Musical Metacreation: Papers from the 2012 AIIDE Workshop
AAAI Technical Report WS-12-16

Computational Music Theory

Georg Boenn
Cardiff School of
Creative & Cultural Industries
University of Glamorgan
Pontypridd, CF37 1DL, UK
gboenn@glam.ac.uk

Martin Brain
Department of Computer Science
University of Oxford
Oxford, OX1 3QD, UK
martin.brain@cs.ox.ac.uk

Marina De Vos and **John ffitch**
Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
{*mdv,jpff*}*@cs.bath.ac.uk*

Abstract

One of the goals of the study of music theory is to develop sets of rules to describe different styles of music. By formalising these rules so that their semantics are machine intelligible, it is possible to use computers to reason about and analyse these rules – *computational music theory*. ANTONis an automatic composition system based on this approach. It formalises the rules of Renaissance Counterpoint using $AnsProlog$ and uses an answer set solver to compose pieces. This paper discusses ANTON, presenting the ideas behind the system and focusing on the challenges of modelling and synthesising rhythm.

Introduction

Music is an important aspect of all cultures, and forms an significant part of many people's lives. Despite this, and much effort by musicologists and others (for example, (Thakar 1990; Huron 2006; Leach and Fitch 1995b; 1995a)), it remains in many ways a mystery.

This paper describes a computational investigation into the basis of music, applying ideas from logic programming and artificial intelligence, with the aim of developing a computational theory of music. This incorporates a logical representation of some musical components so we can apply computational techniques.

Our joint aim is to create a system for autonomous music composition, but the authors have different perspectives and motivations. We include composers, algorithmic composers, computer scientists, logicians, performers and listeners. Underlying aims include learning about music and music theory, demonstrating the simplicity and power of the logic tools, and the fun of building an innovative program. All these attitudes have contributed in some way to the overall project.

We should stress that the interest is in **composition** rather than improvisation, with the ability to consider the whole work without linear time constraints. We are currently only considering styles of music that rely on discrete notes and parts, but fully accept that there are other musical genre to which this does not apply. Following a brief description of the language we use, we describe the development of the basic system and musical style rules. We follow this with a description of the problems associated with rhythm, our current solution and a sketch of a putative improved system. We end with some examples and consideration for future work.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background

Our work is predicated on the over-strong axiom that for a sequence of musical notes to be considered as a piece of music it must satisfy certain requirements, regardless of the style of music one considers. These rules often specify the progression of a melody, both at micro level (the choice of the next note) and at the macro level (the overall structure). Others describe the harmony, which arises from the relationship between the various voices, while others describe rhythm and the interaction between the various parts in a composition. A composer will use her/his skill, knowledge and creativity to provide a euphonious or meaningful composition that meets these requirements. We have restricted our work currently to discrete notes with pitch and duration.

While these rules were developed to guide and support human composers and to teach generations of aspiring musicians and composers, they can, with the appropriate knowledge representation techniques, turned into a computer system that can reason about and apply these rules. Such a system provides an easy and versatile way for automatically composing music. Furthermore, provided that the underlying representation mechanism is sufficiently flexible to allow changes at the level of the individual musical rules themselves, it will give the human composer or musicologist a tool to understand, explore, extend, experiment with the set of musical rules he is working with. In other words, it creates the opportunity to study *computational music theory*.

In this paper we present ANTON v2.0, an extension of ANTON v1.0 reported in (Boenn et al. 2009). ANTON is an automatic composition system based on the set of compositional rules governing tonal Western music, using Answer Set Programming (ASP) (Gelfond and Lifschitz 1991; Baral 2003). ANTON is Knowledge Representation and Reasoning (KR&R) in the truest sense: using the declarative properties of $AnsProlog$, the ASP language used in our approach, one just writes down the rules of music theory and reason about them. The order of the rules does not matter unlike some logic systems (e.g. Prolog).

ANTON v1.0 focussed on melodic and harmonic composition. This gave ANTON the same capabilities as its competitors such as Strasheela (Anders 2007) and Bol (Bel 1998) but with more flexibility, more powerful search algorithms and a more intuitive and smaller code-base. We have now added rhythm, including different durations of notes across parts and chords, to the system. The addition of rhythm to the ANTON v1.0 proved a challenge for both the formalism and the implementations used by answer set programming. So in addition to providing a full description of the system, we also reflect on the use of answer set programming in large applications like this.

ANTON provides an experimental platform for computational music theory, offering tools to discover what aspects of music existing music theory captures and what it does not. ANTON v2.0 shows that all known aspects of tonal Western music can easily be modelled in a computational framework. More generally, it demonstrates that the use of a 'off the shelf' reasoning engine, in this case answer set solvers, can result in a versatile and powerful algorithmic composition system that is fast, intuitive and easy to build and adapt. For answer set programming, the system shows that the paradigm can be used to model complex problems. At the same time it also highlights the current modelling problems and computational bottlenecks.

