PUREMX: AUTOMATIC TRANSCRIPTION OF MIDI LIVE MUSIC PERFORMANCES INTO XML FORMAT

Stefano Baldan, Luca A. Ludovico, Davide A. Mauro

Laboratorio di Informatica Musicale (LIM)
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
Via Comelico 39/41, I-20135 Milan, Italy
{ludovico, mauro}@dico.unimi.it

ABSTRACT

This paper addresses the problem of the real-time automatic transcription of a live music performance into a symbolic format based on XML.

The source data are given by any music instrument or other device able to communicate with Pure Data by MIDI. Pure Data is a free, multi-platform, real-time programming environment for graphical, audio, and video processing. During a performance, music events are parsed and their parameters are evaluated thanks to rhythm and pitch detection algorithms. The final step is the creation of a well-formed XML document, validated against the new international standard known as IEEE 1599.

This work will shortly describe both the software environment and the XML format, but the main analysis will involve the real-time recognition of music events.

Finally, a case study will be presented: PureMX, an application able to perform such an automatic transcription.

1. INTRODUCTION

The study of automatic transcription tools is an interesting matter both in research and in commercial applications.

In this paper we will focus on the design and implementation of *ad hoc* automatic transcription algorithms for real-time instrumental performances. This research presents a number of application fields, ranging from the encoding of unique improvisations in symbolic form to speeding up the score writing process, like a musical dictation.

There are some problems to face even at this early stage. First, which kind of music devices should be supported? Which music features should be extracted and translated? Where and when the required computation should be carried out? Finally, which kind of encoding should represent the results of the process? Of course each question can have multiple answers, but our purpose here is just demonstrating that an efficient and effective solution can be implemented. Then our results can be applied with little or no effort to more general cases.

In short, the process described in this paper starts from a live performance, where MIDI-capable music instruments and devices are used. The resulting data stream is parsed by a real-time environment provided by Pure Data. Through this application, a number of algorithms to achieve an automatic transcription are implemented. The final format to represent music events is the new standard known as IEEE 1599-2008. Our choices will be justified in the next sections, and a brief description of the applications and adopted standards will be given.

2. A SHORT OVERVIEW OF IEEE 1599

Even if the encoding in XML format represents only the final step of the process, it is important to describe this aspect immediately as all the algorithms will be affected by this choice.

The music code we adopt, namely IEEE 1599, is not a mere container for symbolic descriptions of music events such as notes, chords, rests, etc. Thanks to its multi-layer structure, illustrated in detail in [1], IEEE 1599 allows to describe many different aspects of music within a unique document. In particular, contents are placed within 6 layers:

- *General* - music-related metadata, i.e. catalogue information about the piece;
- *Logic* - the logical description of score symbols (see below);
- *Structural* - identification of music objects and their mutual relationships;
- *Notational* - graphical representations of the score;
- *Performance* - computer-based descriptions and executions of music encoded in performance languages;
- *Audio* - digital or digitized recordings of the piece.

The *Logic* layer has a central role in an IEEE 1599 document. In detail, it contains i) the main time-space construct aimed at the localization and synchronization of music events, known as *Spine* sub-layer; ii) the symbolic description of the score in terms of pitches, durations, etc., known as *Logically Organized Symbols* (*LOS*) sub-layer; and iii) information about a generic graphical implementation of symbolic contents.

The *Logic* layer is the only level directly involved in the process of live performance transcription, since it contains the music symbols written in Common Western Notation (CWN). Specifically, music events have to be listed, identified and sorted in a common data structure called *Spine*. Spine translates the typically 2-dimensional layout of a score in a 1-dimensional sorted list of music events, uniquely identified by an ID. Each symbol of spine presents a space and time distance from the previous one, expressed in relative way. In this work, only temporization of events is involved and not their placement on a graphical score; as a consequence, only time-related aspects of *Spine* sub-layer will be discussed.

After providing a list of IDs in *Spine*, music events can be defined in standard notation within the *LOS* sub-layer. Here pitches are described by note names and octaves, and rythmical values are expressed in fractional form. For example, a 𝅗𝅥 corresponds to the XML line:

```
<duration num="1" den="2" />
```

and a 𝅗𝅥· to the XML line

`<duration num="3" den="8" />`.
As regards durations, please note that a reduction to lowest terms of the fraction is not required by IEEE 1599 specifications, even if desirable. Similarly, dotted notations is supported in IEEE 1599 in order to try to obtain 1 as numerator, like in the latter example where the duration ♩· has been encoded as ♪⌒♪⌒♪; alternatively, the following XML lines could be employed:
`<duration num="1" den="4" />`
`<augmentation_dots number="1" />`.

