. The following paragraphs explain the implementation of the encoder.

### 3.1 Audio Tracks

As in [15] the File Type Box *ftyp* describes the characteristics of the audio tracks, containing the same values of the IM AF format. We implemented the im02 brand that supports up to six simultaneous tracks of audio, so the SMI file can be reproduced even with limited processing capabilities, such as a mobile device. The total size of this box is 24 bits.

Differently from the previous implementation, to use the same format of an SMI output a PCM format audio track is encoded. The encoder reads a .wav file, searching for information about the samples in the WAVE header at the beginning of the file. After the interpretation of the information such as sample rate and size, all the samples are stored in the Media Data Box *mdat*. For each track the chunk position in the *mdat* box is saved for boxes described below. The audio tracks are saved in sequence with their corresponding sensor tracks (e.g., the data sequence in *mdat* with an audio track with two sensors and another with one sensor will be: audio track 1 – sensor track 1 – sensor track 2 – audio track 2 – sensor track 3). Further information about the samples will be stored in the Sample Table Box *stbl*.

An SMI file stores the audio tracks as specified by the IM AF standard for PCM format tracks. The internal timescale stored in the Media Header Box *mdhd* is set to the original .wav file sample rate. This means that the *stbl* box the Time To Sample Box *stts* has the time delta set to 1 and sample count is equal to the original file sample number; the Sample To Size Box *stsz* values contain only one entry of sample size corresponding to the size of original track samples for each channel. Finally, the Sample Description Box *stsd* stores the remaining technical specifications of the audio tracks.

The position of the media data is stored in the Chunk Offset Box *stco* and only one chunk is written in the Sample To Chunk Box *stsc*, as the PCM format has a constant bit rate and all the frames have equal size.

### 3.2 Sensor Tracks

An important feature of the SMI format is the encoding of sensor tracks. They are recorded simultaneously along the SMI audio track in separate tracks, each at a 1-kHz sample rate mono channel, with float values ranging from 0.000 to 1.000. Each value is encoded in unsigned 16-bit integers by multiplying the original number by the largest uint16 number (65,535) and stored in a .wav file with a regular WAVE header at the beginning that specifies the sample information.

In a similar way to audio tracks, sensor track details are stored in Track Boxes *trak* but with some values set corresponding to the Timed Metadata type of track. The media data is written in the Media Data Box *mdat* immediately after the referenced audio track. The Track Header Box *tkhd* has the volume value set to 0 and the Handler Box *hdlr* has name and handler type values set to "meta."

The Track Box contains an additional Track Reference Box *tref*, which contains a Track Reference Type Box set to *cdsc*, indicating a reference track. The ID of the SMI track with which the corresponding sensor track was recorded is stored in the track ID reference field.

The Sample Description Box *stsd* contains a Sample Description Table Box with type set to *mebx* that in turn stores Metadata Key Table Box keys. In this table a Metadata Key Box is defined and its type is set to "*sns*" plus the number of the sensor track (e.g., "*sns1*" for sensor track 1), allowing up to nine simultaneous sensors in one file. This definition can be changed to "sn" plus two digits, allowing up to 99 sensors (sn01 to sn99).

Then two additional boxes define the metadata key characteristics: First, Metadata Key Declaration Box *keyd* specifies the type of sensor. For now key namespace is set to "*mdta*," which indicates a string format for the key value, which is set to "*snsr*." This value can be extended to specify the model or type of the sensor.

Second, the data format described in the WAVE header of the sensor tracks is stored in the Metadata Datatype Definition *dtyp*, which sets the data namespace to 0, indicating a well-known value of metadata, and datatype value to 76, representing an unsigned big-endian 16-bit integer.

### 3.3 Static Metadata

There are two types of static metadata included in an .smi file:

- Metadata about groups of tracks present in the file; and
- Metadata related to the instrument configuration with Turtle syntax.

Both boxes are contained at *moov* level, as they specify information relative to the entire file and and are stored after all the audio and sensor tracks.

At first, Group Container Box *grco* is implemented as in the IM AF standard. If both types of tracks are set to be encoded by the user, two groups will be created: the first contains all the audio tracks and the second contains the sensor tracks. For each group a Group Box *grup* stores relative *trak* IDs and sets activation mode value to 1 and reference volume to 1 for audio tracks and 0 for sensor tracks. The name and description fields store additional information representing those groups.