Elements of ASP

ANTON uses Answer Set Programming (ASP) (Gelfond and Lifschitz 1991), a declarative programming paradigm to represent and reason about musical knowledge. While there are other languages that implement this paradigm (ID-Logic (Denecker and Ternovska 2008), logic of propositional schemata (East and Truszczyński 2001)), ANTON uses $AnsProlog$ (Baral 2003), a logic programming language using the answer set semantics, for its encoding. In this section, we only give a brief overview of the language. The interested reader is referred to (Baral 2003) for a more in-depth coverage.

$AnsProlog$ is a knowledge representation language that allows the programmer to describe a problem and the requirements of the solutions rather than developing the algorithm to find the solutions to the problem. The basic components of the language are atoms, elements that can be assigned a truth value. An atom can be negated using *negation as failure* in order to create the *literal* $\text{not } a$. We say that $\text{not } a$ is true if we cannot find evidence supporting the truth of a. If a is true then $\text{not } a$ is false and vice versa. Atoms and literals are used to create rules of the general form: $a \leftarrow B, \text{not } C.$ where a is an atom, B and C are sets of atoms. Intuitively, this means *if all elements of B are known/true and no element of C is known/true, then a must be known/true.*, We refer to a as the head and $B \cup \text{not } C$ as the body of the rule. Rules with empty body are called *facts*; the head should always be true. A *program* is a finite set of rules.

The semantics of $AnsProlog$ are defined in terms of *answer sets* – assignments of true and false to all atoms in the program that satisfy the rules in a minimal and consistent fashion. A program has zero or more answer sets, each corresponding to a solution, or in the case of ANTON, a different musical piece. The semantics are exploited to produce different musical pieces that do not break rules or provide an explanation why a piece is not valid.

```
%% Time steps are given independently for each part
partTime(P,1..TM) :- partTimeMax(P,TM).

%% Each part can only play one note at a given time
%% This is needed so that partial pieces can be supplied.
:- 2 { chosenNote(P,T,NN) : note(NN)}, rest(P,T), partTime(P,T).

%% If we step, we must pick an amount to step by
1 { stepBy(P,T,SS) : stepSize(SS) : SS < 0 } 1 :- stepDown(P,T).
```

Figure 1: A sample from the ANTON v2.0 progression rules

When used as a knowledge representation and programming language, $AnsProlog$ is enhanced to contain constraints (e.g. $\leftarrow b, \text{not } c$) and choice rules (*e.g.* $\{a, b, c\} \leftarrow b, \text{not } c$). The former are rules with an empty head, stating that an answer set cannot meet the conditions given in the body. The latter is a short hand notation for a conditional, non-deterministic choice; if the conditions in the body are met then a number of atoms in the head must be contained in the answer set. These additions are syntactic sugar and can be removed with linear, modular transformations (see (Baral 2003)). Variables and predicated rules are also used and are handled, at the theoretical level and in most implementations, by instantiation (referred to as *grounding*).

Most ASP systems are composed of two processes: grounding and computing the answer sets of the propositional program with an *answer set solver*. We use GRINGO (Gebser, Schaub, and Thiele 2007) and CLASP (Gebser et al. 2007) for these.

Modelling Simple Music

The core of ANTON consists of a collection of files defining the musical rules for modes, notes, melody, harmony, chords and note progression. Without the rhythm mode enabled, all notes are considered of equal length, evenly distributed and all the notes can be part of a chord.

The simplest rules govern the pitches that can be used[1], and simple definitions of sequence. Some of these rules can be seen in Figure 1, covering the concepts that a part can only play one note at a time, that a note is either a pitch or a rest, and notes last for a duration. There are also rule-sets defining the major and minor scales and various modes. The more interesting of these rules define the difference in rising and falling minor scales; that we can do this easily opens the possibility to model Indian ragas. We define constraints as an error state; this allows a diagnostic mode of use where a putative piece can be tested against the rules. This is described in more detail in (Boenn et al. 2008).

