Virtual Gamelan Graz: Rules - Grammars - Modeling
Symposium at the Institute of Ethnomusicology, Oct. 27-28, 2006

The Bol Processor project: musicological and technical issues

Bernard Bel

Background

Bol Processor 2 (BP2) is a program for music composition and improvisation with real-time MIDI, MIDI file, and Csound output. It produces music with a set of rules (a compositional grammar) or from text 'scores' typed or captured from a MIDI instrument. These rule sets are very similar to the formal grammars (context-free, context-sensitive, etc.) that are used in computer science to define machine-readable languages. As a compositional tool, Bol Processor has been successful at modeling music of many styles including Western classical music, serial music, contemporary art music including minimalism, Indian classical music, and jazz. More information about the capabilities of BP2 is available at <http://www.lpl.univ-aix.fr/~belbernard/music/>.

BP2 began its life as a shareware program developed by Bernard Bel with the help of Jim Kippen and Srikumar Karaikudi Subramanian. Recently it was open-sourced by Sourceforge at <http://bolprocessor.sourceforge.net> with the help of Anthony Kozar. BP2 is currently available for both MacOS X and MacOS 7-9. We hope that a community of developers will port it to other platforms and continue to enhance its features.

Readers will find a complete bibliography via the 'Publications' link of my personal web page <http://www.lpl.univ-aix.fr/~belbernard/>.

A brief history of the development in terms of technical constraints

BP2 evolved from the initial bol processor program (BP1) used by Jim Kippen during his field work with tabla musicians in 1982-1985. The programming environment was 6502 assembly language, 1 MHz clock frequency, 64 Kbytes RAM and 143 K floppy discs for data storage… Over the years, these figures grew up exponentially, but in the same time data complexity increased in such a way that the care for time/space complexity remained a priority. In recent years, notably, work on recursive grammars and polymetric structures made it compulsory to design a **quantization** technique.

Rule-based models: are they suitable for music?

The initial requirement for the program was an advanced word-processing package in which (1) keyboard keys would be mapped to strings rather than characters, and (2) search-replace operations could be automated. The latter was solved by implementing **formal grammars** (the Chomsky model) and an **inference engine** capable of applying ('firing') rules on a string of symbols (the **work string**).

It quickly became obvious that the model should be extended to a comprehensible representation of **patterns**, thereby meaning repeated or 'pseudo-repeated' phrases. Pseudo-repetitions are represented with the help of **homomorphic mappings** of the

alphabet.

The model of pattern grammars that I had implemented in BP1 (namely, **bol-processor grammars**) can be used for both the **production** of ('monodic', unilinear) musical pieces, and the **parsing** of a given piece to assess its acceptability by the grammar (**membership test**). Due to technical constraints (time/space complexity), the parsing algorithm needs to be deterministic. To this effect I designed an algorithm for **context-sensitive rightmost derivation**.

When repetitions or pseudo-repetitions are defined in the grammar, it is necessary to produce and store a set of **templates**, one of which is supposed to match the musical exemple if it is accepted the grammar.

BP1 was used for modelling musical improvisation based on the theme-and-variation approach (named *qa'ida* in Urdu/Persian). The machine was used to replicate a musician's behaviour by producing a few variations on a given theme encapsulated in a **bol-processor grammar**. It became obvious that the randomness of the process should be channelized in such a way that the grammar will be more likely to produce the most recurrent patterns. Grammar rules were therefore weighted.

Weights can be set to change (decreasing or increasing) each time a rule is fired, a simple way of avoiding redundancy: if a rule starts with weight=100 set to decrease by 20 each time it is fired, then the same rule will not be used more than 5 times. A **learning algorithm** was implemented to determine **optimum weights** in a grammar on the basis of an example set supplied by the expert musician.

Grammar procedures

The Bol Processor approach is used in three different environments:

1) Trying to construct a grammar representing a set of variations that a musician has in mind when **improvising** on a theme. The output may be text only if it is read or played by the musicologist.

2) A given grammar is set to **produce and play variations in real time**. Some parameters (tempo, flags…) may be modified from a MIDI keyboard during the performance. The program may also wait for instructions sent on a MIDI channel by a human musician, a sequencer — or another BP2 — to take its turn in playing a piece (**group improvisation**).

3) A grammar is set to **produce a unique musical piece**.

In all these environments grammars are divided in **subgrammars**. A subgrammar is used (in production or parsing) until it has no more candidate rule.

Different procedures are used for the selection and firing of rules in a subgrammar, as well as the search for an occurrence of the left argument of a rule in the work string. Selection/search is *random* by default, but it may also be forced to *top down, bottom up*, *left to right*, *right to left*. A rule may also contain a *_goto* instruction telling the inference engine (in production mode) to jump to a given rule in a given subgrammar.

Metavariables

BP1 and BP2 use 'wild cards' in rewrite rules, in a way similar to *grep, Perl* and other string-processing languages.