The second box is a Metadata Box *meta* that, similarly to a common metadata box, contains additional information about the entire track. In this case the Handler Box *hdlr* of *meta* is set to handler type "*meta*" and name "*ttl*," to indicate a .turtle specification file. Next to the Handler Box, a Turtle (TTL) Box *ttl* is created. It contains three fields, corresponding to size (32 bits), type ("*ttl*," 32 bits), and version (32 bits), plus the data field containing the entire turtle preset file. It is set to be statically read from the file "preset.ttl" in the same folder of the encoder if the user accepts to include it in the Terminal dialog.

### 3.4 Dynamic Metadata

The dynamic metadata featured in the SMI file corresponds to a MIDI polyphonic representation of some or all the audio tracks in the file. The actual media data is stored from the .mid file in the *mdat* box, while the information about its type and position is stored next to the static metadata boxes.

Another Metadata Box *meta* is created at *moov* level, with a Handler Box *hdlr* set to handler type "*meta*" and name "*midi*" to indicate a MIDI type of dynamic metadata. Next, Item Location Box *iloc* stores the position of the media data in the base offset field and the size of the entire data in the extent length field; last, the Item Info Box *iinf* contains an Item Info Entry Box *infe* that specifies the type of metadata item. Item name field is set to "score," content type to MIME type "audio/midi," and content encoding to "mid." Table 1 lists the components supported by SMIF for formatting the aforementioned data types.

## 4 IMPLEMENTATION OF THE DECODER AND PLAYER

This section details the implementation of an SMIF decoder as a standalone command line program, as well as three versions of an SMIF player that also integrates an SMIF decoder (using web technologies). For these purposes we followed the requirements described in Sec. 2.3. The three approaches for joint decoding and playing and their respective technologies are illustrated in Fig. 3. Fig. 4 details the SMIF architecture in relation to a player.

### 4.1 Decoder

A command line version of the decoder for .smi files has been developed in C without the use of any external libraries, ensuring compatibility across various operating systems. The program is composed of two parts: the decoder and SMI header file used in the encoding process. The header defines all the structures of the boxes representing the entire SMI file. The program can properly decode only files encoded by the SMI encoder.

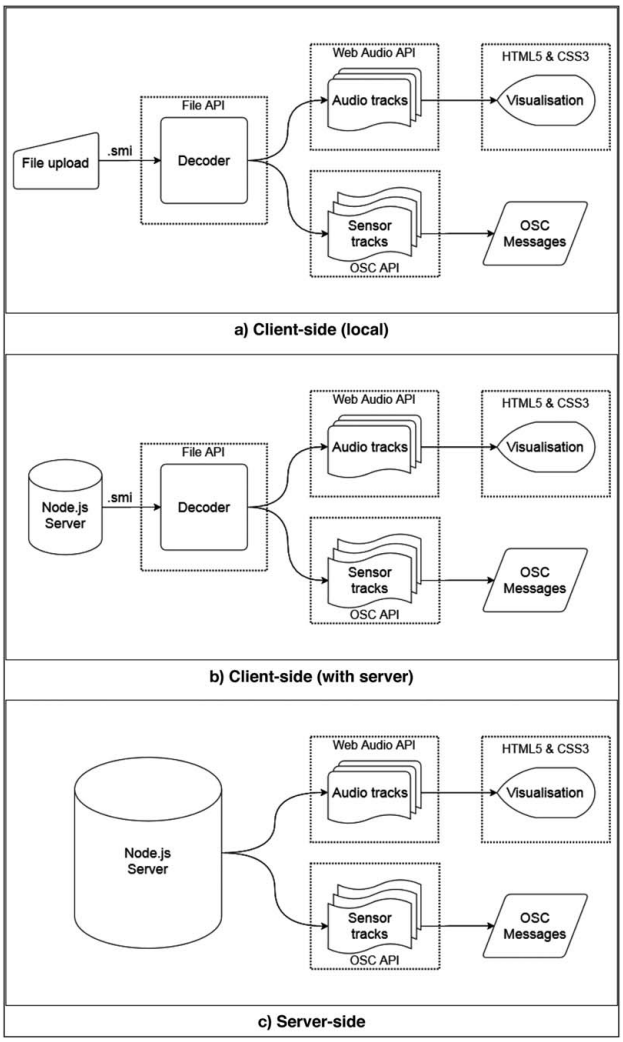


Fig. 3. (a) Client-side (local), (b) client-side (with server), and (c) server-side Smart Musical Instruments Format decoding diagrams with their respective technologies used.