The *Logic* layer - which defines music events from a logical point of view - takes a key role for all the other layers, as they refer to spine identifiers in order to bind heterogeneous descriptions to the same music events. In order to obtain a valid XML file, only spine is strictly required, so that even scores not belonging to CWN are supported by IEEE 1599.

In our context, the first advantage coming from IEEE 1599 consists in the possibility to encode contextually additional information: provided that a live performance can be correctly transcribed, within a unique IEEE 1599 document not only the logic score (notes, rests, etc.), but also the corresponding computer-based performance layer, the resulting digital audio, and even related structural information can be encoded. This aspect highlights the heterogeneity of media types and different kinds of description supported by IEEE 1599. Its multi-layer structure allows to organize such a variety as a broad and comprehensive picture of a unique music piece.

Besides, an IEEE 1599 document can host, for each layer, multiple descriptions of the same piece. For example, the file containing the "logic" score of a piece - namely a sequence of music symbols flowing like in the composer's mind - can present *n* different graphical instances, related to *n* score versions, in the *Notational* layer. Similarly, the *Audio* layer can host *m* sub-sections, corresponding to as many tracks (e.g. historical performances, live unplugged executions, transcriptions, variations, piano reductions, and so on).

As a consequence, IEEE 1599 in our opinion fits very well the purposes of this work. Nevertheless, the algorithms we will describe in the following sections can produce transcriptions in any other music code: binary (e.g. NIFF) as well as plain-text (e.g. DARMS, LilyPond), general markup (e.g. MML) as well as XML (e.g. MusicXML).

3. PURE DATA AS A PLATFORM FOR LIVE PERFORMANCES

In this section we will shortly introduce Pure Data, the open-source counterpart of MAX/MSP system. Both of them were developed by the same author, Miller Puckette; his contribution to the project is presented in [2].

Pure Data is an integrated platform designed for multimedia, and specifically for musical applications. This graphical real-time environment can be successfully used by programmers, live performers, "traditional" musicians, and composers.

As illustrated in Figure 1, both the environments had a long evolution since their author started the development process in the eighties. Some of the key concepts have not changed over time, such as the overall flexibility and modularity of the system. Pure Data functions can be improved by the use of *abstractions*, i.e. sub-patches recalled by the user under other patches, and *externals*, i.e. newly created object programmed in C via the Pure Data framework and its API. Pure Data was written to be multi-platform and portable; versions exist for Win32, IRIX, GNU/Linux, BSD, and Mac OS X. Source code is available too.

The program interface is primarily constituted by two kinds of window: *PureData* and *patch/canvas*. The former gives access to

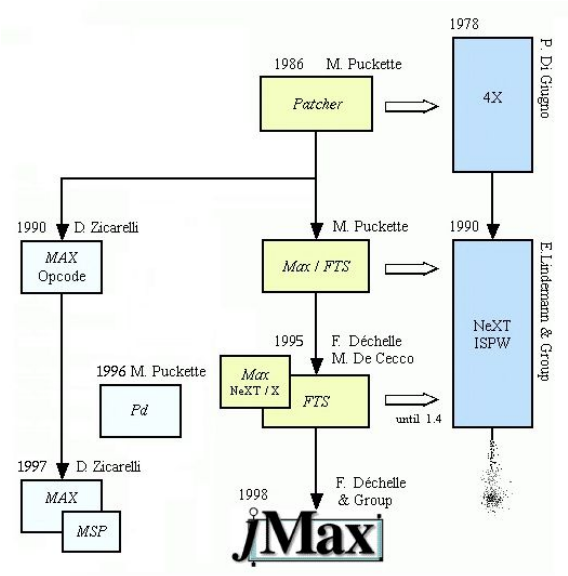

Figure 1: *The evolution of MAX and Pure Data.*

the settings of the program and to the visualization of system messages, allowing the control of the correct workflow. The latter is the place where the user creates and interacts with the application by placing objects and linking them together.

Patches present two different states: *edit mode* and *run mode*. In *edit mode* the user can add *objects*, modify them and link them through *cords*. In *run mode* the patch follows its workflow and the user can interact with it in real-time.

Objects appear like "black boxes" that accept input through their *inlets* or as *arguments* (placed near their name) and return output data through their *outlets*. Programs are built disposing these entities on a canvas (the *patch*) and creating a data flow by linking them together through *cords*. Data are typed; as a consequence not all the possible links are available.

Choosing the linking order has influences on the scheduler priority. Unlike MAX, where the rule is right-to-left execution of links, Pure Data is ruled by the creation time of such links. Even if some patches suggest a certain degree of parallelism, execution is always serialized. This feature can be viewed as a limit but also as a way to simplify priority criteria and execution flows.

Some objects are followed by a "∼" character in their name. This symbol is used to indicate that they are *signal objects*, which means that they can handle audio and video streams.