Style Rules

The second level of rules relate to the stylistic rules of the type of music to be written. Our main concentration has

[1]Currently we are staying within the western musical tradition of twelve notes in an octave.

```
%% Melodic parts are not allowed to repeat notes
#const err_nrmp="No repeated notes in melodic parts".
reason(err_nrmp).
error(MP,T,err_nrmp) :- repeated(MP,T).

%% Dissonant contour
%% distance between lowest and highest note of melody
%% should not be a dissonant interval
#const err_dc="Dissonant contour".
reason(err_dc).
error(MP,TM,err_dc) :- lowestNote(MP,N1), highestNote(MP,N2),
                       chromatic(N1,C1), chromatic(N2,C2),
                       not consonant(C1,C2), N1 < N2,
                       partTimeMax(MP,TM).
```

Figure 2: A selection of the melodic rules of ANTON

```
%% Parts cannot cross over.
#const err_pcc="Parts can not cross".
reason(err_pcc).
error(P,T1,err_pcc) :- chosenNote(P,T1,N1), chosenNote(P+1,T2,N2),
                       N1 < N2, part(P+1), noteOverlap(P,T1,P+1,T2).

%% Parts can only ever meet at a single point, and this can
%% only happen once. P+1 is OK, because for P,P+N (N>1)
%% to meet have to have P,P+1 meet.
unison(P1,P2,T1) :- chosenNote(P1,T1,N), chosenNote(P2,T2,N),
                    P1 < P2, noteOverlap(P1,T1,P2,T2).
haveMet(P,T+1) :- unison(P,P+1,T), not haveMet(P,T),
                  part(P+1), partTime(P,T).
haveMet(P,T+1) :- haveMet(P,T), partTime(P,T).

#const err_cmmt="Parts can not meet multiple times".
reason(err_cmmt).
error(P,T,err_cmmt) :- unison(P,P+1,T), haveMet(P,T), part(P+1).
```

Figure 3: A few of the harmony rules from ANTON v2.0

been on Renaissance Counterpoint as described by Fux (Fux 1965 orig 1725) and Thakar (Thakar 1990). We allow up to four parts. Of interest here are the melodic rules, that capture concepts of tension and release. The coding of these was not as simple as some other components. This was not on the account of the language but defining what we meant. Some experimentation was needed to get them "right".

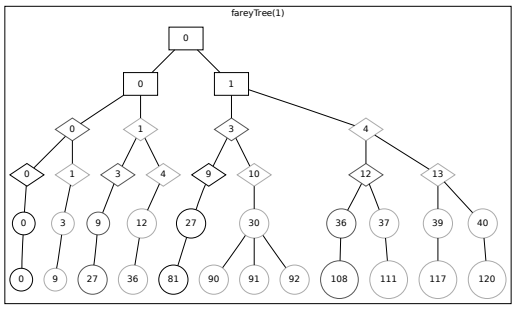
Multiple parts means that there must also be harmonic rules. These include rules that indicate parts do not cross and have limited unison, as shown in Figure 3.

At our current stage of development the separation between style rules and basic rules is not always clear. We hope and intend to codify the rules more carefully so users can be more selective in what is acceptable. We return to this in a later section.

The Problems of Rhythm

The earlier versions of ANTON generate multi-part pieces where the melodic rules governs the sequences of notes, and the harmonic rules ensure concords. The lacking musical element was rhythm, which we can think of as sequences of varying note durations governed by a characteristic metrical framework (London 2004). The addition of rhythm, the third component of music, to melody and harmony does introduce a new level of complexity in the rules, but also requires a new methodology for expressing rhythmic patterns.

Very little music plays all the notes for the same duration at the same times. There are simple rhythmic forms like a waltz or ländler where regular patterns of different lengths and emphasis are used, to more complex forms like swing (one part played at a slightly different speed) or syncopation




```
| 65 67 65 77 76 69 71 76 72 74 76 77
| F G F F′ E′ A B E′ C′ D′ E′ F′
| +2 −2 +12 −1 −7 +2 +5 −4 +2 +2 +1
| (((X X) (X X)) ((X (X X X)) ((X X) (X X))))
```

Figure 4: The partitioning tree and human readable format for a Lydian solo composition.

(one part playing different durations to the others).

Rhythm can be seen as the partitioning of a time interval into a number of simply related subdivisions, and possibly further partitioning of these subdivisions to some depth. In practice the divisions are typically equally sized parts. So it is clear that simple integer ratios are involved. One can order these partitions with the Farey Sequence (Farey 1816). It has been shown that the Farey sequence can be used as an analysis tool for rhythm (Boenn 2007). The Farey Sequence has been known for some time in the area of musical tuning systems[2]. Its use for rhythmic modelling has not been fully exploited yet; ANTON v2.0 is the first music application for composition where rhythms and musical forms will be generated on the basis of the principles outlined here. The Farey sequence of order n, denoted F_n, can defined as the sequence of reduced fractions in the range [0,1], when in lowest terms, have denominators less than or equal to n, arranged in order of increasing size. All music that depends on an underlying beat or pulsation can be represented using F_n with the elements denoting the normalised occurrence of musical events, e.g. note onsets.