The main function of the decoder requires an argument passed in the command line while calling the compiled program, corresponding to the file name of the SMIF file. Every box is parsed in order, and if the main boxes (such as *ftyp*, *mdat*, and *moov*) are present the decoding process starts. At the *moov* level all the *trak* boxes are read and from the *tkhd* box volume field of each track (1 for audio, 0 for sensors) the decoder switches between an audio and sensor decoding type of function. In both cases the parser searches for the information about the samples and their corresponding location in the boxes used by the encoder, as described in Sec. 3.

The main information is stored in the *stco* (chunk offset position), *stsd* (sample description), and *stsz* (sample size)

Table 1. Supported components in SMIF (in bold the formats currently implemented).

| Type | Component Name | Abbreviation | Specification |
|---|---|---|---|
| File Format | **ISO Based Media File Format** | **ISO-BMFF** | **ISO/IEC 14496-12:2008** |
| Audio | MPEG-4 Audio AAC Profile | AAC | ISO/IEC 14496–3:2005 |
| | MPEG-D SAOC | SAOC | ISO/IEC 23003–2:2010 |
| | MPEG-1 Audio Layer III | MP3 | ISO/IEC 11172–3:1993 |
| | **PCM** | **PCM** | ... |
| Sensor | **PCM** | **PCM** | ... |
| Static Metadata | **Turtle Resource Description Framework Data Model** | **TTL** | **RDF1.1 REC 25/02/2014** |
| Dynamic Metadata | **Musical Instrument Digital Interface** | **MIDI** | **ISO/IEC 14772-1:1997** |

AAC, Advanced audio coding; PCM, Pulse-code modulation; SAOC, Spacial Audio Object Coding.

boxes. The information from sample description and size is used to create a new WAVE header that will be concatenated to the media data. Chunk offset position indicates the media data location in the *mdat* box. The size of the chunk is determined from the information about the samples and all of the data is written to the new file. In the case of an audio file the name generated is "a" plus the number of the audio track (e.g., "a1.wav"); in the case of a sensor file it corresponds to "s" plus the number of the referenced audio track and number of the sensor track separated by an underscore (e.g., "s1_1.wav" in the case of the first sensor referencing the first audio track).

Once all the *trak* boxes are parsed and decoded, the parser searches for a group container, where the audio and sensor groups are defined by indicating their corresponding IDs. Subsequently, the *meta* boxes at *moov* level are analyzed. If the handler name is set to "*ttl*," the decoder tries to create a Turtle RDF file, reading the information right below the size and type fields of the turtle box, placed next to the handler box; if the handler name is set to "midi," the decoder reads the information about the location in the *mdat* box and size of the MIDI polyphonic file from the Item Location Box. In both cases the data from the original metadata files is stored entirely in the SMI boxes, so no additional information needs to be added by the decoder. The two optional files are then written respectively as "preset.ttl" and "midi.mid" and saved along the rest of the tracks in the output directory.

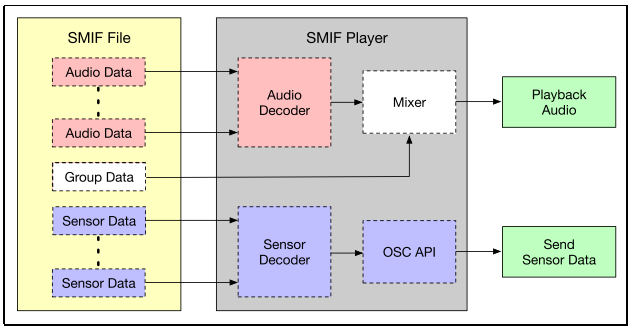


Fig. 4. Smart Musical Instruments Format (SMIF) architecture: (a) the different types of media data supported in the SMIF file format are shown; (b) the SMIF player is shown, including the corresponding decoders/parsers of the media data in the SMIF file format.

## 4.2 Player

With respect to the requirements mentioned in Sec. 2.3 and the aforementioned discussion on the related options considered for a cross-platform and cross-browser SMIF player implementation, we adopted and extended the IM AF player approach reported in [28]. The main extension of the player is the usage of Node.js technology to create the server that decodes (in the server-side case) the .smi file and listens to incoming OSC messages, as well as the reliance on external APIs and modules with the "require" method. Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser. The WebSocket and Express modules are used to host the web server.