In the latest versions *interface objects* exist, too. These objects allow the user to control some parameters of the patch during its execution without the annoyance of setting them by typing. As regards their graphical representation, they can have various forms such as buttons, sliders, scrollbars, menus, etc.

An application of the mentioned concepts will be shown in Section 6, where the inferface to achieve real-time interaction and automatic trascription will be descrived.

Before describing the transcription algorithms, let us justify the adoption of MIDI format for source data. Most of the peripherals that can be attached in a live performance environment are usually MIDI capable. Keyboards, synthesizers, MIDI-equipped guitars, etc., can be used by performers to interact with the system. Pure Data can handle MIDI format though its primitives, thus allowing a simple but effective implementation of our work. However, the algorithms introduced in the next section make use of basic information that is available in a large number of formats.

It would be virtually possible to adopt any other input format for the transcription. For example, Pure Data has primitives for OSC (OpenSound Control), thus the support for that format could be easily implemented.

Please note that, even if MIDI has a lower expressivity than IEEE 1599, it is widely used both for score encoding and for numeric communication among sound devices. Thanks to its extensive employment in standard live performance environments, it has been adopted as the base for music events transcription into IEEE 1599 format.

4. FROM MIDI TO IEEE 1599

The simplicity, extensibility and power of Pure Data make it an ideal choice to develop musical applications based on the IEEE 1599 format, thus improving - as a side effect - the diffusion of this new standard. In order to demonstrate the effectiveness of our approach, we have developed PureMX, a library of Pure Data externals designed to convert a MIDI stream (played live by either performers or sequencers, or a combination of both) into a well-formed and valid IEEE 1599 document. By now, the program focuses just on the construction of the *Logic* layer of IEEE 1599, which mainly contains a sorted list of music events (*Spine* sub-layer) and their symbolic representations (*LOS* sub-layer). See Section 2 for further details.

The PureMX library is written in ANSI C, making exclusive use of the standard libraries and the Pure Data API, in order to be highly portable on a wide variety of platforms. It is also extremely modular, taking full advantage of the Pure Data object paradigm and simplifying the integration of new features not yet implemented in the library itself.

Once loaded in the system, PureMX objects can be used inside a Pure Data patch, in combination with the native primitives of the platform, other libraries of externals or Pure Data abstractions (sub-patches).

One of the most challenging aspects encountered in the developing process of PureMX lies in the conceptual and structural difference between the formats involved in the conversion. First of all, while MIDI is just a sequence of chronologically ordered events, IEEE 1599 represents musical information in a hierarchical and multilayered fashion. So, it is useful to organize the MIDI input stream inside a data structure which mirrors the nature of the IEEE 1599 format. In second instance MIDI, as a performance format, stands at a lower level of abstraction than the *Logic* layer of IEEE 1599. For instance, fundamental entities of symbolic score description in IEEE 1599 (such as clef and tonality), that are very rich in semantic content, are not explicitly present in the input stream. In fact, MIDI was designed to convey semantically poorer concepts such as the mechanical actions made by a performer on a musical instrument.

Going down the hierarchy of musical information layers, we can consider the lower levels a "practical instance" of the abstract concepts contained in the higher ones, so information is just translated into a new form but not completely lost. In our case we can consider the events of a performance format as a practical realization of the concepts which should be written in the IEEE 1599 *Logic* layer, so most of the information we need is "hidden" but still present in MIDI. It is possible to apply musical information retrieval algorithms on the input stream and obtain those missing elements, as they are implicitly contained in the relations among events and in the general context of the whole stream. For complete reference on MIDI see [3].

5. ALGORITHMS FOR PITCH AND TEMPO EXTRACTION

5.1. Event Segmentation

In MIDI - like in most performance formats - music events are represented by the succession of two message types: a *noteon* which activates a note and a *noteoff* which deactivates it. Please note that music events can be interleaved, e.g. a *noteon* could follow another *noteon* message before the *noteoff* of the former. Both message types contain a *pitch* parameter that identifies which note has been activated/deactivated, and a *velocity* parameter that indicates the intensity of the action on the instrument. The duration of a note can be easily calculated by counting the time between a *noteon* and a *noteoff* message sharing the same pitch. In the same way, the duration of a rest can be calculated by counting the time between a *noteoff* and the next *noteon* (in this case *pitch* does not matter).

In MIDI, durations are calculated in Midi Time Clock (MTC) units. Most sequencers use 24 MTC per quarter, and their absolute duration depends on the Beat Per Minute (BPM) value of the sequence. On the other side, in the *Spine* sub-layer of IEEE 1599 durations are calculated and stored in Virtual Time Units (VTUs). The only difference is the following: while MTC granularity is fixed, VTU granularity can change for each IEEE 1599 document. For example, if the shortest note of a given music piece is the quarter note, then in IEEE 1599 we can associate a single VTU to this rhythmic value.