As a computational extension of the Farey Sequence, we use a hierarchical tree of partitions. This is a computationally useful way of constructing a filtered Farey sequence, by removing unnecessary sub-divisions like for example larger primes. We have found that for early renaissance music there are three layers in this partitioning tree: one to represent the duration of the measures, one for the grouping of beats into metres and one for the subdivision of beats into individual notes and durations. The measure level indicates how many bars a part will contain and defines the height of the tree. The metre level defines the metrical structure of a particular bar and will contain the information on stress, beats and subdivisions. The metre layer defines the width of the tree. The subdivision layer groups individual notes together

[2]For example Erv Wilson's annotations of tunings used by Partch (Partch 1979) http://www.anaphoria.com/wilson.html

to determine their duration. The further advantage of having a tree is that it allows easy access to different metrical levels (measure, beats, subdivisions) which is vital for the later additions of rules about impact/resolution and for rules governing usage of consonance/dissonance.

An example of a partitioning tree can be found in Figure 4. Each part is divided in a number of measures, these form the top layer of the tree (rectangles). Each measure is then divided in a number of beats (diamond shapes), which control the emphasis of notes within a binary or ternary metrical pattern (the weight of shading). The subdivisions, or notes (circles), of the part are then grouped with respect to duration and placed within their respective metre. Individual notes are the leaves of the tree.

While each part has its own, but related rhythm, they do interplay. In order to apply harmonic rules it is necessary to relate the different notes in time. Allen (Allen 1983) gives the thirteen mutually exclusive possible relations between a pair of intervals (e.g. starting at the same time, meeting, overlapping, etc.) which can be used to express the relationships between notes.

In ANTON v2.0 we create a partitioning tree, where each node is a musical interval and the children are equal spaced subdivisions. The number of children corresponds to the order of the Farey sequence, each of which can be subdivided. While the code is written to allow any maximum order, for reasons of efficiency, and providing sufficient variety for our style of music, we restrict the order to be less than or equal to 3, encoded by `possibleExpansion`. This filtering provides us with a sufficiently rich subset of trees (and hence rhythms) for our musical genre.

The necessity to ground over the entire domain means that all possible nodes of a tree are instantiated whether they are needed or not. Since this is inevitable we use a ghost tree that is expanded to the maximum order, in this case 3, and label each node either active or not. So branches are rather pruned than generated. This removes some verification and is significantly faster. The code uses the `present` predicate to indicate active nodes.

As mentioned earlier, the rhythm tree for our style has three duration levels (measure, metre and subdivisions) with each having its own set of rules, allowing us to specify stress, chords and time signatures. While for each part an individual note duration layer is created, all parts will have the same structure for the measure and metre layer. Without this constraint no consistency in the piece can be guaranteed.

For each part, we have a one-to-one mapping, via the `timeToNote` predicate, between the leafs of the associated tree, the notes, and the timing in the part, such that the melodic rules and harmonic rules can be applied.

To model the interplay between the various parts, to allow the harmonic rules to take effect and to create chords it is necessary to be able to relate notes across parts, considering that each part has its own time-line and notes have different durations. Previously, we mentioned that Allen (Allen 1983) developed rules to relate pairs of temporal intervals. In practice these rules are very complex to express in *AnsProlog*. The main reason for this difficulty is that current solvers require programs to be instantiated before answer sets can be computed. The 13 different relations over a large domain would result in an extremely large grounded program which current solvers cannot cope with. For this reason we have used a simplification of the Allen intervals, using a single predicate, `noteOverlap` which is true if there is any instant when both its arguments are extant. The `noteOverlap` predicate combines nine (begins, contained, overlaps, ends and their converses plus same) of the different Allen relations. This is are sufficient for the style of music that we are currently modelling. Coupled with the simplified tree this creates a realisable method to encode rhythm. `noteOverlap` uses two iterations to obtain a more compact grounding in terms of grounded rules although more atoms are needed.

Just like the encoding for melody and harmony, rhythm is encoded using error predicates to allow for diagnosis and debugging.

Figure 5 contains a small selection of rhythm rules. They have been selected to demonstrate some of the intricacies of encoding rhythm which is a significantly more complex than the encoding of the other components. The first rule is part of the construction of the Farey tree for a particular part. Its depth is determined by the depth of the various layers. Note that the `measureDepth` is tree independent. Each level has its own set of rules of construction. Figure 5 shows one for the metre level, showing that the level depends on the duration depth. To demonstrate the pruning of the fully expanded ghost tree, a rule was included stating that descendants can only be present if they do not go beyond the expansion determined for that layer. The following rules show how `timeToNode` is encoded using the auxiliary predicate `nodeStep` to fully take into account the duration of notes. The next rule shows that the various musical elements can be easily accessed through the various layers of the tree. The rule determines the note strength as determined by the metre layer. The remaining rules show the double iteration of determine `noteOverlap`.