To send and receive messages in OSC format using values generated by the sensor data of the .smi file, we used the osc.js library. In the client-side version it was incorporated inside the web page, while in the server-side case it was imported by the Node.js server. Other Node.js internal libraries are used by the server, such as "file-system," "child-process," and "body-parser" to read the file and execute the compiled C decoder described above.

All three approaches are able to decode an .smi file and perform the basic functionalities of the format, as well as read the sensor tracks and forward their values to a port of the local machine via OSC. Regarding the latter, the values of the sensor tracks are passed as a float argument, describing the state of a given sensor, and sent as an OSC message to an address specified by "/sensor_#."

### *4.2.1 Client-Side (Local)*

The first version of the player is implemented entirely as a locally running web page, using HTML5, CSS, and JavaScript. The page consists in a file input form and the empty player, with a play/stop button as well as a box containing title and time information. In this case the decoder is implemented as a JavaScript parser that reads all the tracks in the file and stores the corresponding information in memory. A Web Audio buffer is created and the waveforms of each audio track are plotted alongside their volume sliders, enabling easy tracks mixing by the user; while the data of each sensor track is prepared to be sent as an OSC message to the localhost via a user datagram protocol (UDP) broadcast.

#### 4.2.2 Client-Side (With Server)

The second version is a minor variation of the local web page, consisting of the possibility to request and download the .smi file from the server. After downloading, the process of the player is the same as in the first case. The OSC messages sent by the local machine are then received by the server, which can repurpose it for the wanted application.

#### 4.2.3 Server-Side

The third version of the SMI player relies heavily on the Node.js server. The web server contains the .smi file and C version of the decoder in its local directory; the local page is much lighter than the first two versions and it initially allows only the uploading of the .smi file. When the user inputs the file, the server starts the decoding process by executing the compiled version of the C decoder and redirects the user to the player page, where the tracks are plotted as the audio files are created. Then each audio and sensor buffer is returned to the web application, allowing the user to play the audio data and send the sensor data as OSC messages over a UDP broadcast.

## 5 DISCUSSION AND CONCLUSION

In this paper we built an application around IM AF, being aware that our work is not fitting in any current standard. Some components of SMIF are compliant with ISOBMFF, namely the audio and sensor tracks. Moreover we adopted the IM AF's grouping feature to group sensor and audio tracks (which can be activated or deactivated by means of a toggle in the player). We also implemented the IM AF's mixing capability for modifying the volume of the audio tracks, so the end users via one of the provided players can create their own mix, in accordance with the requirements that were defined.

Components that are not compliant are the dynamic metadata for the SMI's engine configuration (in the form of a Turtle file) and polyphonic MIDI scores (in the form of a MIDI file). Differently from IM AF, we did not include in SMIF components such as images and lyrics. We also did not involve interactivity rules that characterize IM AF (such as those for presets, groups, and mixing, which are imposed by the music composers with the aim of fitting their artistic creation). These could be the object of future implementations if we notice the need to reformulate our requirements for SMIF.

We conducted a number of tests to find out the number of audio and sensor tracks that the application is capable of decoding and playing at the same time. Both desktop browsers that have been tested (Chromium and Firefox) allow up to 16 simultaneously decoded audio tracks, although at higher numbers of total tracks the playback quality drops considerably. As for the sensor tracks, the goal was set to eight simultaneous tracks, but the application could not sustain the preset packet rate of 1 kHz per track. Plausibly the reason behind this behavior may be that the osc.js library is too heavy or browsers tested do not support such rates. With lower sample rates (e.g., 100 packets per second), the player successfully sends up to four sensor tracks to localhost synchronized to the audio using the JavaScript *setInterval timer*, resulting in a good quality playback in the case of eight audio tracks.

The limitations of the encoder are given by the 4CC tags of the sensor names ("sn01" to "sn99") and, in the case of very large files, Chunk Offset and Chunk Size values. Being 32-bit unsigned integer fields, offset and size larger than UINT32_MAX have to be set to a larger variable (64 bits), as for the ISOBMFF specification (currently not implemented); this means that up to 99 sensors and 16 audio tracks can be encoded simultaneously, with a maximum total file size of about 4 gigabytes.

These results are in full agreement with the maximum number of simultaneously decoded audio tracks specified in the IM AF standard, where up to 16 tracks are supported for desktop applications (im11), and are within the limit of encoder and decoder features, while for the limits related to sensors, a deeper investigation is required to identify the most efficient and optimized technology that can send OSC packets at higher rates.