In order to achieve a conversion from VTU-based temporal representation to Common Western Notation (CWN), it is necessary to identify and segment music events that are not representable by a single CWN symbol. A rhythmic value whose numerator is not power of 2 should be encoded by using two or more symbols tied together. For instance, the first note in Figure 2 presents a duration that can be expressed as $^7/_8$. The segmentation

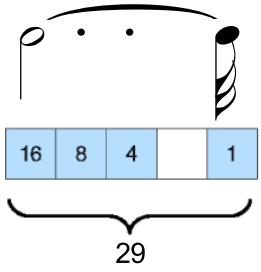

Figure 2: *Segmentation of a VTU duration into parts representable by CWN symbols.*

algorithm implemented in PureMX exploits the similarity between CWN symbols and the binary system, providing an extremely fast and effective way to split complex durations into simple ones. We choose granularity in order to assign the time unit ($2^0 = 1$ VTUs) to the smallest CWN symbol. Going from short to long values, the following symbol will have exactly twice the duration of the first ($2^1 = 2$ VTUs), the third will have twice the duration of the second ($2^2 = 4$ VTUs) and so on. As a consequence, we can represent CWN rhythmic values through variable-length binary strings where the highest order bit is set to `1` and all the others are `0`s.

As a further step, also augmentation dots should be considered. Each augmentation dot increases the duration of a note by half its value, so a dotted value can be represented as a sum of the value itself and of the immediately smaller one. In binary terms, notes dotted n times can be encoded through strings with the highest n+1 bits set to `1` and the others set to `0`.

The segmentation algorithm takes in input an arbitrary binary string which represents a duration in VTUs. If the string encodes a duration longer than $^7/_4$ (whole note with two augmentation dots), events lasting $^7/_4$ are repeatedly written and their duration is subtracted from the original string until the remaining part becomes shorter than this amount. The string obtained in this way is then copied and right bit-shifted until it becomes a sequence of bits all set to `1` (e.g. `1`, `11` or `111`); finally, it undergoes a left bit-shift process by the same amount. In this way, the algorithm finds the largest value representable by a single note symbol, possibly dotted. Such an event is written into the data structure and its duration is subtracted from the original string. The process is repeated until the remaining part of the original string is made of `0`s.

For example, let a note be representable by the fraction $^{13}/_{32}$. This is the case of a dotted quarter ($^3/_8$) tied to a thirty-second note ($^1/_{32}$). If the latter value is the smallest one in the piece, the corresponding binary string would be `1101`. After verifying that $^{13}/_{32} \leqslant {}^7/_4$, the proposed algorithm carries out 2 right bit-shifts, thus producing the string `11` (made of a pure succession of `1`s). Then the process is inverted, and the string becomes `1100`. This bit configuration now corresponds to a single rhythmic value, namely a dotted quarter note. After subtracting the new string from the original one, we obtain the remaining part of the original duration: `1101-1100 = 0001`, i.e. the value to add in order to obtain the whole duration. As a matter of fact, the `0001` string corresponds to a thirty-second note.

Even if metric and accent issues are ignored, the mentioned algorithm is guaranteed to find always a right solution to the segmentation problem.

5.2. Clef Guessing

After facing the problem of complex rhythmical values, we need to define the clef in order to represent pitches by disposing symbols on a staff. In CWN the clef is used to associate a well-defined pitch to each line and space of the staff, thus creating an unambiguous correspondence between the vertical position of a symbol and the name of the note represented.

The problem of finding a clef that fits well a given musical sequence is quite easy to solve, as all the information we need is coded inside the *pitch* parameter of MIDI *noteon* and *noteoff* events. The *pitch* parameter is an integer between 0 and 127, and each number represents a different semitone, like a different key on a keyboard. In MIDI, pitch 60 corresponds to the Middle C, having a frequency of 261.63 Hz approximately. This pitch is usually referred as C4. Consequently, MIDI pitch 61 is assigned to C♯4 (277.18 Hz), MIDI pitch 62 to D4 (293.67 Hz) and so on.

The clef guessing algorithm designed for PureMX is based on the computation of a mean among the various pitches inside a measure, in order to find the "average pitch" of that measure. For the sake of simplicity, the mean is arithmetic: each symbol has the same weight, no matter what its duration is. Please note that this algorithm has to compute results in real time. The whole pitch range is divided into intervals, and a clef is associated to each of them. The clef that fits best the melodic sequence is the one that minimizes the use of additional cuts, as shown in Figure 3.

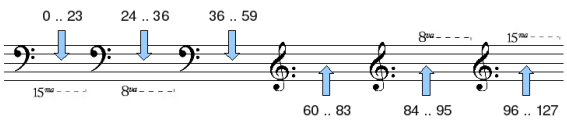

Figure 3: *Average pitch intervals for the PureMX implementation of the clef-guessing algorithm.*

The choice of using just two of the seven available clefs avoids interval overlapping; moreover, in current notation the other five clefs are rarely used, e.g. in particular contexts (such as in vocal music) or for certain musical instruments (such as the alto clef for viola).