To ensure backwards compatability a special rule set, `monorhythmic`, was created to provide a default implementation for `noteOverlap` stating that two notes overlap if their time steps match and allowing each note to be part of a possible chord.

Assemblage

As can be seen from the description above the system has a number of stages. This is controlled by a collection of Perl scripts. The first of these, `programBuilder.pl`, selects the rules required from the base files and sets a number of constants such as length of piece, key and time signature. While in principle one could include all the rules, efficiency considerations motivate this selection. For example, if one is writing a duet the rules for trios and quartets just increase the rule base and hence memory.

The next stage of the process is to ground the rules, and create answer sets. Each answer set in our application corresponds to a new musical piece that satisfies the rules. Controls exist for how many solutions are required and some randomness controls to look at different parts of the solution

```
%% Each Farey tree has a given depth
depth(F,MD + BD + DD) :- measureDepth(MD), meterDepth(F,BD),
                         durationDepth(F,DD).

%% The rules for the meter layer
meterLevel(F,FL) :- depth(F,DE), level(F,FL),
                    durationDepth(F,DD), meterDepth(F,BD),
                    FL <= (DE - DD), FL > (DE - (DD + BD)).

%% Only descendants less than the expansion are present
present(F,FL2,ND2) :- expand(F,FL1,ND1,E),
                      descendant(F,D,FL1,ND1,FL2,ND2),
                      D < E, possibleExpansion(F,E).

%% Map from nodes to time positions
%% Mapping increments each time a node is present
nodeStep(F,0,1).
nodeStep(F,ND,T) :- not present(F,DLL,ND), nodeStep(F,ND-1,T),
                    node(F,DLL,ND), durationLeafLevel(F,DLL),
                    partToFareyTree(P,F), partTime(P,T), ND > 0.
nodeStep(F,ND,T+1) :- present(F,DLL,ND), nodeStep(F,ND-1,T),
                      node(F,DLL,ND), durationLeafLevel(F,DLL),
                      partToFareyTree(P,F), partTime(P,T), ND > 0.

%% From this we derive a unique mapping from node to time step
timeToNode(P,1,0).
timeToNode(P,T,ND) :- present(F,DLL,ND), nodeStep(F,ND-1,T-1),
                      node(F,DLL,ND), durationLeafLevel(F,DLL),
                      partToFareyTree(P,F), partTime(P,T), ND > 0.


%% Meter strength is created at the first level of the meter layer
nodeMeterStrength(F,MLL+1,ND2,1) :- measureLeafLevel(F,MLL),
                                    node(F,MLL,ND1),
                                    descendant(F,0,MLL,ND1,MLL+1,ND2).

timeToNodeOverlap(P1,T1,F2,ND2) :- timeToNode(P1,T1,ND1),
                                   partToFareyTree(P1,F1),
      nodeOverlap(F1,ND1,F2,ND2), F1 < F2.
noteOverlap(P1,T1,P2,T2) :- timeToNodeOverlap(P1,T1,F2,ND2),
                            timeToNode(P2,T2,ND2),
    partToFareyTree(P2,F2), P1 < P2.
```

Figure 5: A small excerpt from the rhythm rule set of ANTON v2.0

space. At present we are searching for a valid solution, not necessarily the best (see later under future work),

These answer sets are just statements of what is the case. We supply a further Perl script to interpret these as text, Lilypond (Nienhuys and Nieuwenhuizen 2003) input, Csound (Boulanger 2000) csd files or an internal format for further processing. This process is shown in Figure 6.

What this does not show is that ANTON can also complete partial pieces, that the Perl scripts allow for the fixing of certain notes at defined times or that complete pieces can be verified. Indeed there are many other options.

In order to make it more accessible to non-technical users a simple FLTK-based GUI has been created to select the options, and then run the scripts. An example of its use is shown in Figure 7.