As the fields of the IoMusT and SMIs emerge, there is a need for realizing standardization activities, given their crucial role for enabling interoperability. However today standardization activities are in vast part unrealized within these two fields [11]. The SMIF format, encoder, and decoder reported in this paper represent a preliminary step toward a discussion about standardizations related to the emerging family of smart musical instruments. The proposed format for the exchange of content generated by a smart instrument would enable novel interactive applications such as the one mentioned in Sec. 1.3.

It is worth noticing that this paper focused on a format for individual instruments while the focus could also be directed toward the exchange of files containing content from multiple smart musical instruments. This matter will be the object of our future investigations. The code and documentation of this study are freely available online.[5]

## 6 REFERENCES

[1] L. Turchet, "Smart Musical Instruments: Vision, Design Principles, and Future Directions," *IEEE Access*, vol. 7, pp. 8944–8963 (2019 Jan.). https://doi.org/10.1109/ACCESS.2018.2876891.

[2] E. R. Miranda and M. M. Wanderley, *New Digital Musical Instruments: Control and Interaction Beyond the Keyboard*, Computer Music and Digital Audio Series, vol. 21 (A-R Editions, Inc., Middleton, WI, 2006).

[3] A. McPherson and V. Zappi, "An Environment for Submillisecond-Latency Audio and Sensor Processing on BeagleBone Black," presented at the *138th Convention of the Audio Engineering Society* (2015 May), paper 9331. http://www.aes.org/e-lib/browse.cfm?elib=17755.

[4] L. Turchet and C. Fischione, "Elk Audio OS: An Open Source Operating System for the Internet of Musical

[5] https://github.com/Ioshiro/smiencoder-poc

Things," *ACM Trans. Internet Things*, vol. 2, no. 2, pp. 1–18 (2021 Mar.).

[5] E. Berdahl, "How to Make Embedded Acoustic Instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 140–143 (London, UK) (2014 Jul.).

[6] C. Rottondi, C. Chafe, C. Allocchio, and A. Sarti, "An Overview on Networked Music Performance Technologies," *IEEE Access*, vol. 4, pp. 8823–8843 (2016 Dec.). https://doi.org/10.1109/ACCESS.2016.2628440.

[7] L. Gabrielli and S. Squartini, *Wireless Networked Music Performance*, Springer Briefs in Electrical and Computer Engineering, (Springer, Singapore, Singapore, 2016). https://doi.org/10.1007/978-981-10-0335-6.

[8] L. Turchet, M. Benincaso, and C. Fischione, "Examples of Use Cases With Smart Instruments," in *Proceedings of the 12th International Audio Mostly Conference on Augmented and Participatory Sound and Music Experiences*, paper 47 (London, UK) (2017 Aug.). https://doi.org/10.1145/3123514.3123553.

[9] L. Turchet, "Smart Mandolin: Autobiographical Design, Implementation, Use Cases, and Lessons Learned," in *Proceedings of the Audio Mostly Conference on Sound in Immersion and Emotion*, paper 13 (Wrexham, UK) (2018 Sep.). http://doi.acm.org/10.1145/3243274.3243280.

[10] L. Turchet, A. McPherson, and M. Barthet, "Real-Time Hit Classification in a Smart Cajón," *Front. ICT*, vol. 5, paper 16 (2018 Jul.). https://doi.org/10.3389/fict.2018.00016.

[11] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet, "Internet of Musical Things: Vision and Challenges," *IEEE Access*, vol. 6, pp. 61994–62017 (2018 Sep.). https://doi.org/10.1109/ACCESS.2018.2872625.

[12] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Comput. Networks*, vol. 54, no. 15, pp. 2787–2805 (2010 Oct.). https://doi.org/10.1016/j.comnet.2010.05.010.

[13] L. Turchet and P. Kudumakis, "Requirements for a File Format for Smart Musical Instruments," in *Proceedings of the International Workshop on Multilayer Music Representation and Processing*, pp. 5–9 (Milan, Italy) (2019 Jan.). https://doi.org/10.1109/MMRP.2019.8665380.

[14] I. Jang, P. Kudumakis, M. B. Sandler, and K. Kang, "The MPEG Interactive Music Application Format Standard [Standards in a Nutshell]," *IEEE Signal Process. Mag.*, vol. 28, no. 1, pp. 150–154 (2011 Jan.).