Flags

In production mode, it is possible to modify the validity of a rule thanks to the presence/absence of a **flag**. A flag exists when it has been produced by a rule with an initial (positive) value decided by the rule. A rule may increment/decrement some specified flags each time it is fired. Once the value of a flag reaches zero, it becomes inactive. This feature proved very useful for grammars in which rules should take into account the 'history' of the making of the musical piece. It will be demonstrated in a set of variations on a Carnatic musical theme.

Infering grammars from examples

This part will be quickly explained though not demonstrated for technical reasons: it has been implemented in an old (1990) version of Prolog 2. The system was called QAVAID (*question-answer validated analytical inference device*, Bel & Kippen 1989). It is worth discussing how a machine-learning approach highlights the need for **eliciting presupposed knowledge** — in this case, implicit segmentation rules.

Polymetric structures

All (symbolic) time-related computing in BP2 is done with integer ratios. This avoids rounding errors that would be dramatic for the symbolic (rather than graphical/numeric) representation of time structures. The timing of musical objects is measured in **beats** and **subdivisions of beats**. Durations expressed in this manner are called **symbolic durations** (akin to *crochets*, *quavers* etc. in Western music) in contrast with the **physical durations** (in milliseconds) of sound-objects that will depend on their metrical properties, the time structure and the global speed of performance (see *infra*, the time-setting algorithm).

A **polymetric structure** is an arbitrary set of sound-objects performed over a given time interval — its symbolic duration.

In the BP2 text format, a polymetric structure is represented between curled brackets as a **polymetric expression**. This expression may contain several lines, akin to the lines of a musical score, that will make it 'polyphonic'. Each line is a sequence of sound-objects. Lines are separated with commas to provide a unilinear representation. Furthermore, this description is recursive: any sound-object might in turn be replaced with another polymetric expression. At the highest level, the entire musical piece is treated as a unique polymetric expression.

On each line, the **terms** (sound-objects or polymetric expressions) may be chunked in groups whose separator is a **period**, e.g.

ab • cde • fghi • j

in which *a, b, c…* are the labels of sound-objects. Each period is the marker of a beat (according to the tempo locally assigned to that particular line).

The **polymetric expansion algorithm** takes care of calculating the durations of all sound-objects (plus unspecified silences) in the 'most evident' manner, which means it takes for granted that *beats should have equal durations*. The above example yields:

a_ _ _ _ _ • b_ _ _ _ _ • c_ _ _ d_ • _ _ e_ _ _ • f_ _ g_ _ • h_ _ i_ _ • j_ _ _ _ _ • _ _ _ _ _ _

Imagine a metronom tick on each period. To deal with polyrhythmicity, beats have been subdivided to 12 parts, i.e. the lowest common multiple of 2, 3, 4. Underscores indicate prolongations of the preceding sound-object; for instance, "e___" is sound-object *e* performed over 4 subdivisions of a beat.

In the compact (internal) representation, this is equivalent to:

a b *2/3 c d e *1/2 f g h i *2/1 j

in which **2/3*, **1/2* and **2/1* are **explicit tempo markers**.

If the musician is not satisfied with this interpretation, s/he may tag more details into the structure.

The lines of a polyphonic structure are treated in a similar way. (In fact the algorithm is the same one.) The algorithm takes for granted that *all lines should have the same duration*. In the end (at the highest level) the algorithm returns the **expanded polymetric expression**, one in which tempo and durations are marked explicitly. For instance, the expression

{ab.cde, fg.hi.j}

containing 2 lines separated by a comma, yields the following expanded expression

{a_ _ b_ _ c_ d_ e_ , f_ g_ h_ i_ j_ _ _}

thereby meaning

```
a _ _ b _ _ c _ d _ e _
f _ g _ h _ i _ j _ _ _
```

which is internally represented as:

{a b *2/3 c d e , *2/3 f g h i *4/3 j }

Flexible tempo

The algorithm for the expansion of polymetric structures has been extended to allow for specifications of **relative changes of tempo** within each polyrythmic phrase without modifying the tempo of any phrase superimposed to the modified one. Changes may also be made progressive throughout the phrase (*accelerando* or *rallentando*). This makes it possible to push the complexity of a polyphonic/polyrhythmic piece to such a degree that the superimposed musical lines sound 'time-independent'. (This is sometimes called 'polychronicity'.)

Quantization

When using flexible tempo, very complex time ratios are generated and the 'grain' of the structure may become smaller than the least perceptible time interval. **Quantization** is an algorithm run after the polymetric expansion, which simplifies the expanded structure in compliance with the speed of performance and the specified least perceptible interval (typically 10 ms).

Time structure

BP2 uses either **streaked** or **smooth time** as defined by Boulez. Streaked time yields regular beats — which does not prevent the musical piece from having local changes of tempo, see 'flexible tempo' supra. Smooth time is the absence of a beat pattern: the timing of the structure is entirely determined by the duration of sound-objects on the first line of its 'score' (polymetric expression).