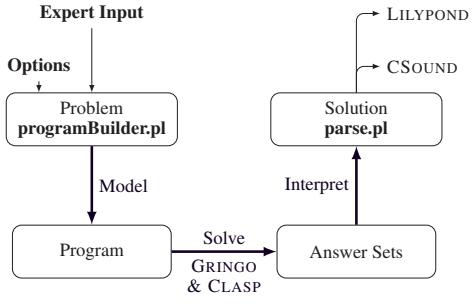


Figure 6: A diagrammatic representation of ANTON

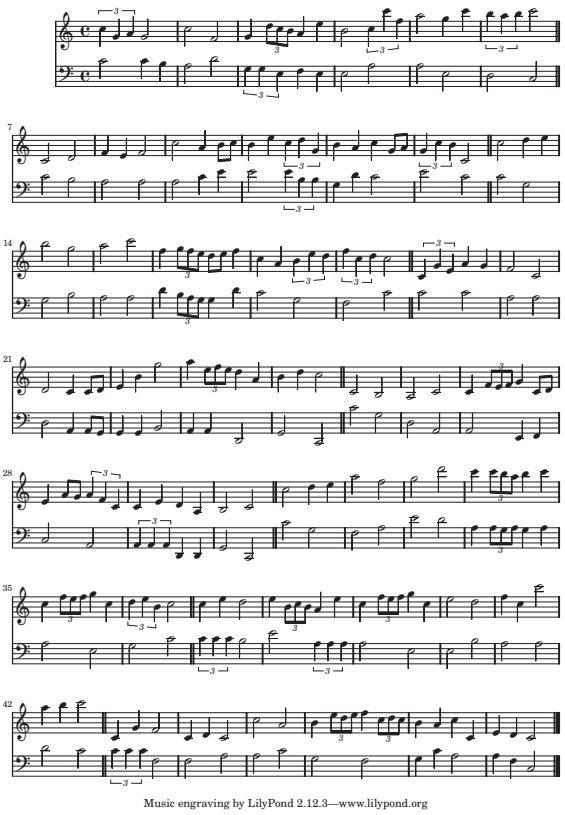


Figure 8: Extract from *Wedding on 2012 6 23*

It should be noted that if the inputs for a piece do not permit any solution, ANTON returns no answers. This response should lead to the operator changing or relaxing the parameters. Experience has shown that a small number of measures often leads to no solution, while large numbers can have large computing time.

Sonic Results

We have not completed a systematic investigation into people's response to the pieces created. Informally it certainly creates short acceptable music, and at times more than that. The two commonest output forms are as Csound scores or Lilypond engravings. A number of audio WAV files can be found in ANTON's web site http://dream.cs.bath.ac.uk/Anton. So pending formal listening tests readers can make their own decisions. Here we present some examples of the scores.

Figure 8 shows an extract from a recently completed occasional piece[3] with rhythm enabled. The whole piece was arranged from three different ANTON option-sets, minor and major sections and the rhythmic ending.

[3]Full version is at http://cs.bath.ac.uk/jpff/Wedding.html

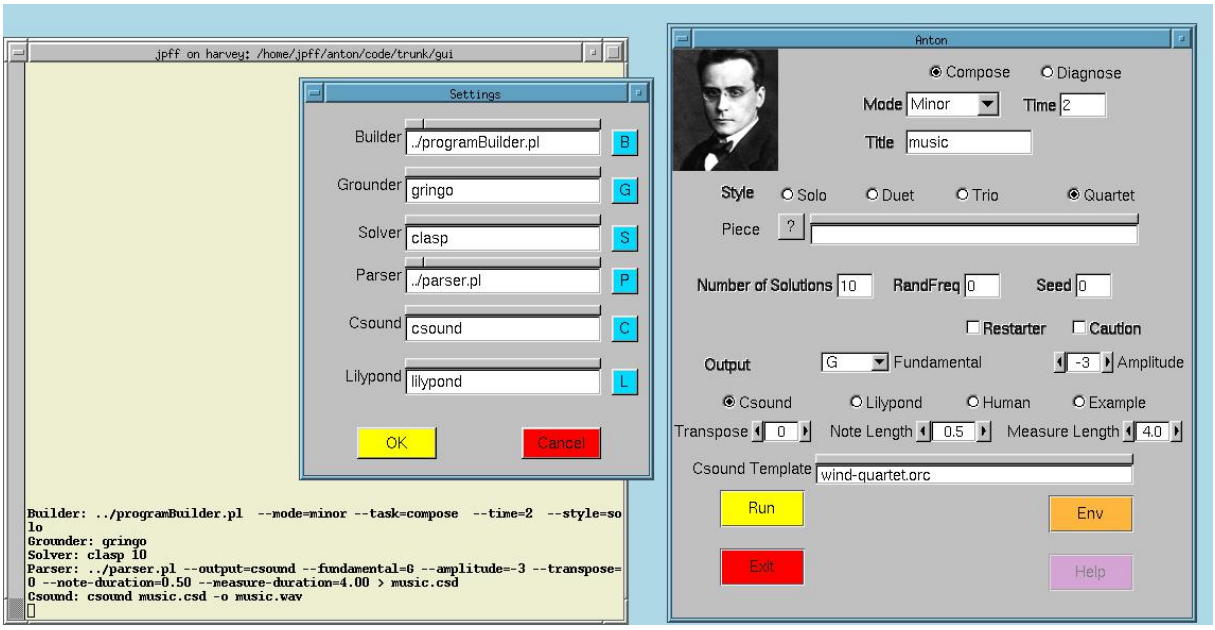


Figure 7: The ANTON GUI in use

We are currently limited to solos, duets, trios and quartets. In Figure 9 we show a more complex piece in four parts.