[15] J. C. García, P. Kudumakis, I. Barbancho, L. J. Tardón, and M. Sandler, "Enabling Interactive and Interoperable Semantic Music Applications," in Rolf Bader (Ed.), *Springer Handbook of Systematic Musicology*, Springer Handbooks, pp. 911–921 (Springer, Cham, Switzerland, 2018).

[16] L. Turchet, P. Bouquet, A. Molinari, and G. Fazekas, "The Smart Musical Instruments Ontology," *J. Web Semant.* (in press).

[17] M. S. Puckette, T. Apel, and D. D. Ziccarelli, "Real-Time Audio Analysis Tools for Pd and MSP," in *Proceedings of the International Computer Music Conference* (Ann Arbor, MI) (1998 Oct.).

[18] A. Hunt, M. M. Wanderley, and M. Paradis, "The Importance of Parameter Mapping in Electronic Instrument Design," in *Proceedings of the Conference on New Interfaces for Musical Expression* (Dublin, Ireland) (2002 May).

[19] K. Janowicz, A. Haller, S. J. D. Cox, D. Le Phuoc, and M. Lefrançois, "SOSA: A Lightweight Ontology for Sensors, Observations, Samples, and Actuators," *J. Web Semant.*, vol. 56, pp. 1–10 (2019 May).

[20] T. Wilmering, G. Fazekas, and M. B. Sandler, "AUFX-O: Novel Methods for the Representation of Audio Processing Workflows," in *Proceedings of the 15th International Semantic Web Conference*, pp. 229–237 (Kobe, Japan) (2016 Oct.). https://doi.org/10.1007/978-3-319-46547-0_24.

[21] Y. Raimond, S. Abdallah, M. B. Sandler, and F. Giasson, "The Music Ontology," in *Proceedings of the 8th International Society for Music Information Retrieval Conference*, pp. 417–422 (Vienna, Austria) (2007 Sep.).

[22] G. Fazekas and M. B. Sandler, "The Studio Ontology Framework," in *Proceedings of the 12th International Society for Music Information Retrieval Conference*, pp. 471–476 (Miami, FL) (2011 Oct.).

[23] L. Turchet, F. Antoniazzi, F. Viola, F. Giunchiglia, and G. Fazekas, "The Internet of Musical Things Ontology," *J. Web Semant.*, vol. 60, p. 100548 (2020 Jan.). https://doi.org/10.1016/j.websem.2020.100548.

[24] L. Turchet, G. Zhu, and P. Bouquet, "Populating the Smart Musical Instruments Ontology With Data," in *Proceedings of the 27th Conference of Open Innovations Association (FRUCT)*, pp. 260–267 (Trento, Italy) (2020 Sep.).

[25] E. Benetos, S. Dixon, D. Giannoulis, H. Kirchhoff, and A. Klapuri, "Automatic Music Transcription: Challenges and Future Directions," *J. Intell. Inf. Syst.*, vol. 41, no. 3, pp. 407–434 (2013 Jul.).

[26] G. Ghinea, C. Timmerer, W. Lin, and S. R. Gulliver, "Mulsemedia: State of the Art, Perspectives, and Challenges," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 11, no. 1, paper 17 (2014 Oct.).

[27] L. Turchet, T. West, and M. M. Wanderley, "Touching the Audience: Musical Haptic Wearables for Augmented and Participatory Live Music Performances," *Pers. Ubiquit. Comput.*, vol. 25, no. 4, pp. 749–769 (2020 Mar.). https://doi.org/10.1007/s00779-020-01395-2.

[28] 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*, pp. 562–569 (Marseille, France) (2013 Oct.).

## THE AUTHORS

Luca Turchet

Dmitry Golyshev

Luca Turchet is an Assistant Professor at the Department of Information Engineering and Computer Science of the University of Trento. He received master's degrees (summa cum laude) in computer science from the University of Verona, in classical guitar and composition from the Music Conservatory of Verona, and in electronic music from the Royal College of Music of Stockholm. He received a Ph.D. in Media Technology from Aalborg University Copenhagen. His scientific, artistic, and entrepreneurial research has been supported by numerous grants from different funding agencies including the European Commission, European Institute of Innovation and Technology, European Space Agency, Italian Minister of Foreign Affairs, and Danish Council for Independent Research. He is co-founder and Head of Sound and Interaction Design at Elk. His main research interests are in music technology, Internet of Things, human-computer interaction, and multimodal perception.

•

Dmitry Golyshev holds a bachelor's degree from the Department of Information Engineering and Computer Science of the University of Trento. His research interests include music technology and standards.