The average pitch calculation presents a problematic issue: as the concept of pitch makes no sense for rests, they should not be included in the mean; but in this case empty measures would not have an average pitch value and, consequently, they would not have a clef. The same clef of the previous measure could be assigned, but the problem remains if the first measure is empty too. The adopted solution consists in assigning a pitch value to rests, in particular the same pitch of the previous event, if any, otherwise the pitch of the following one.

The intervals proposed in Figure 3 are the ones used in the PureMX implementation of the algorithm, however there are many other alternatives: for example, creating a specific interval set for vocal music based on *tessitura* instead of average pitch; calculating this parameter by a weighted mean; taking note durations into account; performing this calculation on the whole part/voice instead of measure by measure.

5.3. Key Finding and Pitch Spelling

The *pitch* parameter is also useful in finding the tonal context of a given sequence of notes. This is a fundamental aspect to make a good CWN transcription because in the equal temperament, universally adopted by current western music, each semitone may correspond to 2 or 3 *enharmonically equivalent* pitches. For instance, MIDI pitch 60 can be written in a score as C, B♯ or D♭♭; and pitch 68 only as G♯ or A♭. Knowing the tonal context allows us to choose the correct note name, and therefore the right position on the staff, for each pitch value in the sequence.

In the least 20 years, many people have proposed studies and methods to solve the key finding and/or the pitch spelling problem, reaching good results. The common element shared by every approach is the analysis of musical events not as isolated entities but as part of a context, which affects their interpretation and at the same time is affected by their presence. Differences lie in the choice of parameters and rules to identify and quantify relations among those events. All key finding and pitch spelling algorithms contain heuristic information, namely prior knowledge about the problem they have to solve, mainly based on the rules of tonal harmony. For this reason, all those approaches do not work well (sometimes they fail at all) when applied to music belonging to different cultural areas or historical periods.

In order to give the PureMX library key finding and pitch spelling capabilities, many different solutions have been examined. With simplicity in mind, we wanted an algorithm reasonably efficient, easy to implement and able to solve both problems at the same time. The Krumhansl-Schmuckler algorithm (and its further improvement by David Temperley) [4] has all these features. It is based on a Bayesian approach: each note gives a certain amount of "points" to each possible tonal centre, and the one which gains the higher score is chosen as the most probable for that sequence of notes.

In the PureMX implementation of the algorithm, twelve possible tonal centres are defined, one for each semitone in an octave. In case of enharmonic equivalence, the tonality with less accidentals is preferred, following the principle of notational economy. Scores to tonal centres are then assigned following two probability distributions, one for major keys and the other for minor keys. Such distributions were experimentally deduced by Krumhansl and Kessler at the beginning of the 80s and then improved by Temperley in 1990. The experiment consisted in asking listeners to rate how

well "probe tones" fitted into various musical contexts (cadences in major and minor) [5]. Results are shown in Figure 4.

The probability distributions provides a measure about the fitness of a particular pitch inside the scale of each tonality. For example, a pitch value of 60 (possibly corresponding to the notes C, B♯ or D♭♭) will give a high score to C major as its tonic, or to F major as its dominant, or to A minor as its median; but it will give a poor evaluation to B major because it is not a natural degree in that scale. Score assignment is also weighted on note durations, so that more importance is given to longer notes. It is worth to underline that small rhythmic values are often used for less relevant notes, such as chromatic passages and embellishments.

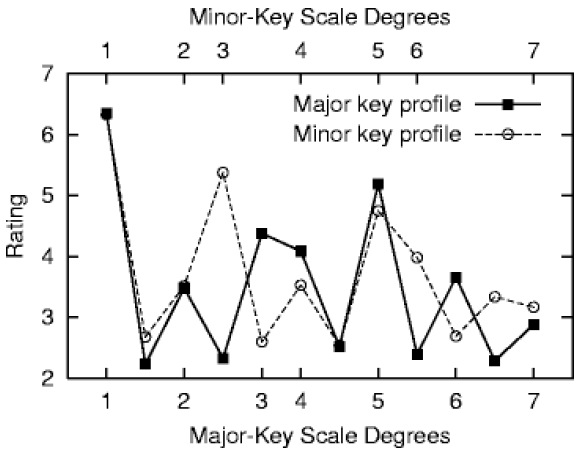

Figure 4: *Probability distributions for score assignment to tonal centres based on pitch values (Krumhansl-Kessler version). Picture taken from [5].*

Once the tonality is determined, pitch spelling for notes on the scale is immediate and unambiguous. For notes outside the scale, note names representing stronger degrees are preferred because they give a better harmonic representation of the key. In particular, major and minor degrees are preferred over diminished and augmented ones.