Future Work

There are many tasks that wait to be done.

We are acutely aware that we need to make a systematic listener-based survey of the acceptability of the pieces written, and also a comparison with humanly written pieces in the same genre. We plan to do this soon, but we need to identify a machine-readable source of comparators, as well as constructing the necessary web-based survey. Not having direct access to a music department we will have to rely on yet another net-based sample of participants.

Our current rule organisation does not make strong distinctions between special rules and more general ones. We have sketched an organisation that would allow for more composer-selection of sets of rules, but this remains to be done together with a usable interface. There are a number of similar organisational tasks that could occupy us.

Perhaps the most pressing direction that is needed is to implement style rules for some other genre of music. We know of an independent attempt to use ANTON to create trance music, but in the light of the name[4] a collection of serial rules would be very welcome. Looking at Hindustanni music is also an interesting idea. Codification of such rules requires the interaction of a suitable musician and a logic programmer; at least that is how our initial rule-set was created. We prefer this approach to say a direct machine-learning one, as the rules themselves are of interest, and permit alternative ways of interpreting them.

[4]The name came from a short conversation with a mathematician who opened the conversation with "Who is your favourite composer of the second Viennese school?"

Elsewhere we have investigated the use of software agents to perform music(ffitch and Padget 2002), and it would be good to move these two projects to an integration.

What ANTON can do at present is construct melodic and harmonic fragments, but it only has rules for local structure. For example in the composition of the work in Figure 8 an number of six-measure duets were created and the human selected and discarded, and added other variations, like a minor key section. What is clear is that consecutive answer-sets tend to be similar, and so a weak variation structure appears. But music has a clearer large scale structure.

We have not solved this difficult global structure problem but we have a starting point from which we can build a system that is hierarchical over time scales; we have a mechanism for building syntactically correct sentences, but these need to be built into paragraph and chapters, as it were. It is not clear if this will be achieved within the current ASP system, or by a procedural layer built on top if it, or some other scheme such as those of Leach (Leach 1999).

Another significant problem arises in extending rhythm. The Allen rules (Allen 1983) are more complex than those we have used, and the numerical nature of these rules is a challenge to the symbolic logic system. We postulate that a hybrid system. with $AnsProlog$ calling out to a more traditionally coded predicate may be necessary. Recent solvers incorporate such a scheme. Even then there remains much to do in terms time and rhythm.

An intriguing possible avenue of exploration is to see if human judgement on resulting pieces can be synthesised into ASP rules, and so develop some mechanism for aesthetic choice – but this is just speculation.

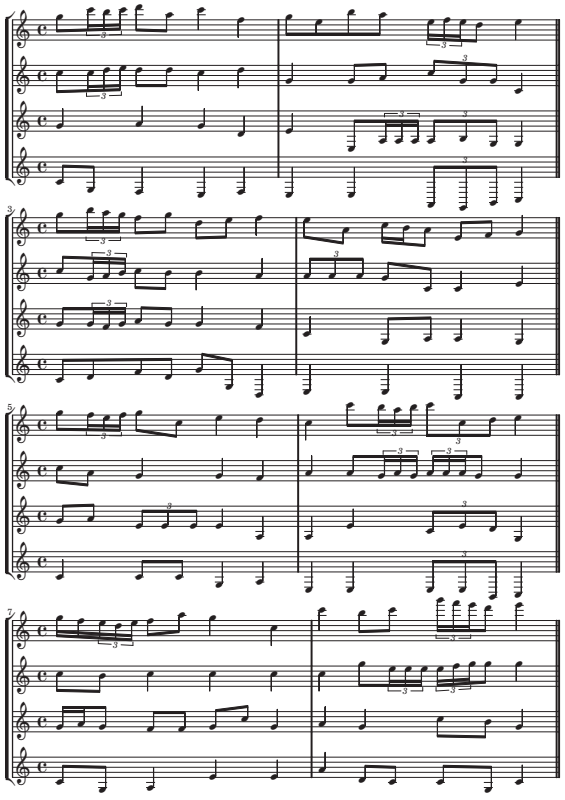


Figure 9: Example of a quartet

Conclusions

We have presented progress in automated composition based on a rule-based system, and shown that it can create acceptable musical pieces.