Sound-objects

Initially, the Bol Processor was designed for the onomatopeic notation of strokes on a percussion instrument (*bol* in Hindi/Urdu). Each stroke may be called a **sound-object** if we define it as a sequence of 'elementary musical gestures'. It was logical to extend the concept of sound-object to sequences of MIDI messages or Csound score lines. The most basic sound-object in MIDI is the NoteOn/NoteOff pair on a single pitch/channel, a predefined object that is called a **simple note** in BP2. Simple notes are notated with predefined symbols reproducing Italian/French, English or Hindustani note names.

The position of a sound-object is defined by its **pivot** which may be the beginning (by default), the end, the middle, the first NoteOn, last NoteOff, middle of MIDI stream, or any distance from the beginning defined in milliseconds or in percentage of its duration. This flexibility is required when sound-objects are performed on a sampler.

Sound-objects protoypes (other than simple notes) may be assigned **metrical** and **topological properties**. Metrical properties determine the limits within which the sound-object may be stretched or contracted with respect to the performance speed. Among the topological properties are the possibilities of truncating part of the object or relocating it if some constraint is set up (by another object) not to overlap or get too close.

A **time-setting algorithm** takes care of propagating constraints and finding a set of acceptable solutions before playing a musical piece. In real-time improvisation, some constraints may be automatically released if a limit computation time has been reached.

Sound-objects may be assigned **pre-roll** and **post-roll** values compensating for mismatches between the beginning/end of their instructions (MIDI messages or Csound score lines) and the actual perception of a beginning and an end.

(Part of) a sound-object may be defined as **cyclical** is it is meant to be repeated instead of stretched beyond a fixed limit.

I will demonstrate the use of sound-objects for producing a sensation of 'swinging' despite a rigid streaked time structure.

Time-objects

Time-objects are two sets of predefined sound-objects containing no sequence of instructions. The first set is 'silent objects' used as 'peg holes' to determine accurately complex time-ratios in a polymetric structure when smooth time is activated. The second set is time-objects of null duration that handle output/input messages in the MIDI environment, such as waiting for a key press, a NoteOn or a particular value of a controller.

Performance tools

Predefined instructions may be inserted in a musical piece to control a few parameters during the performance. For instance, *_pichbend()* will set up the value of *pitch bend* on the current MIDI channel selected by the *_chan()* tool. *Pressure, modulation, volume, panoramic* and *velocity* are similarly controlled by **performance tools**. These parameters may be set up to change abruptedly, stepwise or continuously (by interpolation).

In Csound, the five continuous MIDI controllers can be mapped to arguments of any instrument, using an adjustable conversion graph and a specific generator (by default GEN07). Two arguments can be assigned to the velocities of a simple note — one for the attack and another one for the release. The dilation ratio of a sound-object may be passed as an argument to modify the algorithm accordingly. In addition, a Csound instrument may be assigned up to six additional parameters that do not have an equivalent in MIDI. Each additional parameter has its combination rules (additive or mutiplicative) and an optional argument pointing at a function table if it is meant to vary continuously.

Another set of performance controls modify the durations of time-objects (including simple notes) to produce effects of *staccato* and *legato*. These values may also be interpolated throughout a phrase.

Time base

BP2 has a sophisticated **time base** producing up to three superimposed tick patterns. These may be used as a metronom for music performance or imbedded into the performance of a piece generated by BP2.

Serial tools

Following the advice of Harm Visser, I introduced a few tools facilitating the design of musical structures based on the principles of serial music. Since these tools are directly applied to polymetric expressions, their effect is propagated throughout the polymetric structure. For example, the *_retro* tool will reverse the order of sound-objects in all sequences contained in the following structure (and its substructures).

Current serial tools are *_retro* (reverse the order of a sequence), *_rndseq* (randomize a sequence), *_rotate* (cycle-shift the terms of a sequence), *_ordseq* (play sequence in order irrespective of higher-level instructions), *_keymap* (map a pitch interval to another one) and *_keyxpand* (expand pitch intervals using a given ratio around a given center note).

Graphics

BP2 allows the graphic display of a piece with its actual timings. Sound-objects are displayed as rectangles. Scores of simple notes may also be displayed on a piano-roll.

MIDI import/export

MIDI score files may be imported to (MIDI) sound-object prototypes. Conversely, any piece produced by BP2 may be exported as a MIDI file (type 0, 1 or 2). This makes it possible to use grammars to define very sophisticated (MIDI) sound-objects.

Csound export/import

The same import/export features are available for (Csound) sound-object prototypes and BP2 data. However, the process is simplified by the fact that Csound uses plain text files.

MIDI + Csound

MIDI and Csound co-exist in BP2. Sound-objects prototypes may be designed as **dual**, thereby meaning they contain both a Csound score and a MIDI stream. A conversion (MIDI to Csound) is even implemented, using the currently defined Csound instruments — for instance, MIDI channel 3 may be mapped to instrument number 5 — and the mappings of MIDI controlers to the arguments of Csound instruments.

Scripts

Any sequence of user's instructions in BP2 may be automatically recorded as a script. Scripts are stored as text files and use a syntax ressembling that of HyperCard. A script may call a subscript.