The original version of the Krumhansl-Schmuckler algorithm uses an overlapping window technique to control its reactivity: small windows and few overlapping notes will produce many tonality changes, while longer and more overlapped windows will produce more uniform results. The PureMX approach is quite different as it uses an exponential mean to obtain the same effects: we define a multiplying factor α in the closed range $[0, 1]$ and a "history" array H made of n elements, with n equal to the number of the considered tonal centres. Let $H(x)$, with $x \in \mathbb{N}_0$, be the status of array H at step x. $H(0)$ is set to 0. Let $C(x)$ be an n-elements array where the score assigned to the current measure are stored. Then we calculate the tonality of a given measure by the equation:

$$H(x) = C(x) \cdot \alpha + H(x-1) \cdot (1-\alpha)$$

The maximum of the n elements of $H(x)$ is used to determine the tonality of the current measure, and the array is stored as the new history. The α factor is a measure of the algorithm reactivity, as greater values give more weight to the current measure evaluation whereas smaller values give more weight to history. As the history is iteratively multiplied by a factor smaller than 1, the contribution of earlier measures to the history becomes less important as time passes, until it becomes irrelevant for very distant measures.

Even if this version of the algorithm already provides satisfactory results for well-defined tonal contexts, further improvements could be applied. For example, it would be possible to implement a pitch spelling algorithm (the one proposed by [6] is simple yet efficient) and use its results to help the key finding algorithm, or could be employed other criteria (voice leading, accents pattern) in addition to the fitness functions in order to obtain a more accurate tonality guess. Anyway, this would add complexity to the algorithm, in contrast with our main goals.

5.4. From Absolute To Relative Time Representation

We have already described the concept of VTU in Subsection 5.1, and we have defined it as the temporal representation of events in an IEEE 1599 document. VTU values are stored in the *Spine* (see Section 2) and describe the chronological order of the events in a musical piece. We can see VTUs as ticks of a clock, where the absolute duration (in seconds) of a single tick depends on the chosen granularity and on the BPM value of the piece.These values are intentionally unexpressed at *Logic* level as the virtual temporization of score symbols can correspond to different instances at *Performance* or *Audio* layer. Anyway, a different concept of time is also present in the *LOS* sub-layer. Each note and rest is described as a CWN symbol with its relative duration, so we need to transform VTU durations in fractions of measure in order to obtain the logic description of music events.

Let us recall that the conversion from VTU-based timings to CWN fractional durations has to be achieved in real time, so a fast but effective algorithm has to be designed and implemented. Nevertheless, extemporary improvisation is often made of irregular rhythms, including nested tuplets, so the problem of rhythm transcription is not trivial and cannot be solved by using mere quantization.

In the following, the approach employed in PureMX is described. The inputs of the algorithm are:

1. the meter of the piece in fractional form $\frac{s}{t}$ (e.g. $\frac{4}{4}$, $\frac{6}{8}$, etc.). Please note that s is the number of subdivisions in a measure, whereas $1/\mathrm{t}$ is the rhytmic value corresponding to a subdivision;
2. VTUs per measure (let it be v), or alternatively VTUs per subdivision (let it be w); v and w are related by the formula $v = s \cdot w$;
3. the duration of the music event to parse, expressed in VTUs and referred in the following as x. This amount is available as soon as a *noteoff* event follows the corresponding *noteon* in case of notes, or a *noteon* message follows a remote *noteoff* in case of rests.

The process proposed here will return the CWN symbol to write in the *LOS* sub-layer, even when located inside a tuplet. The duration will be represented through a numerator n and a denominator d, where n is not necessarily equal to 1 whereas d has to be a power of 2. In short, the following algorithm computes n and d starting from s, t, w (or v) and x.

1. Let g be the greatest common divisor (gcd) between x and $(w \cdot t)$;
2. Calculate $a = x/g$ and $b = (w \cdot t)/g$. The fraction a/b represents the reduction to lowest terms of the original fraction $x/(w \cdot t)$;
3. Evaluate the obtained denominator b. If $b = 2^n$, with $n \in \mathbb{N}_0$, namely it is a power of 2, then proceed to branch (a). Otherwise, the value to be parsed belongs to a tuplet. In this case, let d be the floor rounding of b to the closer power of 2, which will allow to write an existing rhythmic value in CWN. Jump to branch (b);

(a) Set $n = a$ and $d = b$. Now the event duration can be encoded in the *LOS* sub-layer. If a more compact notation is required, the previously described segmentation algorithm can be employed in order to express the fraction as a sum of rhythmic values, possibily dotted or double-dotted. Finally, jump to Step 4;