The initial system was fairly simple to create; a composer and a logician sitting together defined the foundation very rapidly. This emphasises our belief that ASP is a good mechanism for knowledge capture. Developing the reasoning rules was a little harder, and required iterations to check that the concept had been encompassed, composing many pieces to see if they were as expected.

We have made some progress with modelling rhythm, but we are aware that our current scheme has limitations, even if it is acceptable for our current style of renaissance polyphony. To extend this approach we need to implement the Allen rules for time relationships, and that will require some re-engineering in order to be able to use external predicates. Despite the limitations we feel this is a useful way to model rhythm, and merits further work (Boenn 2011).

ANTON is one of the largest ASP applications currently reported, and as such has resonance in that community. Indeed this project is multidisciplinary in all its aspects, encompassing a range of areas from musicology, logic and computing.

In conclusion we wish to stress that this project is open source and freely available[5]. We invite anyone who wishes to use or extend the system, develop new style rules, especially for genres not in our experience, and/or contribute to this project of computational music theory, attempting to understand from a computational aspect what it is that makes music music.

References

Allen, J. F. 1983. Maintaining Knowledge about Temporal Intervals. *CACM* 26:198–3.

Anders, T. 2007. *Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System*. Ph.D. Dissertation, Queen's University, Belfast, Department of Music.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 1st edition.

Bel, B. 1998. Migrating Musical Concepts: An Overview of the Bol Processor. *Computer Music Journal* 22(2):56–64.

Boenn, G.; Brain, M.; Vos, M. D.; and ffitch, J. 2008. Anton: Answer set programming in the service of music. In Pagnucco, M., and Thielscher, M., eds., *Proceedings of the Twelfth International Workshop on Non-Monotonic Reasoning*, 85–93. NMR2008. Also Tech report UNSW-CSE-TR-0819.

Boenn, G.; Brain, M.; De Vos, M.; and ffitch, J. 2009. Anton: Composing logic and logic composing. In Erdem, E.; Lin, F.; and Schaub, T., eds., *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, 542–547. Potsdam, Germany: Springer.

Boenn, G. 2007. Composing Rhythms Based Upon Farey Sequences. In *Digital Music Research Network Conference*.

Boenn, G. 2011. *Automated Analysis and Transcription of Rhythm Data and their Use for Composition*. Ph.D. Dissertation, University of Bath, Department of Computer Science.

Boulanger, R., ed. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press.

Denecker, M., and Ternovska, E. 2008. A Logic of Non-Monotone Inductive Definitions and its Modularity Properties. *ACM Transactions on Computational Logic (TOCL)* 9(2).

East, D., and Truszczyński, M. 2001. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI2001*, volume 402 of *LNAI*. Springer Verlag.

Farey, J. 1816. On a curious property of vulgar fractions. *Philosophical Magazine*.

ffitch, J., and Padget, J. 2002. Learning to play and perform on synthetic instruments. In Nordahl, M., ed., *Voices of Nature: Proceedings of ICMC 2002*, 432–435. School of Music and Music Education, Göteborg University: ICMA.

Fux, J. 1965, orig 1725. *The Study of Counterpoint from Johann Joseph Fux's Gradus ad Parnassum*. W.W. Norton.

[5] http://dream.cs.bath.ac.uk/Anton

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. Conflict-Driven Answer Set Solving. In *Proceeding of IJCAI07*, 386–392.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. GrinGo: A New Grounder for Answer Set Programming. In Baral, C.; Brewka, G.; and Schlipf, J., eds., *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*, 266–271. Springer-Verlag.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9(3-4):365–386.

Huron, D. 2006. *Sweet Anticipation. Music and the Psychology of Expectation*. MIT Press.

Leach, J. L., and Fitch, J. P. 1995a. The application of differential equations to the modelling of musical change. In *ICMC'95: Digital Playgrounds*, Banff, Canada, 440–443. ICMA and Banff Centre for the Arts.

Leach, J. L., and Fitch, J. P. 1995b. Nature, music and algorithmic composition. *Computer Music Journal* 19(2):23–33.

Leach, J. L. 1999. *The Aesthetics of Context-Variant Form*. Ph.D. Dissertation, University of Bath, School of Mathematical Sciences.

London, J. 2004. *Hearing in Time. Psychological Aspects of Musical Meter*. Oxford University Press. ISBN 978-0-19-516081-9.

Nienhuys, H.-W., and Nieuwenhuizen, J. 2003. Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*.

Partch, H. 1979. *Genesis of a Music*. New York: Da Capo Press.

Thakar, M. 1990. *Counterpoint*. New Haven.