(b) In order to dimension the tuplet, namely to fix the number i of values the measure should be divided into, calculate $i = s \cdot w/x$. In natural language, this means: "Divide the measure in i equal parts and represent the current note as n/d under the resulting tuplet". In IEEE 1599, the tuplet encoding is in the form "Put i/d in the place of s/t" (e.g. "Put $6/8$, i.e. 6 ♪, in the place of $2/4$, i.e. 2 ♩"), and all the required values have been calculated. According to standard notation rules, necessarily $i \in \mathbb{N}$, but in general this is not guaranteed by the algorithm (refer to the last example of this subsection). In this case, a segmentation algorithm has to be used in order to split the original x into the sum $x_1 + x_2 + ... + x_n$, where each element makes the corresponding i an integer. It is possible to demonstrate that this operation can always be performed in a finite number of steps;

4. Write the n and d values in the corresponding attributes of the IEEE 1599 element `<duration>`. If the event is dotted, or it belongs to a tuplet, compile also those parts of the document.

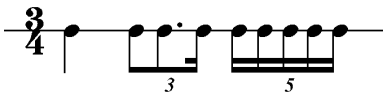

Figure 5: *A measure containing both standard durations and tuplets.*

Now we will provide a number of examples to clarify the applicability of the algorithm. Let us consider Figure 5, where time signature is $\frac{3}{4}$, thus $s = 3$ and $t = 4$. Let 30 be the number of VTUs per quarter, i.e. $w = 30$ and $v = 30 \cdot 3 = 90$. Finally, let us apply the algorithm when $x = 30$, which intuitively corresponds to the first note in Figure 5. From Step 1, g is the gcd between $x = 30$ and $w \cdot t = 30 \cdot 4 = 120$, so $g = 30$. Step 2 provides the reduction to lowest terms by using g, so the numerator $a = 1$ and the denominator $b = 4$. The evaluation process at Step 3 confirms that $a = 1$ and $b = 2^n$, with $n = 2$. As a consequence, the obtained results corresponding to a ♩ are ready to be written in the XML description of the event.

When $x = 10$, g is the gcd between $x = 10$ and $w \cdot t = 30 \cdot 4 = 120$, so $g = 10$. Step 2 states that $a = 1$ and $b = 12$. Step 3 determines that b is not a power of 2, so branch (b) is entered. Through the algorithm, $b = 12$ is rounded to the value 8, and this value is assigned to d. This means that the symbolic value to be written under tuplet is ♪. Now the tuplet has to be determined. From $i = s \cdot w/x = 3 \cdot 30/10$ follows $i = 9$. Finally, the algorithm states how to compute the tuplet: "Put i/d in the place of s/t", namely "Put $9/8$, i.e. 9 ♪, in the place of $3/4$, i.e. 3 ♩". Even if already correct, the result can be improved by dividing the two numerators by their gcd, thus obtaining "Put $3/8$, i.e. 3 ♪, in the place of $1/4$, i.e. 1 ♩", which corresponds to the graphical representation of the second note in Figure 5.

When $x = 15$, $g = 15$, $a = 1$ and $b = 120/15 = 8$. Since the obtained b is a power of 2, branch (a) is entered. As a consequence, the third value in Figure 5 is recognized as ♪, even if its graphical representation was a dotted eighth note inside a triplet. From a mathematical point of view, this is correct: in fact the latter representation implies taking 1.5 parts of the subdivision of a quarter by 3, i.e. $3/2 \cdot 1/3 \cdot 1/4 = 1/8$, corresponding to an eighth note.

When $x = 6$, intuitively we are dividing the VTU duration of the subdivision by 5. This is the case of the last 5 notes in Figure 5. By applying the algorithm, $g = 6$, $a = 1$ and $b = 120/6 = 20$. Since the obtained b is not a power of 2, branch (b) is entered. The rhythmic value to use in the tuplet is $1/16$, as 16 is the floor rounding of 20 to the nearest power of 2. From $i = s \cdot w/x = 3 \cdot 30/6$ follows $i = 15$. Finally, the algorithm says: "Put i/d in the place of s/t", namely "Put $15/16$ in the place of $3/4$", or alternatively "Put $5/16$ in the place of $1/4$".

Finally let us explain the process when applied to a more complex case. Let the meter be $\frac{7}{8}$, thus $s = 7$ and $t = 8$. Besides, let 9 be the number of VTUs per ♪, i.e. $w = 9$ and $v = 9 \cdot 7 = 63$. Please note that no problem is due to the complex time signature: e.g., when $x = 36$ a 𝅗𝅥 symbol is recognized; and when $x = 3$ the algorithm recognizes a sixteenth note in a tuplet made of 21 values of the same kind (in this case, the tuplet puts 21/16 in the place of 7/8, namely a ♪· in the place of a ♪). Rather, a problem arises when $x = 4$. In fact, $g = 4$, $a = 1$ and $b = 72/4 = 18$. Since the obtained b is not a power of 2, once again branch (b) is entered and the value to use in the tuplet results to be $1/16$. From $i = s \cdot w/x = 7 \cdot 9/4$ follows that $i \notin \mathbb{N}$, and the original x has to be split into a sum of integer addends. For example, a simple way to solve the problem is considering $x = x_1 + x_2 = 3 + 1$, which corresponds to representing a unique music event as two tied symbols of different duration. Now the algorithm is able to work on such values.

6. PUREMX INTERFACE

As a library of Pure Data externals, PureMX shares the user interface of its host program. Figure 5 shows an example patch in which all of the PureMX externals are used together with some native Pure Data primitives.

The first external of the set is *mxsheet*, which records, segments and stores events of the input stream inside an organized and hierarchical data structure. Its leftmost inlet accepts some reserved messages that control the recording process (start, stop) and the pitch number of incoming events. The next two inlets accept velocity values and channel numbers, respectively. The other inlets are used to indicate BPM value, time signature (numerator and denominator), VTU per quarter and whether the recording should start automatically as soon as the first event is received or not. By now, these values are set manually through numeric inputs, but further development could involve the creation of beat tracking or metric structure recognition objects.

The *notein* object attached to the first three inlets is a native Pure Data primitive, and it is used to translate MIDI events into Pure Data messages. This means that, even if PureMX has been conceived with MIDI in mind, it is theoretically possible to use the library with any other performance format that has the same data organization of MIDI. For example, attaching an OSC translator instead of the *notein* object, we could make use of OSC messages instead of MIDI messages.

In terms of outlets, the first two simply send out *bangs* (sort of "wake up" messages) in time with the metronome, while the third sends out a custom message (*mxfeed*) once the recording stops.

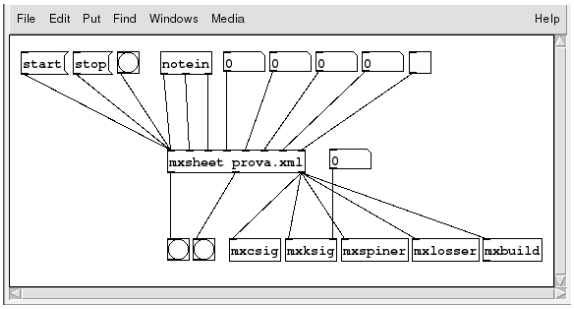

Figure 6: *Example patch showing PureMX externals, together with some Pure Data primitives.*

This last message contains, among other things, the pointer to the first memory cell of the recorded data structure, and it is used by other PureMX externals to read the recorded information or write new elements.

The other objects of the library are *mxcsig*, which implements the clef guessing feature; *mxksig*, which implements the key finding and pitch spelling features; and finally *mxspiner*, *mxlosser* and *mxbuild*, which respectively write the *Spine*, the *LOS* and the whole IEEE 1599 document to text files. It is not mandatory to use all of these objects: there are cases in which, for example, the IEEE 1599 document is used just for synchronization so the *LOS* element is not needed, or it makes no sense to guess tonality as the musical piece recorded does not belong to the tonal repertoire.

7. CONCLUSIONS

In this paper we have presented a working application that can fulfil the needs of musicians who want a transcription of their live performance. This can be useful for recording of live electronics, for improvisation, that are unique and always varying, for music archives, for analysis of the recorded material and so on. We have faced the problem adopting IEEE 1599 and Pure Data, namely open and extensible environments that allow the tailoring of the application to the user needs. In the paper we have also proposed the use of some algorithms, both designed *ad hoc* and adapted from literature, that can solve the various problems encountered in the definition of the problem domain and in its practical resolution.

8. REFERENCES

[1] L.A. Ludovico, "Key Concepts of the IEEE 1599 Standard", *Proceedings of the IEEE CS Conference The Use of Symbols To Represent Music And Multimedia Objects*, IEEE CS, Lugano, Switzerland, 2008.

[2] Miller S. Puckette, "Pure Data: another integrated computer music environment", *Proceedings of Second Intercollege Computer Music Concerts*, Tachikawa, Japan, pp. 37-41, 1996.

[3] V. Authors. The Complete MIDI 1.0 Detailed Specification. Document version 96.1. The MIDI Manufacturers Association, 1996.

[4] D. Temperley, "What's Key for Key? The Krumhansl-Schmuckler Key-Finding Algorithm Reconsidered", *Music Perception*, vol. 17, no. 1, pp. 65-100, Sep. 1999.

[5] C.L. Krumhansl, *Cognitive Foundations of Musical Pitch*, Oxford University Press, Oxford, UK, 1990.

[6] E. Cambouropoulos, "Pitch Spelling: a Computational Model", *Music Perception*, vol. 20, no. 4, pp. 411-430, Jun. 2003.